Preface

This report comprises the contributions to the second workshop on Algebra of Communicating Processes: ACP95. The workshop is meant to provide an overview of the (current) research, design and application activities in the area of ACP. The workshop ACP95 is the successor of ACP9594 which was held at Utrecht University, May 16–17, 1994.

In this report, twenty-one contributions are included. Two of these are classical papers on process algebra. The first one is a technical report written by R.J. van Glabbeek in 1986, and describes an analysis of the methodology of some theories of concurrency (mainly CCS and CSP). The second classical paper is written by J.L.M. Vrancken (based on a 1986 report) and presents a first systematic approach to ACP with empty process. Furthermore, each of the four invited speakers submitted a paper, and the programme committee selected fifteen contributions for presentation at ACP95. However, the paper written by J.J. van Wamelen was not presented, but is included here.

The organizing committee wishes to express her gratitude to Eindhoven University of Technology, the University of Amsterdam, and Utrecht University for financial support to ACP95. Alda Bouten (Eindhoven University of Technology) is thanked for all her efforts.

Finally, the editors gratefully acknowledge the programme committee and subreferees for their careful reviewing.

The editors,

Invited Speakers

J.A. Bergstra (University of Amsterdam & Utrecht University)
M.A. Bezem (Utrecht University)
C.A. Middelburg (KPN Research & Utrecht University)
G.J. Veltink (DeTeWe Kommunikationsysteme)

Programme Committee

J.C.M. Baeten (Eindhoven University of Technology)
R.J. van Glabbeek (Stanford University)
A.S. Klusener (CWI)
J.C. Mulder (Eindhoven University of Technology)
A. Ponse (University of Amsterdam)
C. Verhoef (Eindhoven University of Technology)
S.F.M. van Vlijmen (Utrecht University)

Organizing Committee

A. Ponse (University of Amsterdam)
C. Verhoef (Eindhoven University of Technology)
S.F.M. van Vlijmen (Utrecht University)
# Table of Contents

Notes on the methodology of CCS and CSP  
*R.J. van Glabbeek* ................................................................. 1

The Algebra of Communicating Processes with empty process  
*J.L.M. Vrancken* ................................................................. 23

Alternative implementation strategies for process algebra tools  
*G.J. Veltink* ........................................................................ 63

A general conservative extension theorem in process algebras with inequalities  
*P.R. D'Argenio* ................................................................. 67

A well-formedness checker for μCRL  
*J.A. Hillebrand and H.P. Korver* ........................................ 81

Formalizing process algebraic specifications in COQ  
*M.A. Bezem, R.N. Bol, and J.F. Groote* ............................. 105

Two finite specifications of a queue  
*M.A. Bezem and A. Ponse* .................................................. 121

Prefix iteration in basic process algebra: applying termination techniques  
*W.J. Fokkink and H. Zantema* ............................................. 139

A verification of the Bakery Protocol combining algebraic and model-oriented techniques  
*C. Brovedani and A.S. Klusener* ........................................ 157

Process algebra with autonomous actions  
*T. Basten and M. Voorhoeve* .............................................. 181

Axiomatizing early and late input by variable elimination  
*A. van Deursen* ................................................................... 195

Process algebra with propositional signals  
*J.C.M. Baeten and J.A. Bergstra* ....................................... 213

A simulation of a CIM factory using a parallel object oriented language  
*A. van Delft and L. Stolk* ................................................... 229

Fischer's protocol in timed process algebra  
*J.J. Vereijken* ...................................................................... 245

Formal verification of a leader election protocol in process algebra  
*J.F. Groote, L. Fredlund, and H.P. Korver* .......................... 285

Process algebra semantics of φSDL  
*J.A. Bergstra and C.A. Middelburg* .................................... 309
TSF, a Test Specification Formalism
M. Huizer and L.G.M. Feijs .................................................. 347

Empty interworkings and refinement, semantics of interworkings revisited
S. Mauw and M.A. Reniers .................................................. 367

Generating FSMs from interworkings
L.G.M. Feijs ................................................................. 387

A 3–level algebraic Ethernet specification
J.J. Brunekreef ............................................................... 413

Process Algebra with Language Matching
J.J. van Wamel ............................................................... 429

Author Index ................................................................. 451
Notes on the Methodology of CCS and CSP

R.J. van Glabbeek
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In this paper the methodology of some theories of concurrency (mainly CCS and CSP) is analysed, focusing on the following topics: the representation of processes, the identification issue, and the treatment of nondeterminism, communication, recursion, abstraction, divergence and deadlock behaviour. Process algebra turns out to be a useful instrument for comparing the various theories.

1980 Mathematics Subject Classification: 68B10, 68C01, 68D25, 68F20.

Key words & phrases: concurrency, CCS, CSP, process algebra.

Note: Sponsored in part by Espri project no. 432, Meteor (An Integrated Formal Approach to Industrial Software Development).

1. Introduction: Theories of Concurrency

This is an investigation into the methodology of some theories of concurrency. In general a concurrency theory offers a framework for the specification (or even the design) of parallel processes and the verification of statements about them. The features of concurrency, expressible within such a framework, include communication between parallel processes, deadlock behaviour, abstraction from internal steps, fairness, nondeterminism, priorities in the choice of actions, tight regions, etc.

Some interesting theories of concurrency are:
- The theory of Petri Nets (see for instance Reisig [16])
- Trace theory (see for instance Rem [17])
- Milner's Calculus of Communicating Systems (CCS) ([11])
- Hoare's theory of Communicating Sequential Processes (CSP) ([9])
- The topological process theory of De Bakker & Zucker ([3,4])
- The Algebra of Communicating Processes (ACP) of Bergstra & Klop ([5]).

This paper will be mainly devoted to CCS and CSP.
Table of contents

1. Introduction: theories of concurrency
2. Models and calculi
3. How to represent a process
   3.1 Models of concurrency
   3.2 Atomic actions
   3.3 Trace sets
   3.4 Failure sets
   3.5 State transition diagrams
   3.6 Operational semantics
4. When to identify processes
   4.1 Why identify processes?
   4.2 How to identify processes
   4.3 Bisimulation semantics
   4.4 Trace semantics
   4.5 Failure semantics
   4.6 Ready trace semantics
   4.7 Survey
5. Features of concurrency
   5.1 Nondeterminism
   5.2 Communication
   5.3 Recursion
   5.4 Abstraction
   5.5 Divergence
   5.6 Deadlock behaviour
6. Survey of CCS and CSP
   6.1 An operational semantics for CCS
   6.2 An operational semantics for CSP
   6.3 Equivalences on process expressions
   6.4 Axioms for CSP
   6.5 Axioms for CCS
   6.6 Axioms for identification of CCS expressions
7. References
2. Models and Calculi

A framework for studying concurrency often has the shape of a mathematical model. Parameters in the classification of these models are the features captured by the model, the identifications made on processes and the particular way of representing them. These criteria (in reverse order) will be explained and applied in the next three sections.

Apart from being a mathematical model, the framework in question can also be a calculus for the verification of statements about processes, formulated in an algebraical language. For practical applications this means that instead of checking that a process fits into a selected model, one has to check that it operates in an environment where the rules and axioms of the calculus are satisfied.

Some theories of concurrency use both models and calculi, but with different emphasis on one of those. This provides an important criterion for method decomposition.

The theory of Petri nets establishes a model of concurrency, without a calculus, and so does the topological process theory.

Trace theory also presents a model, but a number of calculi, axiomatising this model, have been developed, starting with Kleene [10] and Salomaa [18].

CSP, as presented in Brookes, Hoare and Roscoe [7] and in Hoare [9], provides a model, illustrated with some algebraical laws. Systematic axiomatisations of CSP can be found in Brookes [8] and De Nicola [13].

CCS is essentially a calculus, but the rules and axioms in this calculus are presented as laws, valid in a given model.

ACP is a calculus that is not bound to a particular model. It is the core of a family of axioms systems, each describing some features of concurrency.

The systematic exploration of (families of) algebraical calculi is called process algebra. In process algebra models are merely used as illustration and for constructing consistency proofs. This model-independence makes process algebra, apart from a tool for studying concurrency directly, also suitable for analysing the different models: the presentation of axiomatisations illuminates their differences and similarities. Axioms for CCS and CSP and for the identification criteria discussed in §4, will be presented in §6. Most of them are taken from [8], [9], [11] and [13].

3. How to represent a process

3.1 Models of concurrency

As can be extracted from the previous section, five of the six concurrency theories mentioned in §1, work with an explicit model. In all these models processes are represented differently. In Net theory one of the ways to represent a process is as a labeled Petri net with a given initial configuration. De Bakker and Zucker use a topological construction to represent processes. In trace
theory a process is represented by a \textit{trace set} and in CSP by a \textit{failure set}; both these concepts will be explained below. Milner represents a process by a \textit{synchronisation tree}. This is the same (though slightly less general) as what is known as a \textit{state transition diagram} or \textit{process graph}, and will be explained in section 3.5. In [5] three models of ACP are mentioned: its initial algebra, a \textit{projective limit} model (resembling the topological construction of De Bakker and Zucker) and a process graph model.

3.2 Atomic actions

In all concurrency theories mentioned in this paper, the most elementary components of a process are the so-called \textit{atomic actions}. They are indivisible and not subject to further investigations. Now a process just performs atomic actions $a,b,c,...$ out of a given alphabet $A$.

3.3 Trace sets

In trace theory a process is considered to be fully determined by the possible sequences of atomic actions it can perform (its \textit{traces}). Therefore a model is created in which a process is represented by the set of its traces. Usually trace sets are required to be prefix closed and to contain only finite traces of infinite processes. In this setting any non-empty prefix closed set of finite words over $A$ represents a process.

3.4 Failure sets

In CSP a process is considered to run in an environment which can veto the performance of certain atomic actions. Moreover the environment can decide to do so during the execution of a process. If, at some moment in the execution, no action in which the process is prepared to engage is allowed by the environment, then deadlock occurs, which is considered to be observable. Now, a finite experiment with a process yields either a trace, or a trace followed by deadlock. In the last case the trace $\sigma \in A^*$ may be recorded, as well as the set $X \subseteq A$ of actions allowed by the environment at the time of stagnation. An element of $X$ is said to be refused by the process and $X$ is called a \textit{refusal set} of the process after performance of $\sigma$. Now the pair $\langle \sigma,X \rangle$ is a \textit{failure pair} of the process and the set of all failure pairs of a process is called its \textit{failure set}. Since in CSP a process is considered to be fully determined by the observations obtainable from all possible finite experiments (as described above) with the process, a failure model of CSP is created in which a process is represented by its failure set. In this model any set $F \subseteq A^* \times \text{Pow}(A)$ satisfying

1. $\langle \varepsilon, \emptyset \rangle \in F$
2. $\langle \sigma \cdot \rho, \emptyset \rangle \in F \Rightarrow \langle \sigma, \emptyset \rangle \in F$
3. $\langle \sigma, Y \rangle \in F \land X \subseteq Y \Rightarrow \langle \sigma, X \rangle \in F$
4. $\langle \sigma, X \cup Y \rangle \in F \Rightarrow \langle \sigma, X \cup Y \rangle \in F$

represents a process. Here $\varepsilon$ denotes the empty trace, $\emptyset$ the empty set and $\sigma \cdot \rho$ the concatenation of the traces $\sigma$ and $\rho$. 

4
3.5 State transition diagrams

In CCS a process is considered to go through a number of states. The states are determined by the possible courses of action the process is ready to engage in. In a state transition diagram the states of a process are pictured as open dots (\(\bullet\)): the nodes of a process graph. Any action \(a \in A\) the process can perform is regarded as a state transition from the state of the process before performance, to the state after. Such a state transition is pictured as an arrow between these two states, labeled by \(a\): an edge of the process graph. If a process passes to another state, without performing an (observable) action, the corresponding state transition in the diagram is labeled by \(t \notin A\) (the invisible action, or \(t\)-step). If a process can remain in a state without terminating, then there is a \(t\)-step from this state to itself (a \(t\)-loop or delay). Finally the initial state in the diagram is denoted by a short arrow (\(\rightarrow \bullet\)): the root of the process graph. Now in the graph model of CCS a process is represented by its state transition diagram, and any state transition diagram over \(A \cup \{t\}\) represents a process. However, different state transition diagrams may represent the same process: two processes are identified if there exists a bisimulation between their state transition diagrams \(g\) and \(h\). This is a binary relation \(R\) between the states of \(g\) and \(h\), containing the pair of roots, such that if \((s, t) \in R\) and \(s \xrightarrow{a} s'\) is an edge in \(g\) then there is an edge \(t \xrightarrow{a} t'\) in \(h\), with the same label \(a \in A \cup \{t\}\), such that \((s', t') \in R\), and, vice versa, if \((s, t) \in R\) and \(t \xrightarrow{a} t'\) is an edge in \(h\), then there is an edge \(s \xrightarrow{a} s'\) in \(g\) with \((s', t') \in R\).

This identification criterion is what Milner calls strong congruence (although his first definition of strong congruence (in [11]) was slightly different). In [11] Milner expresses the wish to identify also processes which are not strongly congruent. Then a process is modeled as an equivalence class of state transition diagrams, under an equivalence relation containing strong congruence. The appropriate equivalences are discussed later.

3.6 Operational semantics

In a calculus processes are represented by process expressions, built from the constants and operators in the language. This representation differs from the model representations in two ways: different expressions may represent the same process, and some processes may have no process expression representation.

The initial algebra of a theory is the set of closed process expressions modulo provable equality (if the theory is an algebraical calculus, then provable equality is always a congruence). If the language used is sufficiently expressive and the calculus complete for closed terms (with respect to an intended interpretation), then the initial algebra models the finite processes, i.e. any finite process is represented by exactly one equivalence class of process expressions. In CCS and CSP also a recursion operator \(\mu\) is present, enabling the construction of process expressions representing infinite processes. In the presence of such an operator the idea of the initial algebra can be generalised, and the set of closed process expressions modulo provable equality again constitutes a model of concurrency. However, in the absence of a complete calculus (with respect to an intended interpretation), this model does not make enough identifications.

This asks for a coarser equivalence on process expressions. Such an equivalence can be obtained by the general method of endowing languages with operational semantics: For any label
\(\ni \in \mathcal{U}\{\tau\}\) define a binary relation \(\xrightarrow{\tau}\) on the set of process expressions, in such a way that \(E \xrightarrow{\tau} F\) means that the process represented by the expression E may perform an \(\tau\)-step, thereby changing into a process that can be represented by the expression F. This makes the domain of process expressions into a state transition diagram (however without a root). From this universal state transition diagram the diagram of a particular expression can be obtained by appointing this expression as root of the diagram.

Since this approach identifies process expressions, processes and states, a bisimulation can be defined as a relation on process expressions, and strong congruence is just the union of all bisimulations. Now a model of closed CCS-expressions modulo strong congruence can be constructed, which is more satisfying than the generalized initial algebra approach of CCS-expressions modulo provable equality. In §6 an operational semantics for both CCS and CSP will be presented.

4. When to identify processes

4.1 Why identify processes?

As remarked in §1, one of the purposes of a concurrency theory is to verify statements about processes. Such a statement can be that a certain system correctly simulates a specified process. In that case the theory has to determine whether the two processes (i.e. the real and the intended behaviour of the system) are equal. This asks for a criterion for identifying processes. Such a criterion determines (partly) the semantics of the theory. The choice of a suitable semantics may depend on the tools an environment has, to distinguish between certain processes. It is conceivable that a concurrency theory is equipped with different semantics, and has the capacity to express equality on different levels.

4.2 How to identify processes

In the various concurrency theories different identification strategies have been pursued. In particular CCS identifies much less than CSP. An advantage of identifying more is that it becomes easier to verify statements in which processes are equated. All true statements \(x=y\) remain true after identifying more. However, one might identify too much, depending on the discriminating capacity of an environment. In particular the identification of two processes that cannot be distinguished with a given set of tools, disables the development of a new tool to distinguish them. Algebraically this means that some operators in a language of concurrency (which correspond to tools that distinguish between processes) are incompatible with some identifications. Moreover some useful conditionals axioms might get lost, because after making certain identifications the premisses of the axioms are true too often.
4.3 Bisimulation semantics

In CCS processes are identified only if there is no environment conceivable in which they can be distinguished. In each case processes should be identified if they are strongly congruent, in the sense of section 3.5. However, Milner identifies some more processes, only differing in their invisible steps. In [11] he proposes the notion of observation equivalence. Later he uses a slightly different version of observation equivalence (see [12]), adapted to the notion of bisimulation, as proposed by Park in [15]. In §6 the syntax of CCS is presented, together with an operational semantics, including the definition of observation equivalence. The basic operators of CCS are the constant 0, the unary operators a (for a ∈ A) and the binary operator “.”. 0 represents the process, unable of doing anything at all; aP represents the process, which will first perform an a-step and then proceeds with P; and P + Q represents the process, which first makes a choice between P and Q, and then proceeds with the chosen process. This is illustrated by the following state transition diagrams:

[Diagrams of state transition diagrams]

In bisimulation semantics the processes a(b0 + c0) and ab0 + ac0 are considered to be different. A motivation for this can be found in the different timing of the choice between b and c. Moreover, if they are placed in an environment that will not allow the execution of c, then they can be distinguished by observation: ab0 + ac0 has the possibility to deadlock after execution of a, while a(b0 + c0) has not: here a will always be followed by b.

Algebraically, such an environment is represented by the restriction operator \c (see §6; on process graphs \c removes all c-edges (as well as the disconnected parts that originate)). Now (ab0 + ac0)\c = ab0 + a0, while (a(b0 + c0))\c = ab0.

In [11], Milner remarks about observation equivalence that "two behaviour expressions should have the same interpretation in the model iff in all contexts they are indistinguishable by observation". However, he gives no clue, how one goes about distinguishing between abc0 + abd0 and ab(0 + c0 + b0) by observation (or it must be that the states of a process are considered to be directly observable). In any case, it cannot be done by any of the CCS operators.
4.4 Trace semantics

In trace theory much more processes are identified than in CCS. By defining trace equivalence on process graphs, it is possible to compare trace semantics with bisimulation semantics. Definition: \( \sigma \in A^* \) is a trace of a process graph \( g \), if there is a finite path in \( g \), starting at the root, with label \( \sigma \). Here the label of a path is the sequence of labels of the composing transitions, where all \( \tau \)-labels are dropped. Now the trace set of \( g \) is the set of its traces, and two process graphs are trace equivalent iff they have the same trace sets. Remark that the model of trace sets of section 3.3 is isomorphic to the model of process graphs modulo trace equivalence. Since any two observation equivalent processes are also trace equivalent, trace equivalence is called a coarser (= less discriminating) equivalence than observation equivalence. This is pictured in section 4.7.

In the setting of trace theory, presented in section 3.3 (or above), no deadlock behavior is displayed. Not only the processes \( abO + acO \) and \( a(bO + cO) \) are identified (both have trace set \{e, a, ab, ac\}) but also the processes \( abO + aO \) and \( abO \) (both have trace set \{e, a, ab\}). However, \( abO + aO \) can deadlock after performing an a-step, while \( abO \) cannot. If deadlock is considered to be observable, a modification of trace theory can be made, in which also traces ending on 0 are allowed. In that setting \( abO + aO \) has trace set \{e, a, aO, ab, abO\} while \( abO \) has trace set \{e, a, ab, abO\}. Call the corresponding equivalence on graphs \( \partial \)-trace equivalence.

4.5 Failure semantics

In \( \partial \)-trace semantics, where deadlock is observable, the processes \( abO + acO \) and \( a(bO + cO) \) are equal (both have \( \partial \)-trace set \{e, a, ab, abO, ac, acO\}). However, the processes \( (abO + acO)\c = abO + aO \) and \( (a(bO + cO))\c = abO \) are different (as remarked above). So Milner’s restriction operators \( \c (\c \in A) \) are incompatible with \( \partial \)-trace semantics. If an environment is equipped with restriction as a tool for analysing processes, then a finer equivalence is needed to model the results of this analysis. As suggested previously, in section 3.4, failure semantics is adequate for restriction and deadlock behavior.

A tuple \(<\sigma, X>\) with \( \sigma \in A^* \) and \( X \subseteq A \) is a failure pair of a process graph \( g \) if there is a path from the root of \( g \) to a node \( p \) with label \( \sigma \), such that the set \( l(p) \) of labels of the outgoing edges of \( p \), is contained in \( A - X \), i.e. if the process can deadlock after execution of \( \sigma \), in case the environment allows only actions from \( X \). Two processes, not containing divergence (= infinite \( \tau \)-paths) are failure equivalent iff they have the same set of failure pairs. Now the model of process graphs (not containing divergence) modulo failure equivalence is isomorphic to the model of failure sets of section 3.4.

A node \( p \) of a process graph is said to be unstable if it has an outgoing \( \tau \)-edge. Remark that because \( \tau \notin A \), a path ending in an unstable node cannot contribute to the failure set of a process. This is on purpose, since deadlock can never occur if a \( \tau \)-step is possible. A consequence of this is that in the presence of divergence some information on the trace set of a process might get lost (in the construction of a failure set). A process containing a \( \tau \)-loop at every node, for instance, has no failure pairs! This is the reason for excluding diverging processes. They will be added however in
A variant of failure semantics is readiness semantics, as presented in Olderog & Hoare [14]. 
\(<\sigma, X> \in A^* \times \text{Pow}(A)\) is a ready pair of a process graph \(g\), if there is a path from the root of \(g\) to a node \(p\) with \(I(p) = X\). Ready equivalence must be a finer equivalence than failure equivalence since the failure set of a process is derivable from its ready set. The reverse however is not true: \(ab0 + ac0\) and \(ab0 + a(b0 + c0) + ac0\) are failure equivalent, but not ready equivalent.

4.6 Ready trace semantics

By now one might think that failure equivalence constitutes a preferable semantics for models of concurrency, since two processes are failure equivalent iff they are distinguishable by observation. However this depends to a great extent on the tools an environment has, to analyse processes. If these tools are unknown, then bisimulation semantics is in each case a safe choice. Therefore also in ACP and the topological process theory bisimulation semantics is used. In [2], Baeten, Bergstra & Klop show that a priority operator (as they introduced in the context of bisimulation semantics in [1]) is incompatible with failure semantics. Such an operator models an environment, which imposes a priority to the execution of certain atomic actions over others, and can be used for the specification of an interrupt mechanism. Moreover they present a semantics intermediate between readiness and bisimulation semantics (but without \(\tau\)-steps) that is compatible with priorities. In this ready trace semantics the role of a ready pair is replaced by an alternating sequence of subsets and elements of \(A\), representing a trace of a process, with for each node on the trace the set of possibilities to continue.

4.7 Survey

<table>
<thead>
<tr>
<th>Trace theory</th>
<th>CSP</th>
<th>Bisimulation: CCS, ACP, topological process theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace equivalence</td>
<td>failure equivalence</td>
<td>ready trace equivalence</td>
</tr>
<tr>
<td>(\delta)-trace equivalence</td>
<td>observation equivalence</td>
<td>observation congruence</td>
</tr>
</tbody>
</table>

more identifications less identifications

deadlock restrictions priorities fairness +

fig. 2
In figure 2, the equivalences mentioned above are classified. In the bottom line, the reasons are displayed to move into the direction of less identifications. Observation congruence will be discussed in section 5.1. If the schema suggests that all interesting equivalences can be linearly ordered by inclusion, then this is misleading; in order to keep the picture simple all equivalences disturbing the linearity are omitted. Furthermore bisimulation semantics identifies strictly less then failure semantics only in the absence of divergence. The differences between the various equivalences are further clarified by the axiomatisations in section 6.6. In figure 3, six process graphs are displayed, in such a way that in order to distinguish a process from the previous one, each time a finer equivalence is needed. This illustrates that in bisimulation semantics all information about the timing of choices is preserved, in trace semantics none, and in the other semantics some.

5. Features of concurrency

Both CCS and CSP capture nondeterminism, communication, recursion, abstraction, divergence and deadlock behaviour, but differently. A discussion per feature will follow below. In §6 the operators (the most important ones anyway) of CCS and CSP are presented, and provided with an operational semantics (as explained in section 3.6). For CSP this is not the usual method, but the failure semantics of both CCS and CSP are derivable by translating graphs to failure sets.
5.1 Nondeterminism

Both of the languages of CCS and CSP are equipped with a constant 0 for deadlock (called NIL in CCS and STOP in CSP) and with prefix multiplication \( aP \) (in CSP denoted as \( a\rightarrow P \)) for representing the sequential composition of \( a \) and \( P \). However they have different operators for choice. Hoare uses two operators for choice: \textit{external choice} \( \Box \) and \textit{internal choice} \( \cap \). The first kind of choice is deterministic: it depends on the environment; the second is nondeterministic: it cannot be influenced by the environment. A nondeterministic choice appears after abstraction from the actions of the environment that cause the choice for one of the alternatives. Both \( \Box \) and \( \cap \) are commutative, associative and idempotent (see the table of CSP axioms in section 6.4), i.e. the alternatives can be regarded to form a set. The difference appears in combination with deadlock: \( P\cap 0 = P \) but \( P\cap 0 \neq P \). Now the influence of the environment can be modeled by Milner's restriction operator: \( a\Box b\cap \backslash b = a\Box \) while \( a\Box b\cap \backslash b = a\Box 0 \) (for \( a \neq b \)). So the environment cannot force the process \( a\Box b\cap \backslash \) to choose its left summand.

Milner makes no distinction between external and internal choice; there is only one choice operator, \( + \). and apart from being commutative, associative and idempotent, it satisfies \( P + 0 = P \). On synchronisation trees, \( + \) composes two processes by identifying their roots. In CCS, nondeterminism is not a property of the operator, making an alternative composition of two processes \( P \) and \( Q \), but of the alternatives \( P \) and \( Q \) together. A choice can be regarded as fully nondeterministic if the environment does not participate in the selection of the alternatives. This can be modeled with the unary operator \( t \). A nondeterministic choice between \( P \) and \( Q \) can now be represented by \( tP + tQ \) (so \( P\cap Q = tP + tQ \)) and a deterministic choice between, say, \( a\Box \) and \( b\Box \) is represented by \( a\Box + b\Box \). On the other hand, the process \( t(a\Box + b\Box) \) can be represented in CSP by \( a\Box n(a\Box + b\Box) \).

If one tries to translate the CSP operator \( \Box \) into CCS (as is done for \( \cap \) above), one might think that it is just \( + \). However, this is not the case. If \( P \) and \( Q \) are starting with a \( t \)-step, then their \( + \)-composition yields a nondeterministic choice, while the operator \( \Box \) intends to remove this nondeterminism: \( t(a\Box + b\Box) = t(a\Box + b\Box) \). Therefore it is not possible to translate \( \Box \) into CCS directly. However it can be axiomatised over \( + \), and 0, as was shown by Brookes in [8], see section 6.5.

In CCS the processes \( a\Box \) and \( t\Box a \) are observation equivalent. However the processes \( a\Box + b\Box \) and \( t\Box a + b\Box \) are not; they are not even failure equivalent: \( (a\Box + b\Box)\backslash a = b\Box \) and \( (t\Box a + b\Box)\backslash a = t\Box a + b\Box \) have a different deadlock behavior. So in the presence of the \( + \)-operator, observation equivalence cannot be a criterion for identification; once \( a\Box \) and \( t\Box a \) are identified, \( a\Box + b\Box \) and \( t\Box a + b\Box \) cannot be distinguished. Summation is incompatible with observational equivalence in the same way as restriction is incompatible with \( \delta \)-trace semantics. For that reason Milner introduced (in [11]) the notion of observation congruence: two processes are observation congruent if they are observation equivalent in every context. This does give a suitable identification criterion (see also figure 2). Now any observation equivalence class contains exactly two observation congruence classes (\( P \) and \( tP \)). In the same way failure congruence can be defined (congruence with respect to \( + \)), but in CSP this is not necessary, since \( + \) is not a
CSP operator and failure equivalence is already a congruence for the CSP operators (as is
observation equivalence, see Brookes [8]).

5.2 Communication

In their treatment of communication, there are three differences between CSP and CCS:
- CSP has different operators for communication and interleaving (|| and |||), while CCS has one
operator (|) doing both.
- In CSP communication between two processes occurs if both of them offer the same action a ∈
A. In CCS this happens if one of them offers an atomic action a ∈ A, and the other its
complementary action ¯a.
- In CSP the communication between a and a results in the same step a. In CCS the
communication between a and ¯a results in a τ-step, i.e. the communication serves only as
synchronisation, the result is not visible.

These differences are illustrated by the following examples:

\[(a0 \parallel b0) || a0 = aa0 \parallel ba0 \parallel a(a0\mid b0)\]

\[(a0 \parallel b0) || a0 = a0\]

\[(a0 + b0) | a0 = a\overline{a}0 + b\overline{a}0 + \overline{a}(a0 + b0) + \tau 0.\]

In CCS there is a restriction operator \(\backslash a\), to remove the results of unsuccessful communication,
i.e. to remove some of the interleaving component of parallel composition:
\{(a0 + b0) | a0\} \backslash a,b = 0+0+0+\tau 0 = \tau 0.

In CSP such an operator is not present, but it is expressible using ||, if the alphabet A is finite.
Suppose that A = {a,b,c} then \(x|\backslash a = x || \mu X.(bX \bigoplus cX)\). In this translation of \(\backslash a\), the
actions A - {a}, allowed by the environment are used, instead of the disallowed action a.
\(\mu X.(bX \bigoplus cX)\) is the unique solution of the equation \(X = bX \bigoplus cX\), i.e. the infinite sequence
of choices between b and c.

The exact meaning of these operators is given by the operational semantics of CCS and CSP in the
sections 6.1 and 6.2. The algebraic laws governing them are listed in the sections 6.4 and 6.5. In
the listing of CCS axioms also the axioms of CSP operators in CCS context are presented, as in
Brookes [8].

5.3 Recursion

Both in CSP and CCS it is possible to specify a process by means of a fixed point equation. Such
an equation has the form \(X = P\) with \(P \in E\) a process expression and \(X\) a variable. The process
a a a a ..., performing an infinite sequence of a-steps, for instance, is specified by the fixed point
equation \(X = aX\). Some fixed point equations, like \(X = aX\), have unique solutions (in the
mentioned failure and graph models) but others have more solutions (any process satisfies \( X = X \)); however there are no fixed point equations without solutions. Both CSP and CCS use the expression \( \mu X.P \) to denote the unique solution of \( X = P \), if there is one. If \( X = P \) has no unique solution, then \( \mu X.P \) should denote some default element from the solution set. The question which one is answered differently in CSP and CCS.

In the failure model of CSP the inclusion ordering on failure sets makes this model into a complete lattice. On process expressions this ordering is characterised by the condition \( X \subseteq Y \) iff \( X \cap Y = Y \) (\( Y \) is less deterministic than \( X \)). Now all CSP operators turn out to be monotonic for this ordering (i.e. \( X \subseteq X' \) implies \( f(X,Y) \subseteq f(X',Y) \)), and using general fixed point theory this implies that any fixed point equation has a minimal and a maximal solution. Hoare chooses the maximal fixed point to be the interpretation of \( \mu X.P \). His reason for doing so is that underspecification expresses uncertainty about the specified process. Therefore the default solution of the equation should be the least deterministic one (the least predictable). In the most extreme case (of the fixed point equation \( X = X \)) there is complete underspecification and no certainty at all. Therefore \( \mu X.X \) is chosen to be the least deterministic of all processes: the process CHAOS. The failure set of CHAOS is \( A \times \mathcal{P}wr(A) \). CHAOS can be regarded as the internal sum \( (\pi) \) of all processes. In the calculus of CSP, CHAOS can be added as a new constant \( \chi \), satisfying the law \( \chi \cap X = \chi \).

In CCS this method cannot be applied, since prefixing is not monotonic for the CSP-ordering (due to the absence of an axiom \( aX \cap aY = a(X \cap Y) \)), and no complete partial order can be found for which the CCS operators are continuous. However a fixed point is found in the graph, generated by the action rules for the operational semantics of section 6.1. That this graph really satisfies its fixed point equation (the recursion axiom in section 6.5) follows trivially from the action rule for recursion in section 6.1. Remark that Milner's fixed point is another one than Hoare's: in CCS \( \mu X.X = 0 \), while in CSP \( \mu X.X = \chi \). Milner's fixed point is failure equivalent to the least fixed point existing in CSP, while in CSP the largest fixed point is chosen.

Also sets of fixed point equations can be used to specify processes. A recursive specification \( E \) is a set \( \{ X = P_X \mid X \in \Xi \} \) with \( \Xi \) a set of variables and \( P_X \) a process expression (for \( X \in \Xi \)). Example: if \( E = \{ X = aY, Y = bX \} \), then \( X = ababab... \) and \( Y = bababa... \). The \( X \)-component of the solution vector of \( E \) is denoted by \( \langle X \mid E \rangle \). Thus, \( \langle X \mid E \rangle \) means: 'the \( X \), as specified by \( E \)'. This is a safer expression than just \( X \), since the variable \( X \) can also occur in other specifications. However, in most contexts the names of the variables in all mentioned specifications are chosen to be distinct, so that \( \langle X \mid E \rangle \) can safely be abbreviated by \( X \).

If \( E \) is finite then the expression \( \langle X \mid E \rangle \) can be translated into a CCS or CSP expression, involving the nested use of the recursion operator \( \mu \). Example: \( \langle X \mid X = aX + bY, Y = cX + dY \rangle = \mu X.(aX + b\mu Y.(cX + dY)) \).

5.4 Abstraction

In CSP there is a concealment operator \( / \alpha \) (for \( \alpha \in A \)) for hiding those actions we are not interested in. As in Brookes [8] and De Nicola [13] the notation \( / \alpha \) is used instead of \( \\not\alpha \), in order
to distinguish abstraction from restriction. Its operational behaviour and the axioms governing it can be found in the sections 6.2 and 6.4. The application of such an operator is called 'abstraction from internal steps'. There is a big difference between abstraction and restriction: \( abcO/b = acO \), while \( abcO\setminus b = a0 \).

In CCS there is no separate concealment operator, since there abstraction and communication are integrated. However, hiding can be expressed by the operators for parallel composition and restriction: \( x/a = (x|\mu Y.ay+\bar{a}y)\setminus a \), where \( \mu Y.ay+\bar{a}y \) is the process, only generating \( a \)- and \( \bar{a} \)-steps. The translation of concealment into CCS can be axiomatised by the axioms in section 6.5. Using the CSP axioms for concealment, one finds (if \( a \neq b \)): \( (acO\setminus bdO)/a/b = cOnd0n(cOnd0) \), and using the CCS axioms: \( (acO\setminus bdO)/a/b = (acO + bdO)/a/b = tcO + tdO = cOnd0 \). This is indeed the same result, since in failure semantics \( xny = xny(n(xny)) \), as can be verified by either using the distributive laws of section 6.4, or the failure axioms of section 6.6.

5.5 Divergence

On process graphs, abstraction from an atomic action \( a \) consists of replacing all \( a \)-edges by \( \tau \)-edges. This might result in divergence (infinite \( \tau \)-paths) as is the case in \( (\mu X.aX)/a \). Here an infinite \( a \)-path changes into an infinite \( \tau \)-path. Contrary to the equation \( X = aX \), that has the infinite \( a \)-path as unique solution, the CCS equation \( X = \tau X \) is satisfied by many processes, of which \( \tau 0 \) is the simplest. However, the process that is selected to be the default solution of \( X = \tau X \) (by Milner's operational semantics of CCS) is just the infinite \( \tau \)-path. Hence \( (\mu X.aX)/a = \mu X.\tau X \), and in general \( (\mu X.P)/a = \mu X.(P/a) \), so abstraction and recursion commute.

In CSP the situation is different in two respects: first the expression \( \mu X.P \) has another meaning than in CCS and second, by the absence of \( \tau \), it is not possible to define a divergent process directly (in section 4.5, divergent processes were even excluded from the failure model). In general \( x/a \) is the process \( X \) from which the \( a \)-steps are removed (\( aabO/a = bO \) in CSP). Thus, the actions 'behind' the \( a \)-steps are moved forwards. But since it is not clear, what can be thought to be behind an infinite sequence of \( a \)-steps, Hoare has some freedom in giving a meaning to \( (\mu X.aX)/a \) in the failure model. He chooses to treat 'overabstraction' like underspecification, and the result is that also in CSP recursion and abstraction do commute, so \( (\mu X.aX)/a = \mu X.((aX)/a) = \mu X.X = X \).

In combination with the interpretation of abstraction on process graphs, this implies that any divergent process (= a process containing divergence at the root) is failure equivalent with the process \( X \). This removes the restriction on failure equivalence, that it is only defined on processes not containing divergence. However, by doing so, a lot of interesting information about divergent processes gets lost: even the processes

\[
\begin{align*}
\tau & \quad \downarrow \quad a \quad \rightarrow \quad O \\
\tau & \quad \downarrow \quad b \quad \rightarrow \quad O
\end{align*}
\]

are identified! This is the reason a different form of failure semantics is presented in Bergstra, Klop.
5.6 Deadlock behaviour

Deadlock is the state of a process where no further action is possible. It can occur in a merge of two processes if both of them are waiting for the other to provide a suitable communication. Example (in CCS):

\[\{[\mu X. a(acX+bdX)](\mu Y. \overline{a}Y)\}a\|b = \tau d\tau c\omega.\]

As explained in §4, deadlock behaviour is preserved by \(\partial\)-trace, failure, ready and ready trace equivalence, but not by trace equivalence. Furthermore \(\partial\)-trace equivalence is disqualified since it is disturbed in the presence of communication and restriction operators, modeling the influence of the environment. In the absence of divergence deadlock behaviour is preserved in bisimulation semantics too, but in the presence of divergence it is preserved only in combination with livelock behaviour.

Livelock is the state of a process where only an infinite sequence of hidden moves is possible, as in \(\mu X. \tau X\). In CSP livelock (being a special case of divergence) is equated with the fully unpredictable process \(\text{CHAOS}\). In CCS it is equated with deadlock: \(\mu X. \tau X = \tau 0\).

Deadlock can be visualised if processes are supposed to make noise. The noise starts at the beginning of a process and ends if the process reaches a state of deadlock. If a component in a merge has to wait for a suitable communication it becomes silent until the communication is enabled, but as long as at least one component is making progress, noise is being made. Only if all components are waiting, the process becomes silent. This guarantees that no further action is possible.

In this interpretation deadlock can be distinguished from livelock. Of course it is also possible to define a version of bisimulation where deadlock and livelock behaviour are distinguished.

6. Survey of CCS and CSP

6.1 An operational semantics for CCS

Let \(\Delta\) be a given set of names. \(\Delta^- = \{\overline{a} \mid a \in \Delta\}\) is the corresponding set of conames. \(\Delta \cap \Delta^- = \emptyset\). Let \(\emptyset \neq A = \Delta \cup \Delta^-\) be the invisible step and write \(A_\tau = \Delta \cup \Delta^- \cup \{\tau\}\). Let \(a, b, c\) range over \(A_\tau\) and put \(\overline{a} = a, (\overline{\tau} = \tau)\). A function \(R: \Delta \to \Delta\) is called a relabeling.

The domain of a relabeling \(R\) can be expanded to \(A_\tau\) by putting \(R(\tau) = \tau\) and \(R(\overline{a}) = \overline{R(a)}\).

Let \(V\) be a given set of variables, then the set \(E\) of CCS expressions is defined inductively by:
Fischer's Protocol in Timed Process Algebra

Jan Joris Vereijken

Department of Computing Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

http://www.win.tue.nl/win/cs/fm/janjoris/
janjoris@acm.org

April 21, 1995

Abstract

Timed algebraic process theories can be developed with quite different purposes in mind. One can aim for theoretical results about the theory itself (completeness, expressiveness, decidability), or one can aim for practical applicability to non-trivial protocols. Unfortunately, these aims do not go well together. In this paper we take two theories, which are probably of the first kind, and try to find out how well suited they are for practical verifications.

We verify Fischer's protocol for mutual exclusion in the settings of discrete-time process algebra (ACPdt) and real-time process algebra (ACPur). We do this by transforming the recursive specification into an equivalent linear specification, and then dividing out the maximal bisimulation relation. The required mutual exclusion result can then be found by reasoning about the obtained process graph.

Finally, we consider the ease of the verification, and ways to adapt the theory to make it more practical. It will turn out that the theories investigated are quite unsatisfactory when verifying real-life protocols.

1991 Mathematics Subject Classification: 68Q10, 68Q22, 68Q60.
Keywords: Fischer's protocol, process algebra, real time, discrete time, ACP, mutual exclusion, verification.

Note: These investigations were supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO).

Contents

1 Introduction 2
2 Fischer's protocol 3
  2.1 History of the protocol ........................................ 3
  2.2 Proof requirements .......................................... 3
  2.3 First informal description .................................. 4
  2.4 Second informal description ................................. 5
Now the action relations $\sigma \rightarrow \subseteq E \times E$ for $\sigma \in A^*$ are generated by the following rules:

- $aP \xrightarrow{a} P$
- $P \parallel Q \xrightarrow{a} P$
- $P \parallel Q \xrightarrow{Q} Q$
- From $P \xrightarrow{a} Q$ (a=t) infer: $P \parallel S \xrightarrow{a} Q$
  \[ S \parallel P \xrightarrow{a} Q \]
  \[ f(P) \xrightarrow{f(a)} f(Q) \]
- From $P \xrightarrow{a} Q$ infer: $P \parallel S \xrightarrow{a} Q \parallel S$
  \[ S \parallel P \xrightarrow{a} S \parallel Q \]
  \[ f(P) \xrightarrow{f(Q)} \]
- From $P \xrightarrow{a} Q$ and $S \xrightarrow{a} T$ (a=t) infer: $P \parallel S \xrightarrow{a} Q \parallel T$
- From $P \xrightarrow{a} Q$ infer: $P \parallel S \xrightarrow{a} Q \parallel S$
  \[ S \parallel P \xrightarrow{a} S \parallel Q \]
  \[ f(P) \xrightarrow{f(Q)} \]
  \[ P/a \xrightarrow{Q/a} \]
  and if a=b: $P/b \xrightarrow{Q/b}$
- From $P[X:=\mu X.P] \xrightarrow{a} Q$ infer $\mu X.P \xrightarrow{a} Q$

### 6.3 Equivalences on process expressions

Let $\sigma \rightarrow \subseteq E \times E$ for $\sigma \in A^*$, the set of finite words over $A$, be the least relation satisfying:

- $\sigma \xrightarrow{a} \sigma$
- If $P \xrightarrow{a} Q$ then $P \xrightarrow{a} Q$ (for $a \in A$)
- If $P \xrightarrow{a} Q$ then $P \parallel Q \xrightarrow{a} Q$
- If $P \parallel Q$ and $Q \xrightarrow{a} S$ then $P \parallel Q \xrightarrow{a} S$

A $\tau$-bisimulation is a relation $R \subseteq E \times E$, satisfying (for all $\sigma \in A^*$)

- If $P \parallel Q$ and $P \xrightarrow{\sigma} P'$ then $Q \xrightarrow{\sigma} Q'$ and $P' \parallel Q'$ for some $Q' \in E$.
- If $P \parallel Q$ and $Q \xrightarrow{\sigma} Q'$ then $P \xrightarrow{\sigma} P'$ and $P' \parallel Q'$ for some $P' \in E$.

$\sigma \in A^*$ is a trace of $P$ if $P \xrightarrow{\sigma} Q$ for some $Q \in E$.

$P$ is divergent if there is an infinite $\tau$-path $P \xrightarrow{a} P_1 \xrightarrow{a} P_2 \xrightarrow{a} \ldots$.

$\langle \sigma, X \rangle \in A^* \times P^{\omega}(A)$ is a failure pair of $P$ if $P \xrightarrow{\sigma} Q$ for some $Q \in E$ such that $Q \xrightarrow{a} S$ implies $a \in A - X$, or if $\sigma = \sigma' \cdot \sigma''$ and $P \xrightarrow{\sigma'} Q$ for some divergent $Q \in E$.

$P$ and $Q$ are observation equivalent ($P \equiv Q$) if $PRQ$ for some $\tau$-bisimulation $R$.

$P$ and $Q$ are observation congruent if $P + S \equiv Q + S$ for any $S \in E$.

$P$ and $Q$ are failure equivalent ($P \equiv Q$) if they have the same failure sets.

$P$ and $Q$ are failure congruent if $P + S \equiv Q + S$ for any $S \in E$.

$P$ and $Q$ are trace equivalent if they have the same trace set.
6.4 Axioms for CSP (and failure equivalence)

**External choice:**
\[ x \circ y = y \circ x \]
\[ x \circ (y \circ z) = (x \circ y) \circ z \]
\[ x \circ x = x \]
\[ x \circ 0 = x \]

**Internal choice:**
\[ x \circ y = y \circ x \]
\[ x \circ (y \circ z) = (x \circ y) \circ z \]
\[ x \circ x = x \]

**Distributive laws:**
\[ x \circ (y \circ z) = (x \circ y) \circ (x \circ z) \]
\[ x \circ (y \circ z) = (x \circ y) \circ (x \circ z) \]
\[ a \circ x \circ y = a \circ (x \circ y) \]
\[ a \circ x \circ y = a \circ (x \circ y) \]

**Communication:**
\[ x \parallel y = y \parallel x \]
\[ (x \parallel y) \parallel z = (x \parallel z) \parallel (y \parallel z) \]
\[ P \parallel Q = \bigcirc_{i} a_{i} (P \parallel Q) \]
\[ a_{i} = b_{j} \]

**Interleaving:**
\[ x \parallel y \parallel x = y \parallel x \]
\[ (x \parallel y) \parallel z = x \parallel z \parallel y \parallel z \]
\[ P \parallel Q = \bigcirc_{i} a_{i} (P \parallel Q) \bigcirc_{j} b_{j} (P \parallel Q) \]
\[ i \quad j \]

**Concealment:**
\[ (x \circ y) / a = x / a \parallel y / a \]
\[ (a \circ x \circ y) / a = x / a \parallel (x \circ y) / a \]
\[ \bigcirc_{i} b_{i} P_{i} / a = \bigcirc_{i} b_{i} P_{i} / a \quad \text{if } \forall i \ b_{i} \neq a \]

**Relabelling:**
\[ f(0) = 0 \]
\[ f(x) = f(x) \]
\[ f(x) = f(x) \]
\[ f(x) = f(x) \]

**Recursion:**
\[ \mu X. P = P[X = \mu X. P] \]

* Here \( P = a_{1} P_{1} \parallel a_{2} P_{2} \parallel \ldots \parallel a_{n} P_{n} \)
\[ \bigcirc_{i=1}^{n} a_{i} P_{i} \] and \( Q = \bigcirc_{j=1}^{m} b_{j} Q_{j} \). Put \( \bigcirc_{i=1}^{n} a_{i} P_{i} = 0 \).
4. STATE OPERATOR.
In this section, we extend any of the theories BPAps, PAlps, ACPps with the state operator of [BAB88]. The interesting aspect here is, that we allow the state to be (partly) visible to the process, i.e. a state can emit a signal.

4.1. SYNTAX AND SEMANTICS.
Let us assume that a state operator in the sense of [BAB88] is given by a domain \( S \) and functions \( \text{act}: A \times S \rightarrow \mathcal{A} \cup \{\delta\} \) and \( \text{eff}: A \times S \rightarrow S \). The expression \( \lambda_s(x) \) with \( s \in S \) denotes process \( x \) working on the state space \( S \) with the current state being \( s \in S \).

We can assume that there is an additional function \( \text{sig}: S \rightarrow B \) which determines for each state the signal that is emitted by that state. The absence of signals is modeled by taking \( \text{sig}(s) = T \) of course. Now the eight equations for the state operator are as shown in table 17, the operational semantics is given in table 18.

Using a state operator that generates signals one can define signaling processes in such a way that the equations need not contain any signal at all, thus considerably optimizing the notation. We will illustrate this in a simple example.

\[
\begin{align*}
\lambda_s(\perp) &= \perp & \text{(SOS0)} & \lambda_s(x + y) &= \lambda_s(x) + \lambda_s(y) & \text{(SOS4)} \\
\lambda_s(\delta) &= \text{sig}(s) \delta & \text{(SOS1)} & \lambda_s(\phi \leftarrow x) &= \phi \leftarrow \lambda_s(x) & \text{(SOS5)} \\
\lambda_s(a) &= \text{sig}(s) \leftarrow \text{act}(a, s) \leftarrow \text{sig}(\text{eff}(a, s)) & \text{(SOS2)} & \lambda_s(x \leftarrow \phi) &= \lambda_s(x) \leftarrow \phi & \text{(SOS6)} \\
\lambda_s(a \cdot x) &= \text{sig}(s) \leftarrow \text{act}(a, s) \leftarrow \lambda_s(\text{eff}(a, s) \cdot x) & \text{(SOS3)} & \lambda_s(\phi : \rightarrow x) &= \text{sig}(s) \leftarrow \phi : \rightarrow \lambda_s(x) & \text{(SOS7)}
\end{align*}
\]

<table>
<thead>
<tr>
<th>x^\phi \rightarrow s_p, \text{sig}(s) \wedge s_p</th>
<th>(x) \neq F, \text{sig}(\text{eff}(a, s)) \wedge s_p</th>
<th>(x') \neq F, \text{act}(a, s) = b \neq \delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lambda_s(x) \quad \phi \cup \lambda \text{eff}(a, s) \quad (x')</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| x^\phi \rightarrow s_p, \psi, \text{sig}(s) \wedge s_p | (x) \neq F, \text{sig}(\text{eff}(a, s)) \wedge \psi \neq F, \text{act}(a, s) = b \neq \delta |
|---|---|---|
| \lambda_s(x) \quad \phi \cup \psi \wedge \text{sig}(\text{eff}(a, s)) \wedge \psi |

TABLE 17. State operator generating signals.

\[
\begin{align*}
\lambda_s(\perp) &= \perp & \text{SOS0} & \lambda_s(x + y) &= \lambda_s(x) + \lambda_s(y) & \text{SOS4} \\
\lambda_s(\delta) &= \text{sig}(s) \delta & \text{SOS1} & \lambda_s(\phi \leftarrow x) &= \phi \leftarrow \lambda_s(x) & \text{SOS5} \\
\lambda_s(a) &= \text{sig}(s) \leftarrow \text{act}(a, s) \leftarrow \text{sig}(\text{eff}(a, s)) & \text{SOS2} & \lambda_s(x \leftarrow \phi) &= \lambda_s(x) \leftarrow \phi & \text{SOS6} \\
\lambda_s(a \cdot x) &= \text{sig}(s) \leftarrow \text{act}(a, s) \leftarrow \lambda_s(\text{eff}(a, s) \cdot x) & \text{SOS3} & \lambda_s(\phi : \rightarrow x) &= \text{sig}(s) \leftarrow \phi : \rightarrow \lambda_s(x) & \text{SOS7}
\end{align*}
\]

TABLE 18. Operational semantics.

4.2 EXAMPLE.
Let \( D \) be a finite alphabet of data, and let \( D^* \) be the collection of finite sequences over \( D \). The empty sequence is denoted by \( \varepsilon \) and adding an element \( d \) to the list \( x \) results in \( ad \). The propositional constants are as follows: \( \text{on}_\text{top}(d) \) for \( d \in D \), and \( \text{empty} \).

We will assume that these signals are exclusive, i.e. we will assume that the following formula always holds: \( \Phi = (\text{empty} \supset \wedge \text{d} \in D \text{ on}_\text{top}(d)) \wedge \wedge \text{d} \in D \text{ on}_\text{top}(d) \supset \wedge \text{d} \in D \text{ on}_\text{top}(d) \).

\( D^* \) will be the state space for a process that represents a stack over \( D \). The signal function \( \text{sig} \) is defined by \( \text{sig}(\varepsilon) = \text{empty} \), \( \text{sig}(\varepsilon d) = \text{top}(d) \). The atomic actions are:
6.6 Axioms for identification of CCS expressions

**Observational congruence:**
\[ a \cdot x = a \cdot x \]
\[ \tau x + x = \tau x \]
\[ a(\tau x + y) = a(\tau x + y) + a x \]

**Ready congruence:**
\[ a(\tau x + \tau y) = a x + a y \]
\[ \tau x + x = \tau x \]
\[ \tau (\tau x + y) = \tau x + y \]
\[ \tau (a x + a y + z) = \tau (a x + z) + a y \]

**Failure congruence:**
\[ a(\tau x + \tau y) = a x + a y \]
\[ \tau x + y = \tau (x + y) + \tau x \]
\[ \tau (a x + a y + z) = \tau (a x + z) + a y \]

**Trace equivalence:**
\[ a x + a y = a(x + y) \]
\[ \tau x = x \]

7. References


The Algebra of Communicating Processes with Empty Process

Abstract. The axiom systems $\text{BPA}^e$, $\text{BPA}_6^e$, $\text{PA}^e$ and $\text{ACP}^e$ are presented, which are in essence the systems $\text{BPA}$, $\text{PA}$ and $\text{ACP}$ extended with the empty process. For the system $\text{ACP}^e$, that contains the other three systems, the first steps are taken towards making it operational: the most basic supplementary axioms are given, also adapted such that they can deal with the empty process and a graph model is constructed. The validity of the supplementary axioms is proven in both the closed term model and the graph model.

Contents

1. Introduction.

2. The axiom systems $\text{BPA}^e$, $\text{BPA}_6^e$, $\text{PA}^e$ and $\text{ACP}^e$.

3. Supplementary axioms.

4. A graph model for $\text{ACP}^e$.

5. The supplementary axioms in the graph model.

1. Introduction.

1.1. Process Algebra is an axiomatic theory of processes, comparable to CCS ([M]) or CSP ([BHR]). It studies processes via axiom systems in which notions like sequential execution, parallel execution, alternative choice, communication etc are formalized by means of operators and equations. Its main goal consists of the specification and verification of concurrent processes (e.g., communication protocols). Key topics in the theory are the semantics of the axiom systems (initial algebras, graph models, projective limit models and many others), recursive specifications (sets of equations that determine a process in a model), fixed-point operators, methods to reduce the extensive mass of calculation that is often needed, process execution under priority conditions, the expressive power of axiom systems, the relationship with other concurrency theories, interleaving semantics versus true concurrency, etc.

For quite a while, Process Algebra has been developed without a neutral element for sequential execution, a so-called empty process. There happened to be no imperative motivation for its introduction. Nevertheless in [KV] it was shown that a lot of things are easier when there is a constant for the empty process. E.g., the translation of recursive specifications into graphs, or the specification of a stack with more than one data element. That is why the extension of Process
Algebra with an empty process was undertaken. In the beginning this was rather problematic. Especially the axiomatization of the left merge operator (an auxiliary operator needed for the axiomatization of parallel execution) and the construction of a graph model were not trivial. For these two problems a solution is offered in the present paper.

The empty process will also be the neutral element for parallel execution (merge, $\parallel$) and the right-hand side neutral element for the left merge operator ($\ll$). For intuitive reasons there is no neutral element for the communication merge operator ($\mid$) or a left-hand side neutral element for the left merge. The deadlock constant $\delta$ furnishes the neutral element for alternative choice, also called sum.

Following the not altogether successful, yet highly instructive introduction of the empty process $e$ in the axiom systems BPA and PA in [KV], the next stage concerns the axiom system ACP ([BKl]). Together with a number of other changes, the introduction of the empty process in the system ACP leads to the new axiom system ACP$^e$. This transformation process is discussed in some depth in section 2. Also improved versions of BPA$^e$ and PA$^e$ ([KV]) are given in this paper, which are called BPA$^e$, BPA$^e_\delta$ and PA$^e$.

Things most needed to make ACP$^e$ ready for applications are a number of supplementary axioms like handshaking, the projection axioms, the $e$-abstraction rule etc, and a graph model for ACP$^e$. These are supplied in section 3 where the supplementary axioms are given and their validity in the closed term model is proven, and in section 4 where a graph model is constructed. In section 5 the validity of the supplementary axioms in this graph model is proven.

Certainly, this is not enough to make ACP$^e$ a full-grown Process Algebra axiom system. Topics that remain for future treatment are (among others) a rewrite analysis of ACP$^e$, axioms like AIP, RDP and RSP that deal with infinite processes, and an abstraction mechanism.

1.2. This is not an introductory text about Process Algebra. Introductory texts are [BW], [BKl] and [BK2].

This paper replaces rather than extends its predecessor [KV]. Reading the latter may be useful but is certainly not necessary.

Acknowledgements.

Here I would like to express my gratitude to all the members of the PAM-group and notably to Karst Koymans (the usefulness of an empty process was his idea) and Jos Baeten, without whose help I could not have finished this text.

2. The axiom systems BPA$^e$, BPA$^e_\delta$, PA$^e$ and ACP$^e$.

2.1. The axiom systems presented in this section, BPA$^e$, BPA$^e_\delta$, PA$^e$ and ACP$^e$, all consist of a signature (summing up sorts, functions and constants) and a set of equations. The signature is denoted by $\Sigma$, and the set of equations by $E$. For example $\text{ACP}^e = (\Sigma_{\text{ACP}^e}, E_{\text{ACP}^e})$. Like other Process Algebra axiom systems, they have therefore the form of an algebraic specification. The systems BPA$^e$, BPA$^e_\delta$, and PA$^e$ are given just for the purpose of reference, this paper will mostly concentrate on ACP$^e$ as this system contains the others.

2.2 Let us spend a few words on the building principles behind Process Algebra axiom systems.

• As mentioned before, the systems have the form of algebraic specifications or rather are denotations which after some expansions and substitutions yield a possibly infinite algebraic specification. In this sense anything is allowed as long as the system has an initial algebra. The initial algebra or closed term model will be denoted CTM in the sequel.
• Only sum and product are primitive operators. The other operators (∥, ⋯, ) are "defined" in terms of these primitive operators. This means that when applied to closed terms they can be eliminated. The merge operator can be removed immediately, with the only axiom that contains the merge operator on the left hand side. The axioms for the operators ∥, ⋯ follow the induction scheme by which closed terms are built:

\[ \varepsilon, \delta, a \cdot x, x+y \]

(This scheme does not produce all closed terms but it does produce a representative for all elements of the closed term model, even for ACP', although this will not be proven in this paper.)

For renaming operators one prefers to use a somewhat stronger induction scheme (which is not possible for ∥ or ⋯):

\[ \varepsilon, \delta, x \cdot y, x + y \]

It is not hard to show that for renaming operators restricted to closed terms the two schemes lead to equivalent definitions.

• Axioms that would be provable in CTM (the closed term model) are avoided in order to make the system as basic as possible. Sometimes an axiom is included although it is derivable in CTM, for example the axiom \( x \parallel y = y \parallel x \). This axiom brings a great simplification. Four former axioms in ACP' are rendered unemployed, which is given higher priority.

• The incremental construction and an effort towards standardisation serve an obvious didactic and practical purpose.

2.3.1. The axiom systems treated here are in incremental order: BPA', BPA'I, PA' and ACP'. One might expect PA' and PA'I to appear separately but the new axioms for \( \varepsilon \parallel \ldots \) make it impossible to define the left merge without the constant \( \delta \).

As usual the binding order of the operators is such that sum is weakest, product is strongest and the other operators are on an equal level in between. The product dot is usually omitted.

All these systems depend on parameters, among others the set A of atomic steps. This means that for different values of A you get a different axiom system. The expansion variables I and r in ACP' are used to avoid, by means of textual expansion, repetition in the definition of the renaming operators.

2.3.2 \( \text{BPA}' = \left( \Sigma \text{BPA}', E \text{BPA}' \right) \).

\[ \Sigma \text{BPA}' \]

<table>
<thead>
<tr>
<th>Sort</th>
<th>P</th>
<th>(processes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>( + : P \times P \rightarrow P )</td>
<td>(choice)</td>
</tr>
<tr>
<td></td>
<td>( \cdot : P \times P \rightarrow P )</td>
<td>(sequential execution)</td>
</tr>
<tr>
<td>Constants</td>
<td>( \varepsilon )</td>
<td>(empty process)</td>
</tr>
<tr>
<td></td>
<td>elements of A</td>
<td>(atoms)</td>
</tr>
</tbody>
</table>

| Parameter of BPA' : A, the set of atomic steps or atoms. The elements of A are constants in the system. |
The Algebra of Communicating Processes with Empty Process

\[ E_{\text{BPA}}^{\varepsilon} \]

| x + y = y + x  |
| (x + y) + z = x + (y + z) |
| (x + y) z = x z + y z  |
| (x y) z = x (y z) |
| \varepsilon + \varepsilon = \varepsilon |
| \varepsilon x = x |
| x \varepsilon = x |

table 2

2.3.3 \( BPA^{\varepsilon \delta} = (\Sigma_{\text{BPA}}^{\varepsilon \delta}, E_{\text{BPA}}^{\varepsilon \delta}) \).

\[ \Sigma_{\text{BPA}}^{\varepsilon \delta} \]

| Sort          | P                      |
| Functions     | \( + : P \times P \rightarrow P \) (choice) |
|               | \( \cdot : P \times P \rightarrow P \) (sequential execution) |
| Constants     | \varepsilon             (empty process) |
|               | \delta                   (deadlock) |
|               | elements of A           (atoms) |

table 3

Parameter of \( BPA^{\varepsilon \delta} \): A, the set of atomic steps or atoms. The elements of A are constants in the system.
The Algebra of Communicating Processes with Empty Process

\[ E_{BP} \]

\[
\begin{align*}
x + y &= y + x \\
(x + y) + z &= x + (y + z) \\
(x + y)z &= xz + yz \\
(x y)z &= x(y z)
\end{align*}
\]

\[
\begin{align*}
\epsilon + \epsilon &= \epsilon \\
\epsilon x &= x \\
x \epsilon &= x
\end{align*}
\]

\[
\begin{align*}
\delta + \epsilon &= \epsilon \\
\delta x &= \delta
\end{align*}
\]

table 4

2.3.4 \( PA^\epsilon = (\Sigma PA^\epsilon, E PA^\epsilon) \)

\[
\Sigma PA^\epsilon
\]

<table>
<thead>
<tr>
<th>Sort</th>
<th>P</th>
<th>(processes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>+ : ( P \times P \rightarrow P )</td>
<td>(choice)</td>
</tr>
<tr>
<td></td>
<td>\cdot : ( P \times P \rightarrow P )</td>
<td>(sequential execution)</td>
</tr>
<tr>
<td></td>
<td>| : ( P \times P \rightarrow P )</td>
<td>(merge)</td>
</tr>
<tr>
<td></td>
<td>| : ( P \times P \rightarrow P )</td>
<td>(left merge)</td>
</tr>
<tr>
<td>Constants</td>
<td>\epsilon</td>
<td>(empty process)</td>
</tr>
<tr>
<td></td>
<td>\delta</td>
<td>(deadlock)</td>
</tr>
<tr>
<td></td>
<td>elements of A</td>
<td>(atoms)</td>
</tr>
</tbody>
</table>

table 5

Parameter of \( PA^\epsilon \) : \( A \), the set of atomic steps or atoms. The elements of \( A \) are constants in the system.
The Algebra of Communicating Processes with Empty Process

\( E_P_{\text{ACP}}\)

\[
\begin{align*}
x + y &= y + x \\
(x + y) + z &= x + (y + z) \\
(x + y)z &= x(z + y) \\
\varepsilon + \varepsilon &= \varepsilon \\
\varepsilon x &= x \\
\delta + \varepsilon &= \varepsilon \\
\delta x &= \delta
\end{align*}
\[
\begin{align*}
x \parallel y &= x \parallel y \parallel x \\
\varepsilon \parallel \varepsilon &= \varepsilon \\
\varepsilon \parallel \delta &= \delta \\
\varepsilon \parallel a x &= \delta \\
\varepsilon \parallel (x + y) &= \varepsilon \parallel x + \varepsilon \parallel y \\
a x \parallel y &= a(x \parallel y) \\
(x + y) \parallel z &= x \parallel z + y \parallel z
\end{align*}
\]

Table 6

2.3.5 \( \text{ACP}^\varepsilon = ( \Sigma_{\text{ACP}^\varepsilon}, E_{\text{ACP}^\varepsilon} ) \)

\( \Sigma_{\text{ACP}^\varepsilon} \)

<table>
<thead>
<tr>
<th>Sort</th>
<th>P</th>
<th>(processes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>( + : P \times P \rightarrow P )</td>
<td>(choice)</td>
</tr>
<tr>
<td></td>
<td>( \cdot : P \times P \rightarrow P )</td>
<td>(sequential execution)</td>
</tr>
<tr>
<td></td>
<td>( \parallel : P \times P \rightarrow P )</td>
<td>(merge)</td>
</tr>
<tr>
<td></td>
<td>( \parallel : P \times P \rightarrow P )</td>
<td>(left merge)</td>
</tr>
<tr>
<td></td>
<td>( \parallel : P \times P \rightarrow P )</td>
<td>(communication merge)</td>
</tr>
<tr>
<td>Constants</td>
<td>( \varepsilon )</td>
<td>(empty process)</td>
</tr>
<tr>
<td></td>
<td>( \delta )</td>
<td>(deadlock)</td>
</tr>
<tr>
<td></td>
<td>elements of ( A )</td>
<td>(atoms)</td>
</tr>
</tbody>
</table>

Table 7

Parameters of \( \text{ACP}^\varepsilon \):
- \( A \), the set of atomic steps or atoms. The elements of \( A \) are constants in the system.
- \( \gamma : A \times A \rightarrow A \cup \{ \delta \} \) the communication function on atoms.
  \( \gamma \) must obey \( \gamma(a,b) = \gamma(b,a) \) and if \( \gamma(a,b), \gamma(b,c) \in A \) then \( \gamma(\gamma(a,b),c) = \gamma(a,\gamma(b,c)) \).

Expansion variables:
- \( I \) ranges over \( \wp(A) \) (\( \wp \) denotes powerset). I may be empty; this means that the system does not include an \( \varepsilon \)-renaming operator.
- \( r \) ranges over \( \{ \varepsilon, \delta \} \). So in the axioms below, also \( r_I \) should be expanded into \( \varepsilon_I \) and \( \delta_I \).
The Algebra of Communicating Processes with Empty Process

\[ E_{ACP} \]

\[
\begin{align*}
\varepsilon + \varepsilon &= \varepsilon \\
\varepsilon x &= x \\
x \varepsilon &= x \\
\delta + \varepsilon &= \varepsilon \\
\delta x &= \delta \\
x \parallel y &= x \parallel y + \parallel x + x \parallel y \\
\varepsilon \parallel \varepsilon &= \varepsilon \\
\varepsilon \parallel \delta &= \delta \\
\varepsilon \parallel a x &= \delta \\
\varepsilon \parallel (x + y) &= \varepsilon \parallel x + \varepsilon \parallel y \\
\delta \parallel x &= \delta \\
a x \parallel y &= a (x \parallel y) \\
(x + y) \parallel z &= x \parallel z + y \parallel z \\
a \parallel b &= \gamma(a, b) \\
x \parallel y &= y \parallel x \\
\delta \parallel x &= \delta \\
a x \parallel y &= (a \parallel y) \parallel x \\
(x + y) \parallel z &= x \parallel z + y \parallel z \\
\eta(\varepsilon) &= \varepsilon \\
\eta(\delta) &= \delta \\
\eta(a) &= r \quad \text{if } a \in I \\
\eta(a) &= a \quad \text{if } a \notin I \\
\eta(x + y) &= \eta(x) \eta(y) \\
\eta(x + y) &= \frac{\eta(x)}{1 + \eta(y)} \\
\end{align*}
\]

Table 8

2.4. Comment.
A lot of individual changes, most but not all of them connected with the introduction of the empty process, made up the transformation of ACP ([BK1]) into ACP\( ^\varepsilon \). Earlier stages of this transformation process were BPA\( ^e \) and PA\( ^e \) in [KVJ]. Unfortunately the merge operator turned out not associative in these systems. (Counterexample: \((a + \varepsilon) \parallel b) \parallel c \neq (a + \varepsilon) \parallel (b \parallel c) \) in PA\( ^{e} \). That is why they are replaced here by BPA\( ^{e} \) and PA\( ^{e} \) as defined in the previous section.
We will try to give some motivation for the more important among these changes.

1. The set of atoms or atomic steps is defined here as a parameter of the axiom systems and not as a subsort or second sort inside the systems. The latter ways of introducing atoms would require an injection function from atoms to processes or a predicate "atomic," both of which one likes to avoid if possible. The injection function would be cumbersome to use and the predicate would introduce all the problems of conditional equations. Moreover the axiom schemes for the renaming operators \((r(a) = a \text{ if } a \in I \text{ for example})\) become hard to formulate as an algebraic specification. That is why the atoms have been chosen to be a parameter of the axiom systems. They simply furnish a number of constants which can be made explicit for each application. In each application one chooses a set of atomic steps that all have a unique name. The possibility of an additional bunch of nameless atoms, present in the subsort or second sort solution has never been used.

2. The constant \( \delta \) is no longer an atom which it was in the systems ACP and ACP\( _\tau \). This makes a lot of things easier and no information is lost. Axioms in ACP like \( ax \parallel y = a (x \parallel y) \) are derivable for \( a = \delta \).
3. The axioms \( \varepsilon + \varepsilon = \varepsilon \) and \( \delta + \varepsilon = \varepsilon \) replace the ACP axioms \( x + x = x \) and \( \delta + x = x \) which are now derivable (multiply the new axioms on the right by \( x \)).

4. The axioms for \( \varepsilon \parallel \ldots \) are chosen such that in CTM one has \( x \parallel \varepsilon = x \), \( \varepsilon \parallel x = x \) and that the merge operator becomes associative. There are other possibilities (see 3.2.4), but the one chosen was considered the most natural solution.

5. The axiom \( a \parallel b = \gamma(a, b) \) must be added as terms like \( a \parallel b \) can not be considered atomic without it.

6. The axiom \( a x \parallel y = (a \parallel y) \parallel x \) replaces the ACP axiom \( ax \parallel by = (a \parallel b)(x \parallel y) \). It fits better the induction scheme mentioned in 2.2 and although not formally equivalent, they are equivalent in CTM.

7. Abbreviative names for the axioms have been abolished, at least for a while. It was too much trouble to maintain them all along the many intermediate stages.

8. One might use yet another parameter to denote the range of the expansion variable \( I \) in \( \gamma \) (now it is \( \phi(A) \)). But as renaming operators are fairly unproblematic in ACP, such a complication cannot be expected to be of much use.

### 3. Supplementary axioms.

3.1. In many applications and theoretical considerations, the basic systems as described in section 2 are extended with a number of supplementary axioms or axiom blocks, e.g., axioms that are provable in CTM, the projection axioms, the priority axioms, etc. We will enounce a number of such axioms and axiom blocks in this section and consider their validity in CTM. In the next section their validity in a graph model, which contains infinite processes, will be studied.

3.2. The proposition below consists of a number of axioms that can be proven to hold in CTM and that are preparatory to the proof of the most important one of this kind of axioms, the associativity of the merge operator.

**3.2.1. Proposition** In CTM, the closed term model of ACP the following hold:

1. \( x \parallel \varepsilon = x \)
2. \( x \parallel \varepsilon = x \)
3. \( \varepsilon \parallel x = \varepsilon \) or \( \varepsilon \parallel x = \delta \)
4. \( \varepsilon \parallel x = \varepsilon \iff x + \varepsilon = x \)
5. \( \varepsilon \parallel (x \parallel (y + z)) = \varepsilon \parallel (x \parallel y) + \varepsilon \parallel (x \parallel z) \)
6. \( \varepsilon \parallel x \delta = \delta \)
7. \( \varepsilon \parallel (x \parallel y) = \delta \)
8. \( \varepsilon \parallel (x \parallel y) = \varepsilon \iff (\varepsilon \parallel x = \varepsilon \& \varepsilon \parallel y = \varepsilon) \)
9. \( \varepsilon \parallel (x \parallel y) = \varepsilon \parallel (x \parallel y) \)
10. \( (\varepsilon \parallel x) \parallel y = \varepsilon \iff (\varepsilon \parallel x = \varepsilon \& \varepsilon \parallel y = \varepsilon) \)

All of these can be proven easily by means of structural induction on closed ACP terms following the scheme \( \varepsilon \delta a x \ x + y \). Maybe some remarks are useful:

- first 3 and 4, then 1 and 2 simultaneously, using 3 and 4.
- 7 needs additional induction for \( y \).
- 9 with 7 and 8.
The Algebra of Communicating Processes with Empty Process

• in 10 distinguish cases $e \parallel x = e$ and $e \parallel x = \delta$.

3.2.2. Proposition. In the closed term model of $ACP^e$ the following hold:

1. $x \delta \parallel y = (x \parallel y) \delta$
2. $y \parallel x \delta = (y \parallel x) \delta$
3. $x \delta \parallel y = (x \parallel y) \delta$
4. $x \delta \parallel y = (x \parallel y) \delta$

The proof of this proposition is possible with simple induction on the structure of closed terms but we will instead prefer to use a representation for the elements of CTM combined with induction on the sum of the lengths of the two terms involved. The length function $len$ on closed terms (not processes!) can be defined inductively in the obvious way:

- $len(e) = len(\delta) = len(a) = 1$
- $len(x \cdot y) = len(x) + len(y)$
- $len(x + y) = len(x) + len(y)$

Each closed term $x$ has a representation of the form:

$$x = \sum_{i=1}^{n} a_i x_i + e_x$$

$n \geq 0, a_i \in A$

Here $e_x$ equals $e$ if $x$ contains $e$ (formally $x+e=x$) and $\delta$ otherwise.

The existence of such a representation is easy to show by means of structural induction. As $n$ may be 0, the bottom terms are taken along in this representation.

(by convention $\sum_{x \in \emptyset} x = \delta$).

Proof: We take

$$x = \sum_{i=1}^{m} a_i x_i + e_x$$

$m \geq 0, a_i \in A$

and $y = \sum_{i=1}^{n} b_i y_i + e_y$

$n \geq 0, b_i \in A$.

Assertions 1, 2 and 3 add up to 4. We will prove 1, 2 and 3 and thereby 4 but 4 will be the only assertion drawn from induction.

1: We want to reduce $x \delta \parallel y = (x \parallel y) \delta$ to a statement that can be drawn from induction or which is true on itself. Both left and right hand sides of the equation are distributive in $x$ for summation. So we can drop out the $\sum$ (which is also typographically easier) and treat the cases $a_i x_i$ and $e_x$ separately.

- $a_i x_i \delta \parallel y = a_i (x_i \parallel y) \delta =$ (by induction) $a_i (x_i \parallel y) \delta = (a_i x_i \parallel y) \delta$
  (we used $len(x_i) < len(x)$)
- $e_x \delta \parallel y = \delta \parallel y = \delta = (e_x \parallel y) \delta$ as $e_x = e$ or $\delta$

2: analogous to 1, using 3.2.1.6.

3: fill in the representation for both $x$ and $y$. □
3.2.3. **Proposition** In the closed term model of ACPE the following hold:
1. \((x \parallel y) \parallel z = x \parallel (y \parallel z)\)
2. \((x \mid y) \parallel z = x \mid (y \parallel z)\)

**Theorem** In the closed term model of ACPE the merge operator is associative.

Although the proposition easily implies the theorem we prefer to prove the theorem in a way completely analogous to the method used in 3.2.2. This proof will contain a proof of the proposition.

Proof of the theorem: The assertion \((x \parallel y) \parallel z = x \parallel (y \parallel z)\) is split up into 7 equations that can be reduced separately to the point where induction can be applied.

\[
\begin{align*}
(x \parallel y) \parallel z &= (x \parallel y) \parallel z + (y \parallel x) \parallel z + (x \mid y) \parallel z + (x \parallel y) + \\
&\quad + (x \parallel y) \mid z + (y \parallel x) \mid z + (x \mid y) \mid z \\
x \parallel (y \parallel z) &= x \parallel (y \parallel z) + (y \parallel z) \parallel x + (z \parallel y) \parallel x + (y \mid z) \parallel x + \\
&\quad + x \mid (y \parallel z) + x \mid (z \parallel y) + x \mid (y \mid z)
\end{align*}
\]

Equality of the two right hand sides is implied by the following 7 equations:

1. \((x \parallel y) \parallel z = x \parallel (y \parallel z)\)
2. \((y \parallel x) \parallel z = (y \parallel z) \parallel x\)
3. \((x \mid y) \parallel z = x \mid (y \parallel z)\)
4. \(z \parallel (x \parallel y) = (z \parallel y) \parallel x\)
5. \((x \parallel y) \mid z = x \mid (z \parallel y)\)
6. \((y \parallel x) \mid z = (y \mid z) \parallel x\)
7. \((x \mid y) \mid z = x \mid (y \mid z)\)

For a reduction of these equations, the representation mentioned in 3.2.2 will be applied:

\[
x = \sum_{i=1}^{n} a_i x_i + e_x \quad n \geq 0, a_i \in A
\]

Starting with equation 1, we see again that both sides are distributive in \(x\) for summation. So if we fill in the representation for \(x\), we do not need to take the \(\Sigma\) along. The cases \(e, \delta\) and \(a_i x_i\) can be treated separately which is straightforward using 3.2.1. The associativity of \(\parallel\) then appears to be the only conclusion drawn from induction. The same story can be told for equation 2 (reducing via \(y\) instead of \(x\)) and 4 (reducing via \(z\)). In order to reduce 3 you have to fill in the representation for both \(x\) and \(y\). Again the \(\Sigma\)'s don't need to be taken along and we get 3 times 3 separate cases of which 8 are
trivial (those in which $\varepsilon$ or $\delta$ occurs) and the ninth \((a_j x_j, b_j y_j)\) is not very hard neither. About the same can be said of 5 and 6. Equation 7 requires substitution of the representation for $x$, $y$ and $z$ and the associativity of the $\gamma$-function (2.3.5). □

Remarks

- The equations in the proposition above seem to be the best replacement for the axiom block which in ACP-context is called Standard Concurrency.
- This method also applies in ACP and ACP$_\tau$. In ACP one can use the representation
  \[ x = \sum a_i + \sum b_j x_j \]

($\delta$ is an atom in ACP). In ACP$_\tau$ the representation becomes
  \[ x = \sum a_i x_i + \sum \tau y_j \]

(a can be represented as at but the bottom term $\tau$ must be treated separately). One cannot make the split up into 7 separate equations immediately as these do not hold in ACP$_\tau$ (for example if $y = \tau$ then equation 3 doesn’t hold). All the nasty subterms, involving both $\tau$ and communication, however happen to be summand of some of the other terms and can therefore be eliminated. The remainder of the proof is then straightforward.

3.2.4. One might choose other axioms concerning $\varepsilon \parallel \ldots$ that also lead to an associative merge operator and to $x \parallel \varepsilon = x$. The following is such an alternative:

\[
\begin{align*}
\varepsilon \parallel x &= \delta \\
\varepsilon \mid \varepsilon &= \varepsilon \\
\varepsilon \mid a x &= \delta
\end{align*}
\]

(These replace all the axioms in ACP$^\varepsilon$ concerning $\varepsilon \parallel \ldots$ and $\varepsilon \mid \ldots$, so ACP$^\varepsilon$ would become two axioms shorter. Associativity of the merge operator can be proven in the same way as in 3.2.3.) This alternative, although technically simpler, has several significant disadvantages. The assertion $x \parallel \varepsilon = x$ no longer holds in this system’s closed term model ($\varepsilon \parallel \varepsilon = \delta$). And in PA$^\varepsilon$ the trick with the communication bar ($\varepsilon \mid \varepsilon = \varepsilon$) cannot be done. This has the effect that $x \parallel \varepsilon = x$ doesn’t hold either in PA$^\varepsilon$-s closed term model. Yet the alternative has been used in [Gr] and [GrP]. They are not concerned with PA-versions of their main axiom systems.

Whether $\varepsilon \parallel \varepsilon = \varepsilon$ is intuitively appealing may be disputable but the alternative $\varepsilon \mid \varepsilon = \varepsilon$ is at least as disputable. One may adopt the view that $\parallel$ and $\mid$ only serve the purpose of axiomatizing $\parallel$. As long as the latter operator has the desired properties (in this context especially associativity and $x \parallel \varepsilon = x$), the details of the axiomatization of $\parallel$ and $\mid$ are insignificant. These two operators are not intended to be used in actual specifications.

A different approach can be found in [BG] and [BW]. It consists of the introduction of a separate operator $\triangledown$ that gives the termination behaviour of a parallel composition:

\[
\triangledown(x) = \varepsilon \text{ if } x = x + \varepsilon = \delta \text{ otherwise}.
\]

Now one can introduce the axioms

\[
\begin{align*}
\varepsilon \parallel x &= \delta \\
\varepsilon \mid x &= \delta \\
x \parallel y &= x \parallel y + y \parallel x + x \mid y + \triangledown(x) \cdot \triangledown(y)
\end{align*}
\]
This system removes the intuitively not quite satisfactory axiom $E \parallel E = E$, at the cost of an extra operator. The operator can be considered as equal to $\partial_A$, i.e. $\delta$-renaming for all atoms.

3.3. Handshaking and the Expansion Theorem.

The *handshaking axiom*

$$a \parallel b \parallel c = \delta$$

(which expresses that each communication takes place between no more than two processes) is used in almost every application of Process Algebra. It serves as a great help in taming exploding calculations. It can cooperate as smoothly with $ACP$ as it does with $ACP$ and $ACP_T$. An easy consequence in CTM of this axiom is the statement $x \parallel y \parallel z = \delta$. Another more involved consequence in the same context is the expansion theorem:

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i \parallel \left( \sum_{j=1, j \neq i}^{n} x_j \right) + \sum_{1 \leq i < j \leq n}^{n} (x_i \parallel x_j) \parallel \left( \sum_{k=1, k \neq i, j}^{n} x_k \right)$$

The proof is a straightforward induction on $n$ using the equations given in 3.2.3.

3.4. The Projection Operators $\pi_n$.

The projection operators $\pi_n$ project a process on the "space" of processes of at most $n$ consecutive atomic steps. The axioms for $\pi_n$ ($n \geq 0$) are:

**The Projection Axioms**

<table>
<thead>
<tr>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi_n(x) = E$</td>
</tr>
<tr>
<td>$\pi_n(E) = E$</td>
</tr>
<tr>
<td>$\pi_{n+1}(\delta) = \delta$</td>
</tr>
<tr>
<td>$\pi_{n+1}(a \cdot x) = a \cdot \pi_n(x)$</td>
</tr>
<tr>
<td>$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$</td>
</tr>
</tbody>
</table>

Structural induction easily shows that $\pi_n$ can be eliminated from $\pi_n(x)$ for every closed term $x$ and that in CTM $\pi_n$ has indeed the typical projection operator property: $\pi_n \circ \pi_n = \pi_n$. More information on the question whether $\pi_n$ is well-defined in CTM and whether it is a conservative extension of $ACP^\varepsilon$ will be given in section 5.6.5.
3.5. The Priority Operators \(<\) and \(\otimes\).

In this section an attempt is made to introduce the empty process in the priority axioms ([BBK2]). These axioms define the properties of the \(<\) and \(\otimes\) operators that are used to express that in a sum expression, some atoms have priority over others. The priority relation among atoms is given by a strict partial order (denoted by \(<\)). The \(\otimes\) operator applied to a process means that the process has to be executed according to the priority rules. This operator is defined by means of the auxiliary unless operator \(<\).

Example: Assume \(a < b\).

Then \(\otimes(a + b) = b\) because in the sum, \(b\) has priority over \(a\).

Using the axioms below:

\[
\begin{align*}
\otimes(a+b) &= \otimes(a) \otimes b + \otimes(b) \otimes a = a \otimes b + b \otimes a = \emptyset + b = b
\end{align*}
\]

### The Priority Axioms

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \otimes b = a)</td>
<td>if not (a &lt; b)</td>
</tr>
<tr>
<td>(a \otimes b = \emptyset)</td>
<td>if (a &lt; b)</td>
</tr>
<tr>
<td>(x \otimes \emptyset = x)</td>
<td></td>
</tr>
<tr>
<td>(x \otimes \emptyset = x)</td>
<td></td>
</tr>
<tr>
<td>(x \otimes \emptyset = x)</td>
<td></td>
</tr>
<tr>
<td>(x \otimes (y + z) = (x \otimes y) \otimes z)</td>
<td></td>
</tr>
<tr>
<td>(\emptyset \otimes x = \emptyset)</td>
<td></td>
</tr>
<tr>
<td>(\emptyset \otimes x = \emptyset)</td>
<td></td>
</tr>
<tr>
<td>(\emptyset \otimes x = \emptyset)</td>
<td></td>
</tr>
<tr>
<td>(x \otimes (y + z) = x \otimes z + y \otimes z)</td>
<td></td>
</tr>
<tr>
<td>(\otimes(x) = x)</td>
<td></td>
</tr>
<tr>
<td>(\otimes(x) = x)</td>
<td></td>
</tr>
<tr>
<td>(\otimes(x) = x)</td>
<td></td>
</tr>
<tr>
<td>(\otimes(x + y) = \otimes(x) \otimes y + \otimes(y) \otimes x)</td>
<td></td>
</tr>
</tbody>
</table>

One easily checks that from closed terms the \(\otimes\) operators can be removed. The ACP\(\otimes\) axiom \(\otimes(x \cdot y) = \otimes(x) \cdot \otimes(y)\) had to be replaced by

\[\otimes(a \cdot x) = a \cdot \otimes(x)\]

because of the following counterexample:

Suppose \(b < a\).

\[\otimes((a + b) \cdot b) = a \cdot b \otimes b \otimes a \cdot b = (a \otimes b) \cdot b + b \otimes a = a \cdot b, \quad \text{but}\]

\[\otimes((a + \emptyset) \cdot b) = (\text{according to } \otimes(x \cdot y) = \otimes(x) \cdot \otimes(y)) = (a + \emptyset) \cdot b = a \cdot b + b \quad \text{which is not equal to } a \cdot b.\]

More information about the conservativity of the priority axioms will be given in section 5.6.5.
3.6. The ε-counterpart of Koomen's Fair Abstraction Rule or KFAR ([BBK3]) is called The Epsilon Abstraction Rule or EAR. In its formulation we will use infinite sums although these have not been defined yet. This is not a mortal sin however. The reader can easily adapt the formulation of EAR such that the sum is no longer potentially infinite. Moreover, the introduction of infinite sums is currently subject of research.

\[
\begin{align*}
\text{EAR} \\
\varepsilon \left( x_0 \right) &= \sum_{n \in \mathbb{N}} \varepsilon \left( y_n \right) \\
\end{align*}
\]

Here \( \mathbb{N} \) denotes the set of natural numbers. The sum can be made finite for example by the requirement that the set \( \{ \varepsilon (y_n) \} \) be finite. Little can be said about EAR in CTM as it essentially considers an infinite process. In section 4, EAR will be proven to hold in the graph model defined there and we will see that on the whole it looks a lot easier than the \( \tau \)-version KFAR, not unlike what one would expect.

EAR has been used extensively in [BBK4].

3.7. Renaming Operators.

3.7.1. In section 4, a general simultaneous renaming operator \( \rho_f \) will be defined (see 4.4.7) of which \( \varepsilon_I \) and \( \delta_I \) are special cases. This operator however is hard to introduce by means of an axiom system as it takes a function \( f : A \rightarrow P \) which fits badly in an axiom system's signature (\( A \) is not a second sort). That is why we restrict ourselves here to the axiomatic introduction of a less general renaming operator \( \eta_I \), still sufficiently general to be an interesting extension of ACP\(^e \). It renames the atoms of \( I \subseteq A \) occurring in a process \( x \) into some process \( r \). It can be introduced by adding to the signature a function \( R_I : P \times P \rightarrow P \) such that \( \eta_I(x) \) is an abbreviation for \( R_I(r, x) \). The set of axioms has to be extended with the renaming axioms (the set \( I \) ranges over an application dependent, user defined subset of \( \mathcal{F}(A) \)):

The Renaming Axioms

\[
\begin{align*}
\eta_I(\varepsilon) &= \varepsilon \\
\eta_I(\delta) &= \delta \\
\eta_I(a) &= r & \text{if } a \in I \\
\eta_I(a) &= a & \text{if } a \notin I \\
\eta_I(x, y) &= \eta_I(x) \cdot \eta_I(y) \\
\eta_I(x + y) &= \eta_I(x) + \eta_I(y)
\end{align*}
\]

Any operator \( \eta_I \) can probably be added conservatively to ACP\(^e \) (see 5.6.5) due to the fact that all the atoms in the axioms of \( E_{ACP^e} \) are protected by operators other than + or \( \cdot \) (a renaming operator can
not be removed from a term containing such operators). This is unlike the situation in the system ACP_τ where for instance the third τ-law
\[ a(\tau x + y) + a x = a(\tau x + y) \]
forbids ab-renaming. It would lead to
\[ a b(\tau x + y) + a b x = a b(\tau x + y) \]
which is an unwanted intimacy among terms. It does not hold for instance in the graph model of ACP_τ defined in [BBK3].

3.7.2. An important aspect of a renaming operator is its ability to remove or to introduce deadlocks in a term. We will study this property in the CTM context although proofs and an intuitive support for the definitions in this section have to be deferred to section 5.6.6. It is a curious fact that providing proofs via structural induction for the propositions below seems intractable.

**Definition** The predicate ND (no deadlocks) on CTM is the least predicate such that:

1. ND(ε)
2. ND(x) ⇒ ND(ax)
3. ND(x) & ND(y) ⇒ ND(x + y)

ND(x) means that x does not contain any sturdy δ-'s (weak δ-'s like the one in a + δ can immediately be removed).

**Proposition** Let r be a process. If ND(r) then for all x ∈ CTM:
\[ ND(x) ⇒ ND(τ_l(x)) \]
For a proof see section 5.6.6.

3.7.3. The only elements r of CTM such that τ_l may eliminate deadlocks are ε and δ. For instance:
\[ ε(a)(\ aδ + b) = b \]
\[ δ(a)(\ aδ + b) = b \]
The fact that the other elements of CTM don't, is expressed in the proposition below.

**Definition** The predicate D on CTM is the least predicate such that

1. D(δ)
2. D(x) ⇒ D(ax + y)

D is intended as the negation of ND. That this is true will be proven in section 5.6.6.

**Proposition** If r ≠ ε, δ then
\[ D(x) ⇒ D(τ_l(x)) \]
For a proof see section 5.6.6. The value of $\varepsilon$-renaming as an abstraction mechanism instead of $\tau$-renaming is made questionable by the proposition above. However $\varepsilon$-renaming has already shown its usefulness in a different way in [BBK4].

3.7.4 Another useful and easy to prove proposition:

**Proposition** Let $I$ and $J$ be two subsets of $A$ and $r, s \in \text{CTM}$.

- $r I (r) = r \quad \Rightarrow \quad r I o r J = r I \cup J$
- $r I o s I = (r I (s)) I$

More about renaming operators in ACP-context, especially atom to atom renaming, can be found in [BB]. The predicates ND and D are also considered in [BW].

4. A graph model for $\text{ACP}^\varepsilon$.

4.1 The importance of a graph model for an axiom system in Process Algebra hardly needs any explanation or emphasis. Many case-studies of varying nature are described making heavy use of graphs. This is not typical for Process Algebra, most concurrency theories do so. In Process Algebra such a graph language is based on a graph model for the axiom system in use.

In this section a graph model is constructed for $\text{ACP}^\varepsilon$ more or less along the usual lines to do this (confer [BBK3]). Obviously this will also be a (somewhat overcomplicated) graph model for $\text{BPA}^\varepsilon$, $\text{BPA}^\varepsilon_\delta$ and (taking the communication function $\gamma = \delta$) $\text{PA}^\varepsilon$.

A graph model for $\text{ACP}^\varepsilon$ is in short a set of graphs modulo a certain equivalence relation, called $\varepsilon$-bisimulation, in which the operators and constants can be defined and in which the axioms then hold. We will first define the kind of graphs we will use and the bisimulation relation on it. Then we can show how the signature $\Sigma_{\text{ACP}^\varepsilon}$ is represented in this set of graphs modulo $\varepsilon$-bisimulation and that the operators, first defined on graphs, respect the bisimulation relation. Finally the axioms will be proven in this model.

4.2.1. The graphs used for the model, called process graphs, consist of a set of nodes, a set of edges and some functions connecting these two sets. The edges are directed and have a starting node and an arrival node. Each edge has in addition a label, an element of the set $A_{\varepsilon\delta} = A \cup \{ \varepsilon, \delta \}$. One of the nodes is called the root.

Within this setting there are no further restrictions on a graph. Cycles, a cyclic root, infinite paths, disconnected parts (parts of a graph not accessible from the root), the trivial graph (a single node) are all allowed. Neither is there a restriction on the outdegree of a node, the cardinality of the set of edges starting from it. From the model that will be constructed here one can obtain a large number of submodels by restricting the outdegree to be less than some infinite cardinal $\kappa$. This restriction is often made in order to assure that the set of graphs underlying the model is a set in the Zermelo-Fraenkel sense and not a proper class (which it is in our case, if there is at least one atom, even modulo bisimulation). As this $\kappa$ would play in this paper no other role than satisfying a somewhat emotional preference for sets, we have chosen to omit it, rather hoping that a real problem will be caused by this omission. Moreover we will not be precise about the distinction between sets and classes, using the word set even for objects which in some cases might be classes.
4.2.2. In order to talk conveniently about graphs the following generic functions are defined (and illustrated below):

Let \( f \) be a process graph, \( n \) one of its nodes, \( e \) one of its edges. Then we denote by:

- \( N(f) \) or \( N_f \): the set of nodes of \( f \),
- \( E(f) \) or \( E_f \): the set of edges of \( f \),
- \( r(f) \) or \( r_f \): the root of \( f \),
- \( sn(e) \): the starting node of \( e \),
- \( an(e) \): the arrival node of \( e \),
- \( lb(e) \): the label of \( e \),
- \( se(n) \): the set of edges starting at \( n \),
- \( ae(n) \): the set of edges arriving at \( n \),
- \( ind(n) \): the indegree of \( n \), the cardinality of \( ae(n) \),
- \( outd(n) \): the outdegree of \( n \), the cardinality of \( se(n) \).

We say that a graph \( f \) can be mapped into a graph \( g \) if there are injective mappings

\[
\begin{align*}
m_n &: N_f \rightarrow N_g \\
m_e &: E_f \rightarrow E_g
\end{align*}
\]

such that:

\[
m_n(r_f) = r_g
\]

and

\[
m_n \circ sn = sn \circ m_e \quad \& \quad m_n \circ an = an \circ m_e.
\]

(Easy examples show that leaving out the word "injective" would define a different notion.)

It can be mapped into \( g \) with preservation of labels if in addition

\[
\forall e \in E_f : lb(e) = lb(m_e(e)).
\]

The set of labels is almost always \( A_{\epsilon\delta} = \{ \epsilon, \delta \} \). Edges labeled with an atom \( a, \epsilon \) or \( \delta \) are called \( a \)-edges (or atomic edges), \( \epsilon \)-edges and \( \delta \)-edges respectively. Moreover, we often use the word step instead of edge.

Saying that node \( n \) is accessible from node \( m \) means that there is a path from \( m \) to \( n \) not containing any \( \delta \)-steps. \( \epsilon \)-accessibility means that the path consists solely of \( \epsilon \)-steps.

The picture below is supposed to illustrate most of these notions:
The open dots denote the roots $r_f$, $r_{g1}$ and $r_{g2}$ of these graphs. The capital letters are just names for the nodes that they are closest to, they are not part of the graph $f$. The characters close to the arrows are their labels. The node $A$ is arrival node of a $c$-labeled edge and starting node of an $e$-labeled edge. $se(r_f)$ consists of an $e$-edge and a $\delta$-edge. $ae(r_f)$ consists of only an $a$-edge. Therefore $ind(r_f) = 1$ and $outd(r_f) = 2$. One can see that the graphs $g_1$ and $g_2$ can be mapped into $f$ but only $g_1$ can be mapped into $f$ with preservation of labels. All the nodes in $f$ are accessible from $r_f$ with the exception of the nodes following the $\delta$-edge. The node $B$ is $e$-accessible from the root.

4.2.3 The set (or class) of process graphs with labels from $L$ is denoted by $G(L)$. $G$ denotes $G(A,\delta)$. The set of process graphs with the outdegree of the nodes is less than $\kappa$ is denoted by $G_{\kappa}(L)$ or $G_{\kappa}$. The subset of cycle-free graphs is denoted $G_{cf}(L)$ or $G_{cf}$. In the definition of the merge operator on graphs we will need the subset $G_{I}$ of $G$ which consists of the graphs with the following properties:

1. the graphs are cycle-free;
2. no node of any graph has both incoming $e$-edges and atomic edges;
3. no node can be accessed from the root via both a pure $e$-path and via a path containing atomic edges;
4. all nodes are accessible from the root.

The subset of trees of $G(L)$, $G$, $G_{\kappa}(L)$, $G_{\kappa}$ etc are denoted $T(L)$, $T$, $T_{\kappa}(L)$, $T_{\kappa}$ etc.

4.3.1 $e$-Bisimulation.

**Definition** Let $f$, $g \in G$ and $R$ a relation between $N_f$ and $N_g$. We call $f$ and $g$ $e$-bisimilar by the relation $R$ (or $R$-bisimilar) if $R$ is such that

1. $r_f \ R \ r_g$
2. if $m \ R \ n$ for nodes $m \in N_f$ and $n \in N_g$ and $p$ is a path in $f$ from $m$ to $m'$ consisting of a finite number of $e$-steps followed by a step $a \in A$, then there must be a path $q$ in $g$ from $n$ to $n'$, also consisting of a finite number of $e$-steps followed by a step $a$. And $m' \ R \ n'$.
3. if $m \ R \ n$ and there is a path in $f$ from $m$ to an end node, consisting of a finite number of $e$-steps, then there must also be a path in $g$ from $n$ to an end node, consisting of a finite number of $e$-steps.
4. 5: the same as 2 and 3 but with the roles of $f$ and $g$ interchanged.

If such a relation $R$ exists we say that $f$ and $g$ are $e$-bisimilar, notation

$f \ e\equiv R \ g$.

4.3.2 One can see that in comparison with the definition given in [KV] 2.2.2, the present definition allows for two additional features of process graphs, namely infinite $e$-paths and $\delta$-edges. The following examples show how this works out. An open dot denotes the root. Consecutive dots (...) denote infinite continuation.
Examples

(a) 

(b) 

(c) 

(d) 

(e) 

(f)
4.3.3. Remarks.
1. $\varepsilon$-Bisimulation is an equivalence relation between graphs and the identity relation on the nodes of a graph is an $\varepsilon$-bisimulation.
2. Sharing, which means identifying nodes with identical or even just $\varepsilon$-bisimilar subgraphs, respects $\varepsilon$-bisimilarity. The sharing may even be partial; identifying just some nodes, for instance some of the end nodes, also respects $\varepsilon$-bisimilarity.
3. Unfolding (complete or partial), the opposite of sharing, respects $\varepsilon$-bisimilarity.
4. If $f \leftrightarrow_\varepsilon g$ and $g$ is the subgraph of a node in a graph $h$, then $h' \leftrightarrow_\varepsilon h$ where $h'$ denotes $h$ with $g$ replaced by $f$.
5. The $\varepsilon$-bisimulation relation in fig. 1 (h) relates the two roots and it relates the nodes of $y_i$ in the graph on the left with the same nodes of $y_i$ in the graph on the right, making an exception for the roots of the graphs $y_i$ for $i \geq 2$. This picture is in essence the proof of EAR in the graph model (see section 4.11). It is assumed that the graphs $y_i$ have acyclic root and are not the single node graph. This in order to get an easier picture.
6.Disconnected parts of a graph or parts that can only be accessed from the root via a $\delta$-step can be removed without changing its bisimilarity class (except the terminal nodes of $\delta$-edges, whose starting nodes are accessible from the root).
7. The union of a non-empty set of $\varepsilon$-bisimulations between two graphs is again an $\varepsilon$-bisimulation. The union of all $\varepsilon$-bisimulations between two bisimilar graphs is called the saturated bisimulation.
8. If two graphs are $R$-bisimilar then the subgraphs of any pair of related nodes are $R'$-bisimilar where $R'$ is the restriction of $R$ to the two subgraphs. This is unlike $\tau\rho\delta$-bisimulation described in [BBK3].
9. In [BG] and [BW], an alternative graph model for $\text{ACP}_c$ is defined that does not have $\varepsilon$- or $\delta$-edges. In [BG], the equivalence of the two graph models is shown.

4.3.4. A special $\varepsilon$-bisimulation between elements of $G_\parallel$ is needed in proofs concerning the merge type operators ($\parallel$, $\ll$, $\llp$). Intuitively, this relation, called the label preserving relation, makes sure that only nodes with the same type of incoming edges (see 4.2.3) are related. Moreover, it prohibits relating some nodes that do not need to be related and could cause trouble in the definition of the left merge operator on graphs.
**Definition** Let $f, g \in G\parallel$ with $f \equiv g$. Inductively we define a sequence of relations:

$\mathcal{R}_0$ : $\mathcal{R}_0$ only relates the roots: $r_f \mathcal{R}_0 r_g$.

$\mathcal{R}_{n+1}$ : $\mathcal{R}_{n+1}$ relates all pairs of nodes already related by $\mathcal{R}_n$ and also the pairs $(k_1, k_2)$ such that $k_1$ and $k_2$ have bisimilar subgraphs and have a pair $(k_1', k_2')$ of $\mathcal{R}_n$ related predecessors from which they can be accessed via paths of the form $\varepsilon^{m_1} \cdot a$ and $\varepsilon^{m_2} \cdot a$ ($m_1, m_2 \in \mathbb{N}$) respectively.

The label preserving relation $\mathcal{R}_{lp}$ is now defined as

$$\mathcal{R}_{lp} := \bigcup_{n \in \mathbb{N}} \mathcal{R}_n$$

**Proposition**

1. $\mathcal{R}_{lp}$ is an $\varepsilon$-bisimulation.
2. $m \mathcal{R}_{lp} n$ implies that $m$ and $n$ are both the root of their graphs xor they have both an incoming atomic edge.

The proof is easy. □

4.4. Now we can show how to represent the constants $\varepsilon, \delta$, the elements of $A$ and the operators $+, \cdot, \parallel, \triangleright, 1, \varepsilon_1$ and $\delta_1$ in $G$ and thereby in $G/\equiv_e$.

4.4.1. The examples in fig. 1 make it sufficiently clear how the constants are represented. The constant $\delta$ is represented by a two-nodes graph, connected by one edge with the label $\delta$. The constant $\varepsilon$ is represented by a two-nodes graph, connected by one edge with the label $\varepsilon$. The latter graph happens to be $\varepsilon$-bisimilar with the single node graph.

4.4.2. Let $f, g \in G$. Then *sum* $f + g$ and *product* $f \cdot g$ look like

![Diagram](attachment:image.png)

In words: $f + g$ is constructed by creating a new root and having $\varepsilon$-edges pointing from it to the roots of $f$ and $g$. As usual the extra $\varepsilon$-edge may be left out if the graph to which it points has an acyclic root and is not the single node graph. The product $f \cdot g$ is constructed by appending $g$ with its root to all the endnodes of $f$.

4.4.3. The *merge* $f \parallel g$ of two graphs $f, g$ will only be defined for the elements of the subset $G\parallel$ of $G$ (see 4.2.3). $G\parallel$ is such that each element of $G/\equiv_e$ has a representative in $G\parallel$ (each element of $G$ can be unwound into a tree and $T \subset G\parallel$) and $G\parallel$ will be closed for the merge operator. This will
prove of great value when checking some of the harder axioms. First we have to introduce some terminology for cartesian products.

If \( f, g \in G \) then the cartesian product \( f \times g \) can be represented as follows:

\[
\begin{align*}
N(f \times g) &= N(f) \times N(g) \\
E(f \times g) &= N(f) \times E(g) \cup N(g) \times E(f).
\end{align*}
\]

(\( \cup \) denotes disjoint union). The sn, an and \( lb \) functions (see 4.2.2) for \( f \times g \) are defined in the obvious way. For example if \( m \in N(f) \) and \( e \in E(g) \) then \( (m, e) \in E(f \times g) \) and \( sn((m, e)) = (m, sn(e)) \). We say that an edge \((m, e)\) stems from \( f \) (or \( g \)) if it is an element of \( N(g) \times E(f) \) (or \( N(f) \times E(g) \)). Two edges in \( f \times g \) are called orthogonal if they stem from different components. All these definitions can easily be extended to products of more than two components. And it is then not hard (nor pleasant) to prove that the cartesian product on graphs is commutative and associative, modulo graph-isomorphism.

Let \( f, g \in G \). We define the (communication-) \textit{free merge} \( f \triangleright f g \) as the cartesian product \( f \times g \) where all atomic edges, starting orthogonally to an arriving \( e \)-edge are changed into \( a \)-edges.

From the free merge graph we obtain the \textit{merge graph} \( f \triangleright f g \) by the addition of diagonal edges as follows:

if from a node \((m, n)\) there start in \( f \times g \) (not \( f \triangleright f g \) !) edges

\[
(m, n) \xrightarrow{a} (m', n) \quad \text{and} \quad (m, n) \xrightarrow{b} (m, n')
\]

then we add in \( f \triangleright f g \) the edge

\[
(m, n) \xrightarrow{a l b} (m', n')
\]

The result looks like:

\[
\text{fig. 3}
\]

The atomic edges with labels \( a \) or \( b \) in fig. 3 may however have become \( \delta \)-edges in the free merge graph and then will remain so in the merge graph. For simplicity we add these diagonal edges even if \( a \mid b = \delta \). Examples, together with examples of the left merge and the communication merge are given in 4.4.5.

4.4.4. The \textit{left merge} \( f \ll g \) (for \( f, g \in G_\| \)) is obtained from the merge graph \( f \triangleright g \) by

1. making \( \delta \) all the communication edges accessible from the root via a finite \( e \)-path
The Algebra of Communicating Processes with Empty Process

2. making δ all the atomic edges stemming from g and accessible from the root (of \( f \parallel g \)) via a finite ε-path.

The communication merge \( f \mid g \) is obtained from the merge graph \( f \parallel g \) by making δ all the non-diagonal atomic edges and terminal ε-edges accessible from the root via a finite ε-path.

4.4.5. Examples.

(An open dot denotes the root.)

\[
\begin{align*}
\text{fig. 4} & \\
\text{fig. 5}
\end{align*}
\]
The Algebra of Communicating Processes with Empty Process

The cartesian product $f \times g$:

$$\begin{array}{|c|c|c|c|}
\hline
& a & b & c \\
\hline
\varepsilon & b & d & e \\
\hline
\varepsilon & a & e & f \\
\hline
\varepsilon & g & h & i \\
\hline
\end{array}$$

$f \parallel_f g = (a + b) \parallel_f (c + d) = a (c + d) + b (c + d) + c (a + b) + d (a + b)$:

$$\begin{array}{|c|c|c|c|}
\hline
& a & b & c \\
\hline
\delta & b & d & e \\
\hline
\delta & a & e & f \\
\hline
\delta & g & h & i \\
\hline
\end{array}$$
The Algebra of Communicating Processes with Empty Process

\[ f \parallel g = (a + b) \parallel (c + d) = a (c + d) + b (c + d) + c (a + b) + d (a + b) + a + c + a + d + b + c + b + d: \]

![Diagram 9](image)

(All \( \delta \)-edges and some \( \varepsilon \)-edges have been left out.)

\[ f \parallel g = (a + b) \parallel (c + d) = a (c + d) + b (c + d): \]

![Diagram 10](image)

\[ f \parallel g = (a + b) \parallel (c + d) = a \parallel c + a \parallel d + b \parallel c + b \parallel d: \]

![Diagram 11](image)

The next example shows that condition 3 in the definition of \( G_{\parallel} \) (see 4.2.3), cannot be simply omitted.
The Algebra of Communicating Processes with Empty Process

The merge graph is still ok; condition 3 of definition 4.2.3 does not seem necessary for the merge operator nor for the communication merge although this seems less obvious (might even be wrong). But the left merge operator certainly goes wrong without condition 3:

The present definition of the merge operator on graphs, based on the new axiomatization of the left merge with respect to $\varepsilon$, appears to be simpler than the one given in [KV] 3.2.1, at least after subtraction of the complications caused by the addition of communication. The difference in treatment between intermediate and terminal $\varepsilon$-edges is no longer needed. This in contrast with the more complicated axiomatization.

$G_{\parallel}$ is closed under the merge operator. The proof is easy. The same holds for the left merge and the communication merge operators.

4.4.6. Remarks.

1. The present definition of the merge operator on graphs, based on the new axiomatization of the left merge with respect to $\varepsilon$, appears to be simpler than the one given in [KV] 3.2.1, at least after subtraction of the complications caused by the addition of communication. The difference in treatment between intermediate and terminal $\varepsilon$-edges is no longer needed. This in contrast with the more complicated axiomatization.

2. $G_{\parallel}$ is closed under the merge operator. The proof is easy. The same holds for the left merge and the communication merge operators.
The Algebra of Communicating Processes with Empty Process

3. The ε-structure as well as the end node structure are identical in the cartesian product graph and in the merge graph. That means, in the transformation from cartesian product graph to merge graph, no ε-edges disappear or are created, and the same holds for end nodes.

4. It is possible to give an equivalent definition of the merge operator in which ε-edges that start orthogonally to an arriving intermediate ε-edge are made δ. This leads to a simpler merge graph but the proofs of the axioms become much more involved.

5. If the left-hand operand f of f ⊚ g is not the single node graph, then the definition of ⊚ can be simplified. The whole copy of g which starts at the root of f ⊚ g can be removed together with all the edges starting orthogonally from this copy. The proof consists of a simple bisimulation argument.

4.4.7. The four equations

\[
\begin{align*}
    x \parallel (y \parallel z) &= (x \parallel y) \parallel z \\
    x \parallel (y \perp z) &= (x \parallel y) \perp z \\
    x \perp (y \parallel z) &= (x \perp y) \parallel z \\
    x \perp (y \perp z) &= (x \perp y) \perp z
\end{align*}
\]
(see 3.2.3, theorem)

have in common that they all hold in \( G/\parallel \), that they all involve two merge type operators on both sides of the equals sign (which means 3-dimensional graphs in \( G \)) and that they are hard to prove in \( G/\parallel \). Their proof requires a sort of administration system which automatically generates all the many cases that have to be considered. In the next section alternative definitions for \( \parallel \), \( \perp \) and \( \perp \) will be given that will be the core of such an administrative system. These definitions are based on a variation of the cartesian product, to be called "full product" and denoted by 0, which consists of the cartesian product with all diagonal edges filled in:

**Definition** Let \( f, g \in G \). The full product \( f \bowtie g \) of \( f \) and \( g \) is a label-free graph which can be represented as follows:

- **Nodes**: \( N(f) \times N(g) \)
- **Edges**: \( E(f) \times E(g) \cup E(f) \times N(g) \cup N(f) \times E(g) \)

The \( sn \) and \( an \) functions are defined in the obvious way. The edges in \( E(f) \times E(g) \) are called diagonal edges. The canonical label function \( \lambda_0 \) is defined in the obvious way after extending the communication function \( \gamma \) as follows:

\[
\begin{align*}
\gamma(\varepsilon, \ldots) &= \gamma(\ldots, \varepsilon) = \delta \\
\gamma(\delta, \ldots) &= \gamma(\ldots, \delta) = \delta
\end{align*}
\]

This label function is not considered part of the object \( f \bowtie g \). If \( l \) is any function \( E(f) \times E(g) \to A_{\varepsilon\delta} \) the graph \( f \bowtie g \) with label function \( l \) is denoted \( (f \bowtie g, l) \).

The function \( \lambda_0 \) is a so-called generic function: you have to look at its argument to know exactly what function it actually denotes.

Although a bit more complicated, the full product does not seem less natural than the cartesian product. The objects nodes and edges in \( f \bowtie g \) can be enumerated in a perfectly straightforward way (\( n_f \in N_f, n_g \in N_g, e_f \in E_f, e_g \in E_g \)):

- \( (n_f, n_g) \) nodes
- \( (n_f, e_g) \) edges stemming from \( g \)
- \( (e_f, n_g) \) edges stemming from \( f \)
The Algebra of Communicating Processes with Empty Process

diagonal edges.

The graph \((f \circ g, lb_0)\) is meaningless in Process Algebra and it does not carry over to \(G/\mathcal{E}_e\). It serves just as a primitive object to give easier definitions of \(\parallel\), \(\|\) and \(\mathcal{L}\). It is not hard to show that the operator \((f, g) \rightarrow (f \circ g, lb_0)\) is commutative in \(G\). Somewhat more involved is a proof of associativity which is also left to the (poor) reader.

Summing up nodes and edges in \(x \circ y \circ z\) looks as follows:

\[
\begin{align*}
(n_x, n_y, n_z) & \quad \text{nodes} \\
(n_x, n_y, e_z) & \quad \text{edges stemming from } z \\
(n_x, e_y, n_z) & \quad \text{edges stemming from } y \\
(e_x, n_y, n_z) & \quad \text{edges stemming from } x \\
(n_x, e_y, e_z) & \quad \text{diagonal edges between } y \text{ and } z \\
(e_x, n_y, e_z) & \quad \text{diagonal edges between } x \text{ and } z \\
(e_x, e_y, n_z) & \quad \text{diagonal edges between } x \text{ and } y \\
(e_x, e_y, e_z) & \quad \text{triple communication edges}.
\end{align*}
\]

It will be handy to put objects stemming from the i-th operand in \(x_i \circ x_{i+1} \circ \ldots \circ x_k\) at the i-th place. And to use self-explanatory variables for nodes and edges. E.g. the variable \(n_x\) denotes a node in \(x\).

4.4.8. Using the full product, one can give alternative definitions for the merge, left merge and communication merge operators. We regret that readability of these definitions is greatly hampered by their formal wording but this is a consequence of the proof method in which they will be used. Whenever it is not clear to which graph a predicate pertains in the definition below or elsewhere, one should always take the full product with its canonical label function. It is one of the main points of this method that this simplification is possible. For instance, one can check below that the predicate \(oe\) is identical when applied to edges of either \(x \circ y, x \parallel y, x \| y\) or \(x \mathcal{L} y\).

**Definition** (Alternative definition, for original definition see 4.4.3/4)

Let \(x, y \in G\). Then we define the following predicates on \(N_x, E_x\) and \(E_x \circ y\):

- \(t(\ n_x) \iff n_x\) is an end node
- \(t(\ e_x) \iff e_x\) is a terminal edge
- \(a (\ n_x) \iff n_x\) is accessible from the root via zero or more \(\mathcal{E}\)-steps and zero or more atomic steps
- \(a (\ e_x) \iff a (\ sn (\ e_x))\)
- \(\mathcal{E}a (\ n_x) \iff n_x\) is \(\mathcal{E}\)-accessible from the root, using only zero or more \(\mathcal{E}\)-steps
- \(\mathcal{E}a (\ e_x) \iff \mathcal{E}a (\ sn (\ e_x))\)
- \(aa (\ n_x) \iff a (\ n_x) \mathcal{E}a (\ n_x)\)
- \(aa (\ e_x) \iff aa (\ sn (\ e_x))\)
- \(t\mathcal{E}a (\ e_x) \iff e_x\) is a terminal \(\mathcal{E}\)-edge, \(\mathcal{E}\)-accessible from the root
- \(\mathcal{E}a (\ e_x)\) (\(\iff \mathcal{L}b (\ e_x) = \epsilon \& t (\ e_x) \& \mathcal{E}a (\ e_x)\))
- \(\mathcal{I} n (\ n_x) \iff n_x\) has incoming \(\mathcal{E}\)-edges
- \(\mathcal{I} n (\ e_x) \iff \mathcal{I} n (\ sn (\ e_x))\)
- \(\mathcal{O} e (\ n_x, e_y) \iff n_x, e_y\) has orthogonal incoming \(\mathcal{E}\)-edges at its starting node
- \(\mathcal{O} e (\ n_x) \iff \mathcal{I} n (\ n_x)\)
The merge graph \( x \parallel y \) is defined as the graph \((x \hat{\otimes} y, \text{lb}_x)\) with

\[
\begin{align*}
1 \quad \text{lb}_x(n_x, e_y) &= \varepsilon \quad \text{iff} \quad \text{lb}(e_y) = \varepsilon \\
&= \alpha \quad \text{iff} \quad \text{lb}(e_y) = \alpha \land \neg \text{in}(n_x) \\
&= \delta \quad \text{otherwise (this case is omitted below)} \\
2 \quad \text{lb}_x(e_x, n_y) &= \varepsilon \quad \text{iff} \quad \text{lb}(e_x) = \varepsilon \\
&= \alpha \quad \text{iff} \quad \text{lb}(e_x) = \alpha \land \neg \text{in}(n_y) \\
3 \quad \text{lb}_x(e_x, e_y) &= \gamma(e_x, e_y) \quad (\gamma(e_1, e_2) \text{ stands for } \gamma(\text{lb}(e_1), \text{lb}(e_2))
\end{align*}
\]

The left merge graph \( x \parallel y \) is defined as the graph \((x \hat{\otimes} y, \text{lb}_L)\) with

\[
\begin{align*}
1 \quad \text{lb}_L(n_x, e_y) &= \varepsilon \quad \text{iff} \quad \text{lb}(e_y) = \varepsilon \\
&= \alpha \quad \text{iff} \quad \text{lb}(e_y) = \alpha \land \neg \text{in}(n_x) \land \neg \text{e}(n_x, e_y) \\
2 \quad \text{lb}_L(e_x, n_y) &= \varepsilon \quad \text{iff} \quad \text{lb}(e_x) = \varepsilon \\
&= \alpha \quad \text{iff} \quad \text{lb}(e_x) = \alpha \land \neg \text{in}(n_y) \\
3 \quad \text{lb}_L(e_x, e_y) &= \gamma(e_x, e_y) = \alpha \land \neg \text{e}(e_x, e_y)
\end{align*}
\]

The communication merge graph \( x \mid y \) is defined as the graph \((x \hat{\otimes} y, \text{lb}_I)\) with

\[
\begin{align*}
1 \quad \text{lb}_I(n_x, e_y) &= \varepsilon \quad \text{iff} \quad \text{lb}(e_y) = \varepsilon \\
&= \alpha \quad \text{iff} \quad \text{lb}(e_y) = \alpha \land \neg \text{in}(n_x) \land \neg \text{e}(n_x, e_y) \\
2 \quad \text{lb}_I(e_x, n_y) &= \varepsilon \quad \text{iff} \quad \text{lb}(e_x) = \varepsilon \\
&= \alpha \quad \text{iff} \quad \text{lb}(e_x) = \alpha \land \neg \text{in}(n_y) \\
3 \quad \text{lb}_I(e_x, e_y) &= \gamma(e_x, e_y)
\end{align*}
\]

4.4.9. The renaming operators \( \varepsilon_1 \) and \( \delta_1 \) will be defined in \( G/\varepsilon_1 \) by defining a general simultaneous renaming operator \( \rho_f \) (where \( f \) is a function \( A \rightarrow G/\varepsilon_2 \)), this being about as easy and twice as efficient. The operator \( \rho_f \) replaces edges in a graph, with label \( a \) (\( a \in A \)), for a graph representative of \( f(a) \). How this is done exactly is explained below. The operators \( \varepsilon_1 \) and \( \delta_1 \) are special cases of \( \rho_f \) for functions \( f = f_\varepsilon, f_\delta \) defined by:

\[
\begin{align*}
f_\varepsilon &: \quad a \rightarrow \varepsilon \quad \text{if } a \in I \\
&= a \rightarrow a \quad \text{if } a \not\in I, \\
f_\delta &: \quad a \rightarrow \delta \quad \text{if } a \in I \\
&= a \rightarrow a \quad \text{if } a \not\in I,
\end{align*}
\]

The operator \( \rho_f \) will be shown (in 4.6.8) to have the following properties:

\[
\begin{align*}
1. \quad \rho_f(\varepsilon) &= \varepsilon \\
2. \quad \rho_f(\delta) &= \delta \\
3. \quad \rho_f(x \cdot y) &= \rho_f(x) \cdot \rho_f(y) \\
4. \quad \rho_f(x + y) &= \rho_f(x) + \rho_f(y) \\
5. \quad \rho_{\text{id}}(x) &= x \quad \text{(id is the canonical embedding } A \rightarrow G/\varepsilon_2) \\
6. \quad \rho_f \circ \rho_g &= \rho_h \quad \text{where } h = \rho_f \circ g
\end{align*}
\]
These properties cannot be called axioms as the function \( f: A \rightarrow P \) fits badly in an algebraic specification. If \( A \) becomes a subsort or a second sort, then the situation is different in this respect. And also, if all the processes \( f(a) \) happen to be constants or closed terms and \( f \) can be defined in a finite number of equations, then the axiomatic introduction of \( P_f \) together with properties 1-5 is no problem. This applies among others to \( \varepsilon, \delta \) and atom to atom renaming.

Property 6 is slightly but significantly different from the corresponding axiom in [V], 1.11.

**Definition** Let \( x \in G \) and \( f: A \rightarrow G \). Then \( P_f(x) \in G \) is defined as the graph \( x \) in which each atomic edge \( e \) with label \( a \) has been replaced by a copy of \( f(a) \). The starting node of \( e \) is identified with the root of \( f(a) \), the endnodes of \( f(a) \) are identified (if they exist), the subgraph of the arrival node of \( e \) is then appended to this single end node. The subgraph is left disconnected if \( f(a) \) has no end nodes.

**Remark**

In 4.5.5 it will be shown that this definition of \( P_f: G \rightarrow G \) can be lifted to an operator \( G/\varepsilon \rightarrow G/\varepsilon \), the one that we are actually interested in. As usual, we will be using the same notation for both operators.

4.5.1. In order to make the above definitions (4.4.2 - 4.4.9) carry over to \( G/\varepsilon \) it must be proven that all these operators respect \( \varepsilon \)-bisimulation. We skip the cases of sum and product which are easy and can be looked up in [KV].

4.5.2. The *merge operator* defined in 4.4.3 respects \( \varepsilon \)-bisimulation.

**Proof:** Let \( f_1, f_2, g_1, g_2 \in G/\varepsilon \) be given such that \( f_1 \vDash g_1 \). We have to show that

\[ f_1 \vDash f_2 \vDash g_1 \vDash g_2. \]

Let \( R_i \) be the label preserving relation between \( f_i \) and \( g_i \). We define a relation \( S \) between \( f_1 \vDash f_2 \) and \( g_1 \vDash g_2 \) as follows:

\[ (m_1, m_2) S (n_1, n_2) \iff \{ (m_1 R_1 n_1) \& (m_2 R_2 n_2) \} \]

The roots are \( S \)-related. Suppose \( (m_1, m_2) S (n_1, n_2) \) and that \( p \) is a path \( \varepsilon^k a \) from \( (m_1, m_2) \) to \( (m'_1, m'_2) \) of which the last step is not a diagonal. Then this must be a path entirely in one of the operands, say \( f_1 \). Otherwise the \( a \)-edge would be orthogonal to a copy of one of the \( \varepsilon \)-steps earlier in the path. We therefore have a path \( \varepsilon^k a \) in \( g_1 \) from \( n_1 \) to \( n'_1 \). As \( m_2 \) and \( n_2 \) are \( R_2 \)-related, there can be no \( \varepsilon \)-edges arriving in \( (m_1, m_2) \) or \( (n_1, n_2) \) that stem from \( f_2 \) or \( g_2 \) respectively (see 4.3.4, statement a in the proposition and remember that nodes of graphs in \( G/\varepsilon \) never have both incoming \( \varepsilon \)-edges and atomic edges). This makes that the path \( \varepsilon^k a \) in \( g_1 \vDash g_2 \) from \( (n_1, n_2) \) to \( (n'_1, n'_2) \) survives and that \( (m'_1, m'_2) = (m_1, m_2) \) is \( S \)-related to \( (n'_1, n'_2) \).

If \( p \) is a path of the form \( \varepsilon^k (a \mid b) \) from \( (m_1, m_2) \) to \( (m'_1, m'_2) \) then there are paths \( \varepsilon^i a \) from \( m_1 \) to \( m'_1 \) in \( f_1 \) and \( \varepsilon^i b \) from \( m_2 \) to \( m'_2 \) in \( f_2 \). This gives paths \( \varepsilon^i a \) in \( g_1 \) from \( n_1 \) to \( n'_1 \) and \( \varepsilon^i b \) in \( g_2 \) from \( n_2 \) to \( n'_2 \). The starting node of the communication \( a \mid b \) in \( g_1 \vDash g_2 \) is therefore accessible from \( (n_1, n_2) \) by \( \varepsilon \)-steps. And the arrival nodes \( (m'_1, m'_2) \) and \( (n'_1, n'_2) \) are clearly \( S \)-related.

Suppose now \( (m_1, m_2) S (n_1, n_2) \) and that \( p \) is an \( \varepsilon \)-path to the end node \( (m'_1, m'_2) \). Then the projections of \( p \) on \( f_1 \) and \( f_2 \) are \( \varepsilon \)-paths to the end nodes \( m'_1 \) and \( m'_2 \). They correspond to \( \varepsilon \)-paths from \( n_1 \) and \( n_2 \) in \( g_1 \) and \( g_2 \). These two paths can be composed into an \( \varepsilon \)-path in \( g_1 \vDash g_2 \) which arrives in an end node. Do remember that the \( \varepsilon \)-structure in \( f \times g, f \vDash g \), and \( f \vDash g \) for two graphs \( f, g \in G/\varepsilon \) are equal.

The other direction (conditions 4, 5 in 4.3.1) follows from symmetry. \( \square \)
### 4.5.3. The left merge operator defined in 4.4.4 respects ε-bisimulation.

Proof: Let \( f_1, f_2, g_1, g_2 \in G \parallel \) be given such that \( f_1 \leftrightarrow \varepsilon g_1 \). We have to show that

\[
f_1 \parallel f_2 \leftrightarrow \varepsilon g_1 \parallel g_2.
\]

The relation \( S \) between these two graphs is defined in the same way as in 4.5.2. We only consider the differences with the merge case. If \((m_1, m_2)\) and \((n_1, n_2)\) are both not accessible from the root in \( \varepsilon \)-steps, then their subgraphs are the same as in \( f_1 \parallel f_2, g_1 \parallel g_2 \) respectively. Here condition 3 in 4.2.3 is applied. Suppose \((m_1, m_2) \, S \, (n_1, n_2)\) and one of \((m_1, m_2), (n_1, n_2)\) is \( \varepsilon \)-accessible from the root of its graph, then \((m_1, m_2)\) and \((n_1, n_2)\) are both the root of their graphs: \( \varepsilon \)-accessibility in the cartesian product graph means that the components \( m_1, m_2 \) (or \( n_1, n_2 \)) are both \( \varepsilon \)-accessible from the root in their own graph. Now apply 4.3.4.

Suppose \( p \) is a path in \( f_1 \parallel f_2 \) from the root, of the form \( \varepsilon^k a \), then this must be a path in \( f_1 \); it cannot end with a diagonal step, it must be in one of the two operands \( f_1, f_2 \) and it cannot be in \( f_2 \). This gives a corresponding path in \( g_1 \), etc...

The pure \( \varepsilon \)-paths are the same as in the merge graphs \( f_1 \parallel f_2 \) and \( g_1 \parallel g_2 \). \( \square \)

### 4.5.4. The communication merge operator defined in 4.4.4 respects ε-bisimulation.

Proof: Easy after reading 4.5.2 and 4.5.3.

### 4.5.5. In order to show that the definition in 4.4.9 of \( \rho_f \) carries over to \( G/\varepsilon \rightarrow G \) and that the function \( f \) may be considered as a function \( A \rightarrow G/\varepsilon \), we have to prove two statements:

1. Let \( f \) be a function \( A \rightarrow G \) and \( x, y \in G \). Then

\[
x \leftrightarrow \varepsilon y \Rightarrow \rho_f(x) \leftrightarrow \varepsilon \rho_f(y).
\]

2. Let \( f \) and \( g \) be two functions \( A \rightarrow G \) and \( x \in G \). Then

\[
\forall a \in A: f(a) \leftrightarrow \varepsilon g(a) \Rightarrow \rho_f(x) \leftrightarrow \varepsilon \rho_g(x).
\]

Proof of 1: A lot of new nodes come into existence in the construction of \( \rho_f(x) \) which are not yet present in \( x \). We have to define a notation for them. First we choose, for the processes \( f(a) \), representatives in \( G \) that have at most one end node (see 4.3.3.2). By \( (m, -) \) we denote a node of \( \rho_f(x) \) which stems from the node \( m \) of \( x \). By \( (k, e) \), where \( e \) is an edge of \( x \) which was replaced by a copy of \( f(lb(e)) \) and \( k \) is node of \( f(lb(e)) \), one of the new nodes is denoted. Some nodes have two names in this notation.

We call \( R \) the relation between \( x \) and \( y \). A relation \( S \) between \( \rho_f(x) \) and \( \rho_f(y) \) is defined as the smallest relation such that:

- \( \forall m \in N_x, n \in N_y : m R n \Rightarrow (m, -) S (n, -) \)

and

53
The Algebra of Communicating Processes with Empty Process

- \forall e_1 \in E_x, e_2 \in E_y:

\[ \text{lb}(e_1) = \text{lb}(e_2) \in A \land \text{sn}(e_1) \sqsubseteq \text{sn}(e_2) \land \text{an}(e_1) \sqsubseteq \text{an}(e_2) \quad \Rightarrow \\
\forall k \in N( f(\text{lb}(e_1))) : (k, e_1) S (k, e_2) \]

Showing that S defines an \( \varepsilon \)-bisimulation is now straightforward, although not trivial.

Proof of 2: Here we have to define a relation S between \( \rho_f(x) \) and \( \rho_g(x) \), the relation between \( f(a) \) and \( g(a) \) being denoted by \( R_{q} \). S is defined as the smallest relation such that:

- \((m, -) S (m, -) \)

and

- \( \forall e \in E_x, k_1 \in N_f(\text{lb}(e)), k_2 \in N_g(\text{lb}(e)) : \\
k_1 R_{\text{lb}(e)} k_2 \Rightarrow (k_1, e) S (k_2, e) \)

Again it is now straightforward to finish the proof. 0

4.6 What remains to be done in order to show that \( G/\leftrightarrow \) is a model of ACP\( e \) is a proof of the axioms and (doing a bit too much) of the properties of the \( p_f \) operator. Some of these proofs are trivial and/or can be looked up in [KV]. We will treat some that are less trivial.

4.6.1. The axiom \( x \parallel y = x \parallel y \parallel x + x \parallel y \) holds in \( G/\leftrightarrow \).

Proof: First choose representatives for \( x, y \) and \( z \) in \( G_{\parallel} \). The graphs on the right-hand side are all copies of \( x \parallel y \) with some edges, \( \varepsilon \)-accessible from the root, changed into \( \varepsilon \)-edges. Denoting nodes of \( x \parallel y \) by \( m_1, m_2 \) and \( m_3 \) respectively, where \( m \) is a node of \( x \parallel y \), we can define the relation \( S \) between \( x \parallel y \) and \( x \parallel y \parallel x + x \parallel y \) as the smallest relation such that:

\[ m \text{ has an atomic incoming edge or } m \text{ is the root of } x \parallel y \quad \Rightarrow \\
m S m_i \text{ for } i = 1, 2, 3. \]

If we denote the root of \( x \parallel y \) by \( r \), then in \( x \parallel y + y \parallel x + x \parallel y \parallel x_1, r_2 \) and \( r_3 \) are identified. It is easy to show that \( S \) is an \( \varepsilon \)-bisimulation. 0

4.6.2. The axiom \( a x \parallel y = a(x \parallel y) \) holds in \( G/\leftrightarrow \). It is a special case of the equation \( (x \parallel y) \parallel z = x \parallel (y \parallel z) \), at least after proving \( a x = a \parallel x \). This last equation is trivial using remark 6 in section 4.4.6. The former equation is one of the two equations in the proposition in 3.2.3 and also holds in \( G/\leftrightarrow \). For the proof we refer to 4.6.7.

4.6.3. The axiom \( x \parallel y = y \parallel x \) and the equation \( x \parallel y = y \parallel x \) hold in \( G/\leftrightarrow \).

Proof: Graph isomorphism, immediate from the definitions. 0

54
4.6.4. The merge operator is associative in $G/\mathcal{E}$.

This is not an axiom of $\text{ACP}^e$ of course but this statement is sufficiently important to be fatal for the graph model that we constructed in this section if it were not true. Below we will give a handwaving proof. Refer to 4.6.7 for a more formal proof.

Proof: We will show first that for graphs $x, y, z \in G/\mathcal{E}$, the graphs $x \parallel_y (y \parallel_z z)$ and $(x \parallel_y y) \parallel_z z$ are graph-isomorphic. The shell of nodes and edges, disregarding the labels, is the same for these two graphs as the cartesian product on graphs is associative. The $\varepsilon$-structure is also the same; the definition of $\parallel_y$ does not add or remove $\varepsilon$-edges. We only need to consider which atomic edges are changed into $\delta$-edges.

Edges are denoted by $(m, n, e_3), (m, e_2, p), (q, n, p)$ where $m, n, p$ are nodes in $x, y, z$ respectively and $e_1, e_2, e_3$ are edges of $x, y, z$ respectively. Suppose $(m, n, e_3)$ is an $\alpha$-edge in $(x \parallel_y y) \parallel_z z$. Then $m$ nor $n$ have incoming $\varepsilon$-edges. Therefore $(n, e_3)$ survives in $y \parallel z$ and $(m, n, e_3)$ survives in $x \parallel_y (y \parallel_z z)$. All the other cases are perfectly analogous.

Now we have to add the diagonals. Here too, we will just consider one out of a number of analogous cases. Suppose $(m, n, p)$ is a node in $x \parallel (y \parallel z)$ and vertex of a cube of atomic edges $a_x$, $b_y$, $c_z$ (looking in the cartesian product, in the free merge they may have become $\delta$). And suppose that $x$ and $y$ don't have arriving $\varepsilon$-edges in $m, n$ respectively and that $z$ does have in $p$. Then one can easily check that in both $(x \parallel y) \parallel z$ and $x \parallel (y \parallel z)$ we find the communication edges $a_x \parallel b_y \parallel c_z$, $a_x \parallel c_z$ and $b_y \parallel c_z$. $\Box$

4.6.5. The axiom $x \parallel (y \parallel z) = (x \parallel y) \parallel z$ holds in $G/\mathcal{E}$.

For a proof we refer to 4.6.7.

4.6.6. The axiom $a \parallel y = (a \parallel y) \parallel x$ is a special case of the second equation in the proposition in 3.2.3: $(x \parallel y) \parallel z = x \parallel (y \parallel z)$ (fill in $y := a, x := y, z := x$). The proof of this equation is analogous to the proof of the equation mentioned in 4.6.2: $(x \parallel y) \parallel z = x \parallel (y \parallel z)$, see 4.6.7.

4.6.7 The proof method introduced in 4.4.7 using the definitions in 4.4.8 will be applied here to the easiest case, the equation $(x \parallel y) \parallel z = x \parallel (y \parallel z)$. Some hints will be given for the other equations mentioned in 4.4.7 but the extensive mass of propositional logical calculation will be left to the reader.

Proof: First we determine the label function $lb_1$ (the label function for the lefthandside expression, not to be confused with $lb_1$) for the graph $x \parallel y \parallel z$ such that $(x \parallel y \parallel z, lb_1)$ represents the graph $(x \parallel y) \parallel z$ for $x, y, z \in G/\mathcal{E}$. The definition in 4.4.8 is such that $lb_1$ can be derived in a perfectly mechanical and reliable way. $lb_1$ equals the function $lb_1$ for the merge graph of $x \parallel y$ and $z$. Below only the conditions in which $lb_1$ equals $\varepsilon$ or $a$ are mentioned. In all other cases it equals $\delta$. Skipping some intermediate steps, the result looks like:

- $lb_1((n_x, n_y), e_z) = \varepsilon$ if $lb(e_z) = \varepsilon$
- $lb_1((n_x, n_y), e_z) = a$ if $lb(e_z) = a \& \neg \text{ine}(n_x) \& \neg \text{ine}(n_y)$

- $lb_1((n_x, e_y), n_z) = \varepsilon$ if $lb(e_y) = \varepsilon$
- $lb_1((n_x, e_y), n_z) = a$ if $lb(e_y) = a \& \neg \text{ine}(n_x) \& \neg \text{ine}(n_z)$

- $lb_1((e_x, n_y), n_z) = \varepsilon$ if $lb(e_x) = \varepsilon$
- $lb_1((e_x, n_y), n_z) = a$ if $lb(e_x) = a \& \neg \text{ine}(n_y) \& \neg \text{ine}(n_z)$

- $lb_1((n_x, e_y), e_z) = a$ if $\chi(e_y, e_z) = a \& \neg \text{ine}(n_x)$

55
The Algebra of Communicating Processes with Empty Process

\[
\text{lb}( (e_x, n_y), e_z) = a \iff \gamma( e_x, e_z) = a \land \neg \text{inE}( n_y)
\]

\[
\text{lb}( (e_x, e_y), n_z) = a \iff \gamma( e_x, e_y) = a \land \neg \text{inE}( n_z)
\]

\[
\text{lb}( (e_x, e_y), e_z) = \gamma( e_x, e_y, e_z)
\]

The label function \(\text{lb}_r\) for \(x \parallel (y \parallel z)\) in order to represent \(x \parallel (y \parallel z)\) looks like:

\[
\text{lb}_r( n_x, (n_y, e_z)) = \varepsilon \iff \text{lb}( e_z) = \varepsilon
\]
\[
= a \iff \text{lb}( e_z) = a \land \neg \text{inE}( n_x) \land \neg \text{inE}( n_y)
\]

\[
\text{lb}_r( n_x, (e_y, n_z)) = \varepsilon \iff \text{lb}( e_y) = \varepsilon
\]
\[
= a \iff \text{lb}( e_y) = a \land \neg \text{inE}( n_x) \land \neg \text{inE}( n_z)
\]

\[
\text{lb}_r( e_x, (n_y, n_z)) = \varepsilon \iff \text{lb}( e_x) = \varepsilon
\]
\[
= a \iff \text{lb}( e_x) = a \land \neg \text{inE}( n_y) \land \neg \text{inE}( n_z)
\]

\[
\text{lb}_r( n_x, (e_y, e_z)) = a \iff \gamma( e_y, e_z) = a \land \neg \text{inE}( n_x)
\]

\[
\text{lb}_r( e_x, (n_y, e_z)) = a \iff \gamma( e_x, e_z) = a \land \neg \text{inE}( n_y)
\]

\[
\text{lb}_r( e_x, (e_y, n_z)) = a \iff \gamma( e_x, e_y) = a \land \neg \text{inE}( n_z)
\]

\[
\text{lb}_r( e_x, (e_y, e_z)) = \gamma( e_x, e_y, e_z)
\]

One can see that \(\text{lb}\) and \(\text{lb}_r\) are exactly the same and that the method has, at least in this case, succeeded in keeping the formal story within reasonable proportions. One of the reasons why this is the simplest case is because graph isomorphism applies here, which does not hold for the three other equations. The label functions involved there sometimes differ but only for edges not accessible from the root. Probably the easiest way to deal with this situation consists of changing the label function such that they yield \(\delta\) for edges not accessible from the root. But still the proofs are much more involved than the one given above and are left to the reader. □

4.6.8. The properties of the \(\rho\) operator defined in 4.4.9 hold in \(G/\epsilon\). The axioms for \(\varepsilon\) and \(\delta\) in \(\text{ACP}_e\) follow trivially from these properties.

Proof: It is easy to show that for all 6 properties, graph isomorphism applies. □

4.6.9. In the preceding sections we have proven:

Theorem \((G/\epsilon, A, \varepsilon, \delta, +, \cdot, \parallel, \parallel, \parallel, \varepsilon, \delta)\) is a model of \(\text{ACP}_e\).

5. The supplementary axioms in the graph model.

5.1 In this section the supplementary axioms and operators of section 3 will be studied in the graph model \(G/\epsilon\). It will be shown that all these operators can be defined in this model and that all the axioms mentioned in section 3 hold in \(G/\epsilon\).
5.1.1. Proposition  In $G/e\leq\rhd$ the equations of 3.2.1 hold:

1. $x \parallel e = x$
2. $x \parallel e = x$
3. $e \parallel x = e$ or $e \parallel x = \delta$
4. $e \parallel (x \parallel (y + z)) = e \parallel (x \parallel y) + e \parallel (x \parallel z)$
5. $e \parallel x \delta = \delta$
6. $e \parallel (x \parallel y) = \delta$
7. $e \parallel (x \parallel y) = \delta$
8. $e \parallel (x \parallel y) = e \parallel (x \parallel y)\delta$
9. $e \parallel (x \parallel y) = e \parallel (x \parallel y)\delta$
10. $(e \parallel x) \parallel y = e \parallel (e \parallel x \parallel y)\delta$

Proof: Most are trivial. If $x \parallel y$ contains an $e$-path from the root to an end node, then this path can be projected onto $x$ and $y$. Which therefore also have $e$-paths to end nodes. □

5.1.2. Proposition In $G/e\leq\rhd$ the equations of 3.2.2 hold:

1. $x \delta \parallel y = (x \parallel y)\delta$
2. $y \parallel x \delta = (y \parallel x)\delta$
3. $x \delta \parallel y = (x \parallel y)\delta$
4. $x \parallel y \delta = (x \parallel y)\delta$

Proof: A simple $e$-bisimulation argument. □

5.1.3. The proposition and theorem of 3.2.3 were dealt with in section 4.6.7.

5.2. The handshaking axiom $a \parallel b \parallel c = \delta$ is rather a property of the communication function $\gamma$ than a property of a model. The interesting statement to prove in $G/e\leq\rhd$ is the following implication which was proven to hold in CTM in 3.3:

$$( \forall a, b, c \in A \quad a \parallel b \parallel c = \delta ) \Rightarrow ( \forall x, y, z \text{ processes } \quad x \parallel y \parallel z = \delta )$$

Proof: According to the definition of $x \parallel y$ in 4.4.5, it starts with a communication step (apart from intermediate $e$-steps and the possibility of immediate deadlock). Therefore $x \parallel y \parallel z$ will start with a triple communication step which means deadlock in this case. □

An important consequence of handshaking is the expansion theorem (see 3.3):

$$\sum_{i=1}^{n} x_i \parallel ( \sum_{j=1}^{n} x_j ) + \sum_{1 \leq i < j \leq n} ( x_i \parallel x_j ) \parallel ( \sum_{k=1}^{n} x_k )$$

Its proof is the same as the one suggested in 3.3 as the ingredients of that proof have already been shown to hold in $G/e\leq\rhd$.

5.3. The Projection Operators $\pi_n$.

5.3.1. For the definition of the projection operators $\pi_n$ in $G/e\leq\rhd$, we will only consider graphs that are trees without exclusive, infinite $e$-paths (paths that contain only $e$-steps and never arrive at an end node). Why this is done is explained in the remarks in 5.3.3.

Definition Let $f \in G$ be a tree without exclusive infinite $e$-paths and with all $\delta$-edges arriving in end nodes. The set of such trees is called $T_\pi$. The graph $\pi_n( f )$ is obtained from the graph of $f$ by
removing all nodes that cannot be accessed from the root by means of a path of at most n atomic or $\delta$-steps. Together with the nodes, their incoming edges are also removed.

It is easy to prove that each $T_\pi$ contains an element in every $\varepsilon$-bisimulation class of $G$: each element of $G$ can be unfolded into a tree (cf 4.3.3.3). In this tree, every $\varepsilon$-edge from which only infinite $\varepsilon$-paths can be accessed, are replaced by a $\delta$-edge. Then, every edge which is only accessible via a $\delta$-step, is removed. Finally, nodes and edges that are no longer connected to the root, are removed.

5.3.2. The operators $\pi_n$ respect $\varepsilon$-bisimulation.

Proof: We have to define a special bisimulation relation $S$ between two bisimilar trees $f, g \in T_\pi$. $S$ is defined as the smallest relation which relates all nodes $m$ and $n$ that have bisimilar subtrees, and are accessible from the root by paths with identical $A$-label and no $\delta$-steps (the $A$-label of a path is defined in the obvious way). It is easy to show that $S$ is indeed an $\varepsilon$-bisimulation and that its restriction to $\pi_n(f)$ and $\pi_n(g)$ is also an $\varepsilon$-bisimulation.

5.3.3. Remarks.

1. The $\pi_2$-images of these two graphs are not $\varepsilon$-bisimilar. That is why the definition of $\pi_n$ is restricted to trees.

2. The $\pi_0$-image of this graph seems to be the whole graph (which denotes the process $\delta$), contrary to the axiom $\pi_0(x) = \varepsilon$. That is why such $\varepsilon$-paths are excluded in the definition of $\pi_n$. They can always be replaced by a $\delta$-edge (cf 4.3.2.e).

5.3.4. The axioms for $\pi_n$ hold in $G_{\varepsilon \leftrightarrow \varepsilon}$:

\begin{align*}
\pi_0(x) &= \varepsilon \\
\pi_n(\varepsilon) &= \varepsilon \\
\pi_{n+1}(\delta) &= \delta \\
\pi_{n+1}(a \cdot x) &= a \cdot \pi_n(x) \\
\pi_n(x + y) &= \pi_n(x) + \pi_n(y).
\end{align*}

Proof: In each case, both sides of the equation yield the same graph. \qed
5.4. The Priority Operators $\triangleright$ and $\trianglerightcirc$.

5.4.1. Given a strict partial order $<$ on the atoms, one can define in $G$ the *unless* operator $\triangleright$ and the priority operator $\trianglerightcirc$ which model priority branching in a process (see 3.5 and [BBK2]). In this case the graphs have to be restricted to trees (for existence of a tree in each $\varepsilon$-bisimulation class, cf 4.3.3.3).

**Definition** Let $f, g \in T$. Define $E(f, n)$, where $n$ is a node of $f$, as the set of atomic edges, accessible from $n$. We say that an edge $e_1$ is majorized by an edge $e_2$ if $lb(e_1) < lb(e_2)$. The graph $f \triangleright g$ is obtained from the graph $f$ by making all the edges in $E(f, rf)$ majorized by an edge in $E(g, rg)$. The graph $\trianglerightcirc(f)$ is obtained from the graph $f$ by changing into, for each node $n$ of $f$, the edges in $E(f, n)$ that are majorized by another edge in this set.

5.4.2. $\triangleright$ and $\trianglerightcirc$ respect $\varepsilon$-bisimulation.

Proof: Easy using the relation defined in 5.3.2. $\square$

5.4.3. The axioms for $\triangleright$ and $\trianglerightcirc$ hold in $G/\varepsilon$:

$$
\begin{align*}
    a \triangleright b &= a & \text{if not } a < b \\
    a \triangleright b &= \delta & \text{if } a < b \\
    x \triangleright \varepsilon &= x \\
    x \triangleright \delta &= x \\
    x \triangleright a y &= x \triangleright a \\
    x \triangleright (y + z) &= (x \triangleright y) \triangleright z \\
    \varepsilon \triangleright x &= \varepsilon \\
    \delta \triangleright x &= \delta \\
    a x \triangleright y &= (a \triangleright y) x \\
    (x + y) \triangleright z &= x \triangleright z + y \triangleright z
\end{align*}
$$

Proof: Easy. $\square$

5.4.4. Remarks.

- The following example shows that in the definition of $\trianglerightcirc$ given above, the graphs have to be restricted to trees. The $\trianglerightcirc$-value of the graph below (an element of $G ||$) following this definition, is different from the value after unfolding.

![Diagram](fig. 3)
• In an early version of the priority axiom system, the axiom \( \delta(x + y) = \delta(x \triangleleft y) + \delta(y \triangleleft x) \) was used instead of \( \delta(x + y) = \delta(x \triangleleft y) + \delta(y) \triangleleft x \). The replacement was an improvement regarding the term rewriting properties of the system. In \( G/\equiv \) however, the equation \( \delta(x \triangleleft y) = \delta(x) \triangleleft y \) holds.

5.5. We can be short about EAR, the Epsilon Abstraction Rule in the model \( G/\equiv \). It was enounced in 3.6 and its proof is given in 4.3.3 remark 5.

5.6.1. The closed term model or initial algebra of \( \text{ACP}^c \) (called CTM in this paper), is isomorphic to a submodel of the graph model \( G/\equiv \), namely the model based on \( T_f \), the subset of finite trees. We will not prove this in detail as it involves techniques (rewriting analysis of the equations) not treated in this paper (one direction at least, the other direction is trivial).

5.6.2. \( (T_f/\equiv, A, \varepsilon, \delta, +, \cdot, \parallel, \perp, 1, \varepsilon_f, \delta_f) \) is a model of \( \text{ACP}^c \) and a submodel of \( G/\equiv \).

Proof: Starting from the canonical injection \( i : T_f \to G \), it is not hard to supply the necessary definitions and verify the necessary statements.

5.6.3. The translation of closed \( \text{BPA}^c \) terms into trees is an easy inductive process: the constants are translated into single edge graphs and \( x + y \), \( x \cdot y \) are translated as shown in 4.4.2. The translation of closed \( \text{ACP}^c \) terms can be done in two ways: eliminate the operators \( \parallel, \perp, 1, \varepsilon_f, \) and \( \delta_f \) and apply the algorithm for closed \( \text{BPA}^c \) terms. Or, alternatively, one can use the definitions in 4.4 followed by unwinding, realizing that this will always yield a finite tree.

The translation of finite trees into closed terms is an analogous inductive process.

5.6.4. Finite trees have an advantage: it is easy to define normal forms for them.

**Definition** An \textit{a-component} \( (a \in A) \) is a tree of the form

\[
\begin{array}{c}
\bullet \\
\downarrow \\
\ \ \\
\ \ \\
x
\end{array}
\]

where \( x \) is a finite tree.

So, each finite tree \( \neq \varepsilon, \delta \) is a sum of a finite number of a-components and possibly an \( \varepsilon \)-edge.

A finite tree is in \textit{normal form} if

• no node has two or more identical a-components;
• the tree does not contain any intermediate \( \varepsilon \)-edges or terminal \( \varepsilon \)-edges starting from a node with outdegree \( 1 \);
• the tree does not contain any terminal \( \delta \)-edges starting from a node with outdegree \( \geq 2 \);
• the tree does not contain any edges, accessible via a \( \delta \)-step.

An easy induction on the depth of a tree shows that two bisimilar trees in normal form must be identical (use the fact that a subtree of a tree in normal form is also in normal form).
The process of alternatingly removing superfluous a-components and removing superfluous ε- and δ-edges and inaccessible parts of the tree, must terminate when applied to a finite tree and must result in its normal form. This shows that ε-bisimilarity between two given finite trees can be reduced to tree-equality and is therefore decidable.

5.6.5. Once the isomorphism between the initial algebra of ACPε and the model T/ε,δ is proven the following two conclusions can be drawn:

- provable equality among closed ACPε terms is decidable.
- the operators r (3.7), πn (3.4), ♦ and δ (3.5) and their axioms are conservative extensions to ACPε.

5.6.6. The proofs and intuitive support concerning the deadlock behaviour of renaming operators, announced in section 3.7.2, can easily be supplied using finite trees. We first define what it means that a tree does or does not deadlock, then we show that these notions obey the same equations as those given for the predicates D and ND in section 3.7.2 and 3.7.3.

Definition DT, a predicate on T/ε,δ, is defined as:

\[ DT(t) \iff \text{the normal form of } t \text{ contains at least one } \delta\text{-edge.} \]

Proposition

1. \( \neg DT(\varepsilon), \neg DT(a), DT(\delta) \)
2. \( DT(x) \& DT(y) \Rightarrow DT(x + y) \)
3. \( \neg DT(x) \& \neg DT(y) \Rightarrow \neg DT(x + y) \)
4. \( DT(x y) \Leftrightarrow (DT(x) \text{ or } DT(y)) \)
5. \( DT(x) \Rightarrow DT(ax + y) \)
6. \( DT(x) \& \neg DT(x + y) \Rightarrow x = \delta \)
7. \( DT(x + y) \& \neg DT(y) \Rightarrow (DT(y) \Rightarrow DT(x + y)) \)
8. \( x \neq \varepsilon, \delta \Rightarrow (DT(y) \Rightarrow DT(x y + z)) \)

Proof: All of these are trivial to prove, using ε-bisimulation and the recipe for the normal form of a finite tree. □

Proposition For \( x \in \text{CTM} \), we have:

1. \( D(x) \Leftrightarrow DT(x) \)
2. \( ND(x) \Leftrightarrow \neg DT(x) \)

Here \( x \) denotes both an element of CTM and its image in \( T/ε,δ \) under the unique ACPε-isomorphism between these two models.

Proof: Easy. □

Finally the proofs can be supplied for the propositions in 3.7.2 and 3.7.3. As the recipe for normal forms never introduces any δ-steps, the proposition in 3.7.2 follows immediately. For 3.7.3 the easiest way is to apply structural induction and property 8 above.
The Algebra of Communicating Processes with Empty Process

References.


ALTERNATIVE IMPLEMENTATION STRATEGIES FOR PROCESS ALGEBRA TOOLS

Gert Veltink
DeTeWe Kommunikationssysteme
Zeughofstraße 1
10997 Berlin
Germany

Extended Abstract

In this talk we will try to analyze the needs and possibilities for a redesign or reimplementation of the PSF Toolkit. We will give an impression of what a redesign in the context of the ToolBus could look like and discuss a prototype of part of such an implementation.

PORTING THE PSF TOOLKIT

The current implementation of the PSF Toolkit (fig. 1) is strongly attached to the SUN hardware platform. Although it should be relatively easy to port the PSF Toolkit to other machine architectures running UNIX and the X-Window System, porting the PSF Toolkit in its current status to personal computer platforms such as Apple Macintoshes and IBM-compatible machines is expected not to be an easy task.

![Figure 1: The PSF Toolkit](image)
It is however, especially such a port that is interesting in order to reach a bigger audience of possible users. So in order to make the PSF Toolkit available for PC-platforms a redesign will probably be needed. This does not necessarily mean that all existing code should be thrown away, but that the existing software should be adapted and more importantly be organized in a different way. A second possible reason for a redesign is to make the PSF Toolkit fit for integration with the ToolBus concept.

EXPECTED DIFFICULTIES

One of the problems in porting the PSF Toolkit to other hardware platforms is that the code necessary to implement the graphical user interface, is mixed with the code implementing the main functionality of a tool in the current implementation. This has as result that the tools are largely depending on one specific windowing system, the X Window System in this case.

The problem with adapting the PSF Toolkit to a concept such as the ToolBus is the way information is exchanged between the different tools. In the PSF Toolkit this information interchange is performed using files in the TIL (Tool Interface Language) format, whereas the ToolBus relies on a more dynamical concept in which several tools running in parallel can exchange messages consisting of terms in a general format by means of the bus to which all tools are connected.

A POSSIBLE TOOLBUS IMPLEMENTATION

In order to achieve a successful redesign a thorough analysis of the different objects (data types) that play a role in an implementation of tools for process algebras as well as the operations that can be performed on these objects is needed. Next, when this analysis has been performed, the objects and operations should be grouped into independent parts each providing a set of services.

![Diagram of A possible ToolBus implementation](image)
Such service entities can then be transformed into tools in the sense of the ToolBus concept. This implies that such ToolBus tools have a much smaller granularity than the current tools in the PSF Toolkit.

Figure 2 depicts a possible set of lower-level entities for an implementation of software tools for process algebras based on the ToolBus concept, which will shortly be explained below.

**Parser**
The parser translates an input file from character strings into terms and sends these terms to the Object Storage.

**Object Storage**
All terms representing objects from a specification are stored in a central data base.

**Type Checker**
The type checker can be activated to check the (type) consistency of the objects stored in the Object Storage.

**User Interface**
All aspects of the graphical user interface are encapsulated in this service entity.

**Data Term Rewriter**
The Data Term Rewriter can be used to transform a data term into its normal form.

**Process Term Rewriter**
The Process Term Rewriter can be used to transform a process term into its (weak) head normal form.

**A PROTOTYPE IN CLEAN**
The final part of the talk will be dedicated to the description of a prototype implementation of the Process Term Rewriter in the functional programming language Clean. We will present the algorithm to calculate the (weak) head normal form of a process term and show how these results can be used as a basis for a simulator and a transition system generator.
A general conservative extension theorem in process algebras with inequalities

Pedro R. D'Argenio
LIFIA. Dept. of Comp. Sci.
Universidad Nacional de La Plata
e-mail: pedro@info.unlp.edu.ar

Abstract

A general conservative extension theorem for process algebras with inequalities is stated. General results for proving operational conservative extension up to a semantic preorder and equational conservative extension of equational specifications with inequalities are proposed. The proof of these facts reduces to check some simple conditions in the term deduction system of the process theory. A general theorem for proving completeness in extended process algebras with inequalities is given as a corollary.

1 Introduction

Process theories such as CCS, CSP and ACP have been extended with new features such as real-time and probabilistics. Hence, it is desirable that any property which has been proved in the old theory remains valid (for the old part) in the extended theory. That is, an extended theory should be conservative somehow with respect to the original one.

Conservativity in transition system specifications (or term deduction systems) was studied in [GV92], [Gro93], [BG91], [Ver94b] and [FV95]. In this setting, (operational) conservativity means that the provable transitions for an original term are the same both in the original and in the extended term deduction systems.

Verhoef proposed in [Ver94b] a general conservative extension theorem for equational specifications. Here, (equational) conservativity means that exactly the same identities between closed terms in the original framework can be proved both in the original and in the extended equational specifications. That theorem solves several complications when an equational specification is extended. For instance, it avoids to deal with term rewriting analysis which are frequently used to prove equational conservativity. These term rewriting systems often have no nice properties. Thus, according to [Ver94b], the problem of proving conservative extension of transition system based equational specifications (namely process algebra) can be reduced to check operational conservative extension of the associated term deduction system.

This article extends that work to deal with semantic preorders defined on transition systems and equational specifications with inequalities or inequational specifications, as they will be called in this article.

The proposed method involves three steps. The first one is to state conservative extension of the term deduction systems. In order to do this, an operational conservative extension
A theorem is given. It is a simple variation of that introduced in [FV95] that considers term deduction systems which have a unique well supported model (see also [Gla95]). The second step states operational conservative extension up to a preorder defined exclusively in terms of transition relations and predicates. This is proved to be an immediate consequence of the first step. The last step is to prove conservative extension of the inequational specifications. With this purpose, a general conservative extension theorem is introduced for inequational specifications that axiomatize that kind of preorders.

Thus, the proof of inequational conservative extension and operational conservative extension up to certain preorder reduces to check operational conservative extension of the term deduction system, which can be done by verifying some simple conditions.

The paper is organized as follows. Section 2 introduces preliminary concepts. The first paragraph briefly explains the SOS theory. The second paragraph introduces basic notions of the algebraic treatment of inequational specifications. Section 3 states the results of this paper. In the first paragraph, operational conservativity results are stated. The second paragraph deals with inequational conservativity and proves the general conservative extension theorem. Finally, some examples of application are given.

Acknowledgements. I am grateful to Chris Verhoef for encouraging me to make this paper, and also for his valuable comments and suggestions. I also thank Twan Basten and the anonymous referees for their useful suggestions.

2 Preliminaries

This section briefly recalls some notions about SOS theory and inequational specifications. The first paragraph explains SOS theory following [Ver94a], [Gla95] and [FV95], since they seem to have the most general treatment. The second paragraph gives some basic notions about the algebraic treatment of inequational specifications.

Some concepts of SOS

Assume an infinite set \( V \) of variables. A (single sorted) signature \( \Sigma \) is a set of functions symbols together with their arity. The notion of term (over \( \Sigma \)) is defined as expected: \( x \in V \) is a term; and, if \( t_1, \ldots, t_n \) are terms and if \( f \in \Sigma \) is \( n \)-ary then \( f(t_1, \ldots, t_n) \) is a term. A term is also called an open term and the set of open terms is denoted by \( O(\Sigma) \). A term containing no variables is called a closed term and the set of closed terms is denoted by \( C(\Sigma) \).

Let \( t \in O(\Sigma) \) then \( \text{var}(t) \subseteq V \) is the set of all variables occurring in \( t \).

A substitution is a function \( \sigma : V \to O(\Sigma) \). This map can easily be extended to the set of all terms by substituting for each variable occurring in an open term its \( \sigma \)-image.

**Definition 2.1 (Term deduction systems)** A term deduction system is a structure \((\Sigma, D)\) where \( \Sigma \) is a signature and \( D \) is a set of deduction rules. The set \( D = D(T_p, T_r) \) is parametrized with two sets which are called respectively the set of predicate symbols and the set of relation symbols. Let \( s, t, u \in O(\Sigma) \), \( P \in T_p \) and \( R \in T_r \). Expressions \( Ps, \neg Ps, tRu \) and \( t\neg R \) are called formulas. Formulas \( Ps \) and \( tRu \) are called positive and \( \neg Ps \) and \( t\neg R \) are called negative. Let \( F \) be a set of formulas. \( PF(F) \) denotes the subset of positive formulas of \( F \) and \( NF(F) \) denotes the subset of negative formulas of \( F \).
A deduction rule $d \in D$ has the form $\frac{H}{C}$; with $H$ a set of formulas, and $C$ a positive formula. Elements of $H$ are called the hypothesis of $d$, and $C$ is the conclusion of $d$. If the set of hypothesis of a deduction rule is empty, it is called an axiom, and it will be denoted only by its conclusion provided that no confusion arises. The notions of "substitution", "var" and "closed" extend to formulas and deduction rules as expected.

Note that arbitrary many premises are allowed in the set of hypotheses of a deduction rule.

Example 2.1 As a running example, the operational semantics of the process algebra PA [BK84, BW90] is presented. The signature contains constants $a$ of a set $A$ of atomic actions, and four binary operators: the alternative composition $(\oplus)$, the sequential composition $(\cdot)$, the parallel composition or merge ($\parallel$) and the left merge ($\ll$). It is easy to see that the above signature plus the deduction rules in Table 1 form a term deduction system. This term deduction system has relations $\xrightarrow{a}$ and predicates $\xrightarrow{a} \checkmark$ for all $a \in A$. The intended interpretation of $x \xrightarrow{a} y$ is that a process $x$ executes an action $a$ and then behaves like $y$. The intended meaning of $x \xrightarrow{a} \checkmark$ is that $x$ terminates successfully after the execution of $a$.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \xrightarrow{a} \checkmark$</td>
<td>$x \xrightarrow{a} x'$</td>
<td>$x \xrightarrow{a} \checkmark$</td>
</tr>
<tr>
<td>$x \xrightarrow{a} x' \cdot y$</td>
<td>$x \cdot y \xrightarrow{a} x' \cdot y$</td>
<td>$x \cdot y \xrightarrow{a} y$</td>
</tr>
<tr>
<td>$x + y \xrightarrow{a} x'$</td>
<td>$y + x \xrightarrow{a} x'$</td>
<td>$x \xrightarrow{a} \checkmark$</td>
</tr>
<tr>
<td>$y + x \xrightarrow{a} \checkmark$</td>
<td>$x + y \xrightarrow{a} \checkmark$</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Operational rules for PA

In addition, notice that the set of operations $\Sigma_{BPA} = A \cup \{\oplus, \cdot\}$ together with the deduction rules above the line in Table 1 form another term deduction system which will be called BPA [BW90]. Yet another term deduction system is formed by the signature $\Sigma_{MRG} = A \cup \{\oplus, \cdot, \parallel, \ll\}$ plus the deduction rules below the line in Table 1. It will be called MRG since it defines the operational semantics of the merge operators.

The following definition tells when a formula is provable from a term deduction system.

Definition 2.2 (Proof of a rule) Let $T = (\Sigma, D)$ be a term deduction system. A proof of a rule $\frac{H}{C}$ from $T$ is a well-founded, upwardly branching tree of which the nodes are labelled with formulas of $T$, such that:

1. the root is labelled with $C$, and
2. if $\phi$ is the label of a node $q$ and $F$ is the set of labels of the nodes directly above $q$, then

- either $F = \emptyset$ and $\phi \in H$,
- or $\frac{F}{\phi}$ is a substitution instance of a rule $d \in D$.

If a proof of $\frac{H}{\phi}$ from $T$ exists, then $\frac{H}{\phi}$ is provable from $T$, notation $T \vdash \frac{H}{\phi}$. ■

**Definition 2.3** Let $T$ be a term deduction system. Let $F(T)$ be the set of all closed formulas of $T$. Let $PF(T)$ be the set of all positive closed formulas over $T$. Let $X \subseteq PF(T)$ and $\phi \in F(T)$. Then $X \models \phi$ (read $\phi$ holds in $X$) is defined according to the form of $\phi$ by:

- $X \models sRt$ if $sRt \in X$,
- $X \models Ps$ if $Ps \in X$,
- $X \models s\neg R$ if $\forall t \in C(\Sigma) : sRt \notin X$,
- $X \models \neg Ps$ if $Ps \notin X$.

■

The purpose of a term deduction system is to define a set of positive formulas that can be deduced using the deduction rules. That is, one wants to talk about models for term deduction systems. Moreover, one would like to work with the most representative model. Meaning of transition system specifications was studied by Bol & Groote in [BG91] and more widely by van Glabbeek in [Gla95]. In this article, the definition of well supported model or stability is taken from [Gla95] although it was originally introduced in [BG91].

**Definition 2.4 (Well supported model)** Let $T = (\Sigma, D)$ be a term deduction system and let $X \subseteq PF(T)$ be a set of positive closed formulas. $X$ is a well supported model $T$ if

$$\phi \in X \iff \text{there exists a closed rule $\frac{H}{\phi}$ without positive hypotheses such that $T \vdash \frac{H}{\phi}$ and for all $h \in H$, $X \models h$.}$$

If $T$ has a unique well supported model, it is denoted by $S(T)$. ■

**Definition 2.5 (Source dependence)** [Gla93a, FV95] Let $d = \frac{H}{\phi}$ be a deduction rule where $C$ has the form $Pt$ or $tRt'$. The collection of source dependent variables $SV(d)$ is defined inductively as follows:

- $\text{var}(t) \subseteq SV(d)$; and
- if $sR's' \in H$ and $\text{var}(s) \subseteq SV(d)$, then $\text{var}(s') \subseteq SV(d)$.

$d$ is called source dependent if $SV(d) = \text{var}(d)$. A term deduction system is called source dependent if all of its rules are.

■

**Example 2.2** The operational rules of PA are in path format [BV93] and they are source dependent. As PA has only positive rules, it has a unique well supported model (see [Gla95]). ■
Many equivalences are definable in terms of relation and predicate symbols only. A complete survey of interleaving semantics equivalences was made by van Glabbeek in [Gla90] (for concrete processes) and [Gla93b] (for processes with abstraction). Equivalences for true concurrency were also defined in that way, for instance, step bisimulation [NP84, Pom86] and pomset bisimulation [BC88]. Besides, not only equivalences are defined in term of relation and predicate symbols but also preorders. Examples of preorders are: simulation, n-nested simulations [GV92], ready simulation [BIMSS], the preorder for the degree of parallelism based on pomset bisimulation of [Ace91], the "more distributed than" preorders of [Cas93] and [Yan93], the preorder for unstable nondeterminism of [VB95] and the preorder of bisimulation with divergence of [Abr87] and the ones of [Wal90].

Example 2.3 Simulation is defined for the PA terms as an example of preorders defined in terms of relation and predicate symbols.

A binary relation $S$ on the set of closed PA terms is a simulation if for all $(s, t) \in S$ and for all $a \in A$, the two following transfer properties hold:

- $\forall s', t': s \xrightarrow{a} s' \Rightarrow \exists t': t \xrightarrow{a} t' \land (s', t') \in S$,
- $s \xrightarrow{a} \sqrt{ } \Rightarrow t \xrightarrow{a} \sqrt{ }$.

If there is a simulation $S$ such that $(s, t) \in S$, then $s$ is simulated by $t$, notation $s \subseteq t$.

Some concepts of inequational specifications

Definition 2.6 (Inequational specifications) An inequational specification is a structure $(\Sigma, E)$ where $\Sigma$ is a signature and $E$ is a set of inequalities of the form $s \leq t$ where $s, t \in O(\Sigma)$. Sometimes, $E$ also contains rules or conditional inequalities $G \Rightarrow s \leq t$ where $G$ is a set of inequalities. $s = t$ is often written standing for $s \leq t$ and $t \leq s$. The notion of "substitution", "var" and "closed" extend to inequalities as expected.

An inequational specification is indeed an equational specification. I chose such a name to make clear that inequalities are explicitly managed and to relate the idea that their models will be based in preorders instead of equivalences.

Definition 2.7 (Derivability) Let $L = (\Sigma, E)$ be an inequational specification. Let $s, t \in O(\Sigma)$. An inequality $s \leq t$ can be derived from $E$, notation $E \vdash s \leq t$, according to the following definition

1. $s \leq t \in E$ implies $E \vdash s \leq t$;
2. for all substitutions $\sigma : V \rightarrow O(\Sigma)$, $E \vdash s \leq t$ implies $E \vdash \sigma(s) \leq \sigma(t)$;
3. let $\sigma : V \rightarrow O(\Sigma)$ and let $G \Rightarrow s \leq t \in E$, if for all $u \leq v \in G$, $E \vdash \sigma(u) \leq \sigma(v)$ then $E \vdash \sigma(s) \leq \sigma(t)$;
4. for all $f \in \Sigma$ with arity $n$, $E \vdash s_i \leq t_i$ for all $i \in \{1, \ldots, n\}$ imply $E \vdash f(s_1, \ldots, s_n) \leq f(t_1, \ldots, t_n)$;
5. $E \vdash t \leq t$;
6. $E \vdash s \leq t$ and $E \vdash t \leq u$ implies $E \vdash s \leq u$.

71
Notice that the rule for symmetry is not included, and so, the equational specifications one uses to manage are a particular case of inequational specifications. Actually, 1, 2 and 3 are enough to define derivability. 4, 5 and 6 may be rules (and axioms) in \( E \) and, moreover, they could not be present.

**Example 2.4** The signature of PA together with the axioms in Table 2 form an inequational specification. It will be called \( \text{PA} \leq \). In addition, two inequational specifications more are considered: the signature of BPA together with axioms in the left column form the inequational specification \( \text{BPA} \leq \), and the signature of MRG together with axioms in the right column form the inequational specification \( \text{MRG} \leq \).

<table>
<thead>
<tr>
<th>A1</th>
<th>( x + y = y + x )</th>
<th>M1</th>
<th>( x \parallel y = x \parallel y + y \parallel x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>( x + (y + z) = (x + y) + z )</td>
<td>M2</td>
<td>( a \parallel x = a \cdot x )</td>
</tr>
<tr>
<td>A3</td>
<td>( x + x = x )</td>
<td>M3</td>
<td>( a \cdot x \parallel y = a \cdot (x \parallel y) )</td>
</tr>
<tr>
<td>A4</td>
<td>( (x + y) \cdot z = x \cdot z + y \cdot z )</td>
<td>M4</td>
<td>( (x + y) \parallel z = x \parallel z + y \parallel z )</td>
</tr>
<tr>
<td>A5</td>
<td>( (x \cdot y) \cdot z = x \cdot (y \cdot z) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SM</td>
<td>( x \leq x + y )</td>
<td>MP</td>
<td>( x \cdot y \leq x \parallel y )</td>
</tr>
</tbody>
</table>

Table 2: Axioms of \( \text{PA} \leq \)

Axioms A1 to A5 and M1 to M4 are the well known axioms for the PA process algebra [BK84, BW90]. SM is the axiom of simulation; it is often used for theoretical reasons in process algebra theory (for instance it was used for proving completeness of BPA. See [BW90]). MP stands for “more parallel” and introduces the idea that \( x \parallel y \) has a “more parallel behaviour” than \( x \cdot y \). For closed terms, MP can be deduced by induction from the other axioms.

**Definition 2.8 (Algebras and axiomatizations)** An algebra is a set \( A \) of elements together with certain functions over \( A \) of arity \( n \geq 0 \).

Let \( \Sigma \) be a signature. A \( \Sigma \)-algebra \( A \) is an algebra within a function for each function symbol in \( \Sigma \) with the same arity. Such a correspondence is called an interpretation. The notion of “interpretation” extends to closed terms as expected and for \( t \in C(\Sigma) \), \([t] \) denotes the interpretation of \( t \) in \( A \). Interpretation extends also to open terms by universally quantifying the variables.

Let \( L = (\Sigma, E) \) be an inequational specification. Let \( A \) be a \( \Sigma \)-algebra with \( A \) being the set of elements. Let \( \leq \) be a preorder on \( A \) that preserves all functions in \( A \), i.e., \( \leq \) is a precongruence on \( A \). \( E \) is a sound inequality axiomatization with respect to \( \leq \) of \( A \) if for all \( s, t \in O(\Sigma) \),

\[ E \vdash s \leq t \implies [s] \leq [t]. \]

Moreover, if for all closed terms \( s, t \in C(\Sigma) \)

\[ E \vdash s \leq t \iff [s] \leq [t] \]

\( E \) is called a complete inequality axiomatization with respect to \( \leq \) for \( A \).
Example 2.5 It is not difficult to see that $\subseteq$ is a precongruence for PA. In order to prove that PA$\subseteq$ is a sound inequality axiomatization with respect to the $\subseteq$ model induced by the PA term deduction system, it is enough to prove that for every axiom $s \leq t$ of PA$\subseteq$ with free variables in $V$, the relation

$$S = \{(\sigma(s), \sigma(t)) | \sigma \text{ substitutes variables in } V \text{ to closed terms} \} \cup \text{Id}$$

is a simulation.

Moreover, BPA$\subseteq$ is a complete inequality axiomatization with respect to the $\subseteq$ model induced by the BPA term deduction system. The proof of this is quite similar to the proof of completeness of BPA with respect to bisimulation (see [BW90]). $\blacksquare$

3 The conservative extension theorems

This section is devoted to states several results of conservative extension. The general conservative extension theorem of [Ver94b] is extended to deal with preorders and inequational specifications.

Definition 3.1 Let $\Sigma_0$ and $\Sigma_1$ be two signatures. If for all $f \in \Sigma_0 \cap \Sigma_1$ the arity of $f$ in $\Sigma_0$ is the same as the arity of $f$ in $\Sigma_1$ then $\Sigma_0 \oplus \Sigma_1$, called the sum of $\Sigma_0$ and $\Sigma_1$, is defined as the signature $\Sigma_0 \cup \Sigma_1$. Note that $\emptyset$ is not simply the union of two signatures since sometimes it is not defined. $\blacksquare$

Definition 3.2 Let $T_i = (\Sigma_i, D_i)$ be term deduction systems with predicate and relation symbols in $T_i$ respectively ($i = 0, 1$). Let $\Sigma_0 \oplus \Sigma_1$ be defined and let $T_0 \cap T_1 = T_0^0 \cap T_1^1 = \emptyset$. Then $T_0 \oplus T_1$, called the sum of $T_0$ and $T_1$, is the term deduction system $(\Sigma_0 \oplus \Sigma_1, D_0 \cup D_1)$ with predicate and relation symbols $T_0^0 \cup T_1^0$ and $T_0^1 \cup T_1^1$. $\blacksquare$

Example 3.1 $\Sigma_{BPA} \oplus \Sigma_{MRG}$ is defined and equals to the signature of PA which is the same as $\Sigma_{MRG}$. Moreover, the term deduction system PA equals to BPA $\oplus$ MRG. $\blacksquare$

Operational conservativity

This paragraph defines the notions of operational conservative extension and operational conservative extension up to some preorder which is defined in terms of predicate and relation symbols.

The following definition is adapted from [BG91] and based on [GV95] which is given below with the name of weak operational conservative extension.

Definition 3.3 (Operational conservative extension) Let $T_i = (\Sigma_i, D_i)$ be term deduction systems with $T = (\Sigma, D) = T_0 \oplus T_1$ defined and let $D = D(T_p, T_r)$. The term deduction system $T$ is an operational conservative extension of $T_0$ if it has $S(T)$ as unique well supported model and $S(T_0) = \{Ps, sRt \in S(T) | s \in C(\Sigma_0), P \in T_p, R \in T_r \}$ is the unique well supported model of $T_0$. $\blacksquare$

Definition 3.4 (Weak operational conservative extension) Let $T_i = (\Sigma_i, D_i)$ be term deduction systems with $T = (\Sigma, D) = T_0 \oplus T_1$ defined and let $D = D(T_p, T_r)$. The term deduction system $T$ is a weak operational conservative extension of $T_0$ if for each well supported model $X$ of $T$ the set $\{Ps, sRt \in X | s \in C(\Sigma_0), P \in T_p, R \in T_r \}$ is a well supported model of $T_0$. $\blacksquare$
Weak operational conservative extension cannot be used in the context of this study since it does not consider unique model. The problem of multiple models is that a preorder defined in terms of predicate and relation symbols may relate different closed terms in each model and this fact can introduce inconsistency of axiomatizations.

The following definition and the next theorem originate in this article. They are the generalization for preorders of the case of operational conservativity up to an equivalence given by Verhoef [Ver94b].

**Definition 3.5 (Operational conservative extension up to a semantical preorder)**

Let $T_i = (\Sigma_i, D_i)$ be term deduction systems and let $T = (\Sigma, D) = T_0 \oplus T_1$ defined. Let $\xi$ be a semantic preorder defined in terms of predicate and relation symbols only. $T$ is an operational conservative extension of $T_0$ up to $\xi$ preorder if for all $s, t \in C(\Sigma_0)$, $s \preceq^\xi t$ $\iff$ $s \preceq^D t$. $
$
**Theorem 3.1 (Conservation of operational conservativity)**

Let $T_i = (\Sigma_i, D_i)$ be term deduction systems and let $T = (\Sigma, D) = T_0 \oplus T_1$ defined. If $T$ is an operational conservative extension of $T_0$, then it is also an operational conservative extension up to $\xi$ preorder, for any preorder $\xi$ defined in terms of predicate and relation symbols only.

**Proof.** (Sketch) Let $s, t \in C(\Sigma_0)$. Since $T$ is an operational conservative extension of $T_0$, the state-transition diagrams (or better: the term-relation-predicate diagrams) of $s$ in both $T$ and $T_0$ are the same, and so are the term-relation-predicate diagrams of $t$. Let $\xi$ be a preorder defined in terms of relation and predicate symbols. Because $\preceq^D$ is defined in the same way for relation and predicate symbols in $T_0$ as $\preceq^\xi$, and the term-relation-predicate diagrams of $s$ and $t$ are the same in both term deduction systems, $s \preceq^\xi t$ implies $s \preceq^D t$. The counterpositive is analogously proved.

Groote & Vaandrager [GV92] gave a first theorem for operational conservative extension in positive transition system specifications. Bol & Groote introduced in [BG91] a set of conditions that ensure conservativity in transition system specifications with negative premises. Verhoef did the same for stratifiables term deduction systems in [Ver94b]. The next conservative extension theorem was introduced by Fokkink & Verhoef in [FV95].

**Theorem 3.2 (Weak operational conservativity)**

Let $T_0 = (\Sigma_0, D_0)$ and $T_1 = (\Sigma_1, D_1)$ be two term deduction system satisfying:

1. $T_0$ is source dependent, and

2. if there is a conclusion $sR's'$ or $P s$ of a rule $d \in D_1$ with $s = x$ or $s = f(x_1, ..., x_n)$ for an $f \in \Sigma_0$, then, there is a hypotheses of $d$ which has the form $P't$ or $tR'u$ where $P' \not\in T_0^0$, $R' \not\in T_0^0$ or $u \not\in O(\Sigma_0)$, $t \in O(\Sigma_0)$ and $\text{var}(t) \subseteq SV(d)$.

If $T_0 \oplus T_1$ is defined then it is a weak operational conservative extension of $T_0$.

**Proof.** [FV95]

As a corollary of this theorem, yet another operational conservative extension theorem is introduced.
Theorem 3.3 (Operational conservativity) Let $T_0 = (\Sigma_0, D_0)$ and $T_1 = (\Sigma_1, D_1)$ be two term deduction systems satisfying statements 1. and 2. of Theorem 3.2 and, in addition,

3. $T_0$ has unique well supported model.

If $T_0 \oplus T_1$ is defined and it has unique well supported model, $T_0 \oplus T_1$ is an operational conservative extension of $T_0$.

Proof. Immediate from Definition 3.3 and Theorem 3.2

The theorem is somehow more general than the one of [BG91]. [BG91] requires that for every rule $\frac{H}{i_0} \in D_1$, $t$ is not in $O(\Sigma_0)$, and moreover, $T_0 \oplus T_1$ should be positive after reduction. Theorem 3.3 is more relaxed about the form of the new rules, and, in addition, a term deduction system may be not reducible to a positive one but may have a unique well supported model (see [Gla95]). However, the statement 3 cannot be omitted while no analogous one is required in [BG91]. I decided to pay this cost for two reasons: first, conditions 1 and 2 are quite more general than that proposed by [BG91], and second, in the context proposed by this article, one already knows whether condition 3 holds.

Example 3.2 It is easy to see that BPA and MRG satisfy the conditions of theorem 3.3. Thus PA is an operational conservative extension of BPA. Moreover, because of theorem 3.1, PA is an operational conservative extension up to simulation preorder.

Inequational conservativity

This last paragraph states the general conservative extension for inequational specifications.

Definition 3.6 Let $L_i = (\Sigma_i, E_i)$ be inequational specifications ($i = 0, 1$). Let $\Sigma_0 \oplus \Sigma_1$ be defined. $L_0 \oplus L_1$ is the sum of $L_0$ and $L_1$ defined as the inequational specification $(\Sigma_0 \oplus \Sigma_1, E_0 \cup E_1)$.

Example 3.3 Notice that BPA$\leq$ $\oplus$ MRG$\leq$ forms PA$\leq$.

Definition 3.7 (Inequational conservative extension) Let $L_i = (\Sigma_i, E_i)$ be inequational specifications ($i = 0, 1$) and let $L = (\Sigma, E) = L_0 \oplus L_1$ be defined. $L$ is an inequational conservative extension of $L_0$ if for all $s, t \in C(\Sigma_0)$

$$E \vdash s \leq t \iff E_0 \vdash s \leq t.$$

If for all $s \in C(\Sigma)$ there is a $t \in C(\Sigma_0)$ such that $E \vdash s = t$, then $L$ has the elimination property.

Theorem 3.4 (The general conservative extension theorem) Let $L_i = (\Sigma_i, E_i)$ be inequational specifications and let $L = (\Sigma, E) = L_0 \oplus L_1$ be defined. Let $T_i = (\Sigma_i, D_i)$ be term deduction systems and let $T = (\Sigma, D) = T_0 \oplus T_1$ be defined. Let $\xi$ be a preorder definable exclusively in terms of predicate and relation symbols. Let $E_0$ be a complete inequality axiomatization with respect to the $\xi$ preorder model induced by $T_0$ and let $E$ be a sound inequality axiomatization with respect to the $\xi$ preorder model induced by $T$. If $T$ is an operational conservative extension up to $\xi$ preorder of $T_0$, then $L$ is an inequational conservative extension of $L_0$.

Moreover, if $L$ has the elimination property, $E$ is a complete inequality axiomatization with respect to the $\xi$ preorder model induced by $T$. 

75
Proof. The proof that for all \( s, t \in C(\Sigma_0) \), \( E_0 \vdash s \leq t \implies E \vdash s \leq t \) is trivial. Now, let \( s, t \in C(\Sigma_0) \) and suppose \( E \vdash s \leq t \). Since \( E \) is sound, \( s \preceq_\xi t \). Because \( T \) is an operational conservative extension up to \( \xi \) preorder of \( T_0 \), \( s \preceq_0 t \). Finally, \( E_0 \vdash s \leq t \) since \( E \) is complete with respect to \( \xi \). So \( E \) is an inequational conservative extension of \( E_0 \).

Now, suppose moreover that \( L \) has the elimination property. Let \( s, t \in C(\Sigma) \) such that \( s \preceq_\xi t \). Then, there are \( s', t' \in C(\Sigma_0) \) such that \( E \vdash s = s' \) and \( E \vdash t = t' \). Since \( E \) is sound, \( s' \preceq_0 s \preceq_\xi t \preceq_0 t' \), where \( \preceq_0 \) stands for \( \preceq_\xi \cap \preceq_\theta \). So, it is enough to prove that \( E \vdash s' \leq t' \), but \( T \) is an operational conservative extension of \( T_0 \) up to \( \xi \) preorder, so \( s' \preceq_\xi t' \) and, because \( E_0 \) is complete, \( E_0 \vdash s' \leq t' \) which trivially implies \( E \vdash s' \leq t' \).

Example 3.4 Example 2.5 states that \( \text{BPA}^\leq \) is complete with respect to \( \leq \) for \( \text{BPA} \) and \( \text{PA}^\leq \) is sound with respect to \( \leq \) for \( \text{PA} \). Since, in addition, \( \text{PA} \) is an operational conservative extension up to \( \leq \) of \( \text{BPA} \), \( \text{PA}^\leq \) is an inequational conservative extension of \( \text{BPA}^\leq \). Moreover, because \( \text{PA}^\leq \) has the elimination property (see [BW90]), it is a complete inequality axiomatization with respect to \( \leq \) of \( \text{PA} \).

4 Further remarks

Applications

Voorhoeve & Basten introduced in [VB95] a preorder for unstable nondeterminism. They dealt with a set of autonomous actions which can be regarded as observable actions that somehow behaves as the silent step. Several algebras were defined there. \( \text{BPA}_{\leq}^{\text{aa}} \) is the basic process algebra with deadlock and autonomous actions. They used results in this article to extend \( \text{BPA}_{\leq}^{\text{aa}} \) with the parallel operator, obtaining thus \( \text{ACP}^\leq_{\text{aa}} \). Moreover, since \( \text{ACP}^\leq_{\text{aa}} \) has the elimination property, completeness was proved using results introduced below. In addition, they added the binary Kleene star [BBP94] to both theories. Since \( \text{BPA}^\leq_{\text{aa}} \) and \( \text{ACP}^\leq_{\text{aa}} \) are sound, and the respective term deduction systems satisfy the conditions of Theorem 3.3, operational and inequational conservative extension can be also proved. Figure 1 shows this overview. There, an arrow \( A \rightarrow B \) means that \( A \) is both an operational conservative extension and an inequational conservative extension of \( B \), and that it can be shown by using results in this article.

![Figure 1: Conservative extension in algebras for Voorhoeve & Basten's preorder](image)

Perhaps, the reader expected the arrow \( \text{BPA}^\leq_{\text{aa}} \rightarrow \text{ACP}^\leq_{\text{aa}} \). In this case only operational conservative extension can be proved using results in this articles (and so operational conservative extension up to the preorder). Since \( \text{BPA}^\leq_{\text{aa}} \) is not complete (see [Sew93, VB95]) Theorem 3.4 cannot be used.
Walker introduced in [Wal90] a complete (but non finite) axiomatization for a preorder that extends \( \tau \) bisimulation with divergence. Now, I sketch some proofs of conservativity and completeness using the results in this article. Let \( ST \) be the algebra of synchronization trees with Milner's \( \tau \) laws [Mil89]. The signature of \( ST \) has prefixing operators, the alternative composition and the nil process. Let CCS be the well known calculus of Milner [Mil89] that extends \( ST \) with renaming, restriction and parallel composition, and the expansion laws. Let \( ST \perp \) and \( CCS \perp \) the respective extensions of \( ST \) and CCS including the divergence operator with the laws for divergence given in [Wal90]. It is worthy to remark that for all CCS term Walker's preorder agrees with rooted \( \tau \) bisimulation [Wal90].

Again, by looking at the term deduction systems and knowing that all the theories are sound and particularly \( ST \) and \( ST \perp \) are complete, theorems of this article can be applied and, by interpreting arrows as before, Figure 2 is obtained as a result. In addition, since \( ST \) is complete for the preorder, CCS is also complete because the new operators can be eliminated. Similarly, \( CCS \perp \) is complete since \( ST \perp \) is complete and \( CCS \perp \) has the elimination property. Nevertheless, nor \( ST \perp \) neither \( CCS \perp \) have the elimination property with respect to \( ST \) or CCS.

Moreover, it deserves to notice that the results labelled with a • are new in this article.

Conclusions

This article extended the general conservative results of [Ver94b] with respect to preorders for transition system based process theories with inequalities. It only required reasonable and easy-to-check conditions. As a simple corollary, a general completeness theorem for inequational specifications was proved.

As it was explained above, the results of this article were already applied in [VB95]. Besides, an example was taken from the literature and results of conservativity and completeness were recreated. In addition, some new results on these examples were quickly proved by means of the techniques introduced here.

The use of preorders and inequational specifications is not so widely diffused as equivalences and equational specifications; perhaps it is due to the fact that they are more difficult to manage. However, results presented here above seem to make it easier.

References


A Well-formedness Checker for $\mu$CRL

Joris Hillebrand\textsuperscript{1}  Henri Korver\textsuperscript{2}*

\textsuperscript{1}University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

\textsuperscript{2}Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
E-mail: joris@fwi.uva.nl korver@phil.ruu.nl

Abstract

We present an algebraic specification for the static semantics of the process language $\mu$CRL. This specification can be executed in the ASF+SDF meta-environment which automatically derives a rewriting system for it. In this way, we immediately obtain a prototype of an enriched type checker (well-formedness checker) for $\mu$CRL. The presence of this tool will stimulate the use of $\mu$CRL, as ASF+SDF has provided us with a syntax-directed editor and a prettyprinter for $\mu$CRL as well.

Key Words & Phrases: Formal specification, Algebraic specification, Specification language, $\mu$CRL, Static semantics, Type checking, ASF+SDF

1 Introduction

This paper presents the algebraic specification of a well-formedness checker for the language $\mu$CRL. We call it a well-formedness checker as it can be considered as an enriched type checker, i.e., it checks the internal consistency of a $\mu$CRL specification and some extra properties that are desirable. As the semantics of $\mu$CRL specifications has only been defined for well-formed specifications, it is desirable to check for well-formedness on top of static-semantic correctness. The definition of $\mu$CRL can be found in [GP95].

The language $\mu$CRL can be compared with CRL [RBE+93], LOTOS [BB87, Bri87], PSF [MV90, Jva91], which are used for the specification of distributed systems. However, there are two main differences. First of all, $\mu$CRL has a considerably smaller syntax (no modularisation facilities etc.); in exchange, it has a sharply defined and well-understood semantic basis. Second, it features a proof theory for proving equality between data and processes.

The specification has been written in the ASF+SDF formalism (see [Kli93]). This approach has three main advantages:

- Our specification is highly readable as ASF+SDF has an algebraic structure and supports mix-fix notation. The constraints as have been posed in [GP95] correspond directly with the ASF+SDF code. In particular, each constraint can be related to an equation in our specification.
- We benefit from the incremental nature of ASF+SDF as $\mu$CRL is still under development.
- The specification can be executed such that we automatically obtain a well-formedness checker.

In this way, a working tool is generated from the formal description of the static semantics of

*Partly supported by the European Communities under RACE project no. 2076, Broadband Object Oriented Service Technology (BOOST), and the Foundation for Computer Science in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO).
μCRL. An example can be found in Figure 1, where we defined a constant function \texttt{foo} of sort \texttt{Bar} without defining the sort \texttt{Bar}.

- A syntax-directed editor and a prettyprinter are automatically generated from the language definition for μCRL. This will stimulate the use of μCRL.

As a final remark, we note that we follow the static semantics described in [GP95] exactly, so we do not comment on design choices regarding syntax or semantics.

The paper is organized as follows. In Section 2, \texttt{ASF+SDF} and its meta-environment are described. In Section 3, the language μCRL is introduced, and we list the properties for which we will check μCRL specifications. Section 4 describes the specification of the well-formedness checker. Finally, related and future work are discussed in Section 5. The SDF definition of the syntax of μCRL can be found in Appendix A.

Acknowledgements. The authors would like to thank the following persons for their help on many levels of the realization of this paper. Alban Ponse explained many details of μCRL and commented on earlier versions of this paper. Wilco Koorn suggested to map a μCRL specification onto a list of errors. He also assisted in the use of the \texttt{ASF+SDF} system. Paul Klint and Mark van den Brand made suggestions to keep this paper readable. Mark van den Brand also commented on earlier versions of the paper and the specification. Jan Rekers and Emma van der Meulen helped in the initial start-up of the \texttt{ASF+SOF} system. Eelco Visser has been of great help in the use of the \texttt{ToLATEX} tool. Without his help, the \texttt{ASF+SDF} code in this paper would have been far less readable.

2 The \texttt{ASF+SDF} Meta-environment

The well-formedness checker described in this paper has been written in the \texttt{ASF+SDF} specification formalism, a combination of \texttt{ASF} (Algebraic Specification Formalism) and \texttt{SDF} (Syntax Definition Formalism).

In \texttt{ASF} [BHK89b], we have modules describing data by means of functions and equations over these functions. Each module has a signature part that defines the abstract syntax of the functions. With these functions, data elements of a certain sort can be constructed. A set of conditional equations provides the functions with semantics. Data elements with the same meaning can be identified using the equations.

In \texttt{SDF} [HHKR89, HK89b], lexical, context-free, as well as abstract syntax can be defined simultaneously. From these definitions, parsers can be generated incrementally. This makes an implicit translation from text strings to abstract syntax trees possible.

The combination of \texttt{ASF} and \texttt{SDF} has resulted in a specification formalism, in which semantics are associated with the tree representation of text strings. This means that the equations are associated

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Example of an erroneous specification.}
\end{figure}
with the abstract syntax trees generated from text strings. It allows complete syntactic freedom. An
ASF+SDF specification consists of modules having the following sections: an imports section, where
other modules can be imported, a section in which the sorts of a signature can be defined, a section for
the definition of lexical syntax, one for the definition of context-free syntax, a section for the definition
of variables to be used in the equations, and a section for conditional equations.

The ASF+SDF meta-environment [Kli93] supports writing specifications in the ASF+SDF formal­
ism. The meta-environment makes it possible to execute ASF+SDF specifications, by generating
parsers from these specifications and deriving term rewriting systems for them. Also, prettyprinting
and syntax-directed editing of modules and terms are supported. This makes the ASF+SDF meta­
environment very suitable for prototyping and testing of specifications.

Example. This document contains a number of ASF+SDF equations. We will now give an example
that shows how these equations should be read.

variables
\[
\begin{align*}
N & \rightarrow [0-9'\ast] \text{ Name} \\
Xns & \rightarrow [0-9'\ast] \{\text{Name } \#\} + \\
Sig & \rightarrow [0-9'\ast] \text{ Signature} \\
Var & \rightarrow [0-9'\ast] \text{ Var} \\
Trms & \rightarrow [0-9'\ast] \{\text{Data-term } ,\} +
\end{align*}
\]
equations
\[
Xns = \text{sort-list(Sig, Var, Trms)}. \\
\text{is-function(sigfun(Sig), N : Xns) } \neq \text{ true} \\
ssc-t(Sig, Var, N(Trms)) = \text{Unknown function } N(Xns) \text{ in data-term} \tag{ssc-t-5}
\]
The first variable declaration states that \( N \) followed by zero or more digits or quotes is a variable of
sort Name. The second declaration states that \( Xns \) stands for one or more Names, separated by "#".
We call this an \( X \)-name-list. We will use this construction frequently throughout this paper. It is a
very usable format, since it is the format for the input signatures in function and action declarations.
Whenever in the sequel we want to have information about the sorts of a list of data terms or a list of
variable declarations, this information is returned as an \( X \)-name-list. Now matching function, action,
or process declarations can be found quickly.

The equation checks as to whether the data term \( N(Trms) \) is an application of a declared function.
First, the function \( \text{sort-list} \) calculates the input-signature of the data term, i.e., it returns the sorts
of the arguments as an \( X \)-name-list. This list is assigned to the variable \( Xns \). Next, the function
\( \text{is-function} \) looks for a function declaration with the given name \( (N) \) and input-signature \( (Xns) \).
If such a function declaration does not exist, the equation returns an error message \( (\text{Unknown function} \ldots) \).
Of course, there is also an equation that handles the case in which a proper function declaration does
exist.

3 \( \mu \)CRL

In this section, the specification language \( \mu \)CRL will be discussed shortly. \( \mu \)CRL stands for \( \mu \)-
Common Representation Language. CRL [RBE*93] was designed in the RACE (Research and
technology development in Advanced Communications in Europe) project SPECS (Specification and
Programming Environment for Communication Software). The purpose was to design a language
containing several features of current specification languages. Then each of these languages could
be translated into CRL. Tools for these languages would then only have to be written for CRL. See [RBE+93] for more information on SPECS and RACE.

$\mu$CRL contains some essential elements of CRL, but has far less language features. It focuses mainly on the specification of data and processes, without modularisation, to make a thorough analysis of processes with data possible. This has resulted in a proof theory, based on natural deduction, by means of which processes and data terms in a $\mu$CRL specification can be proven equal. $\mu$CRL has been defined in [GP95]. A description of the proof theory can be found in [GP94].

### 3.1 Syntax

As just stated, $\mu$CRL is a specification language for data and processes. The specification of data is done by the declaration of sorts and functions over these sorts. Next, the functions can be given a meaning by specifying a many-sorted term rewriting system. See the following example, where we specify the natural numbers with addition.

```
sort Natural
func 0: Natural → Natural
      succ: Natural → Natural
      add: Natural # Natural → Natural

var m,n: Natural
rew add(0,n) = n
      add(succ(m),n) = succ(add(m,n))
```

Processes are described in the style of CCS ([Mil80]), CSP ([Hoa85]), or ACP ([BK84a], [BK84b], for an overview see [BW90]). The syntax of the process expressions has been taken from ACP. Processes are built from atomic actions, using particular operators, e.g., for alternative or sequential composition. As in ACP, it is possible to define communications between atomic actions. In the next example, we specify a two-bit buffer over the naturals.

```
act read1, read2, send2, comm2, send3: Natural

comm read2 | send2 = comm2

proc OneBitBuff1 = sum(n:Natural, read1(n) . send2(n)) . OneBitBuff1
OneBitBuff2 = sum(n:Natural, read2(n) . send3(n)) . OneBitBuff2
TwoBitBuff = hide({comm2}, encap({read2, send2}, OneBitBuff1 || OneBitBuff2))
```

For the formal syntax of $\mu$CRL we refer to [GP95]. There the syntax is given in BNF. An SDF definition of the syntax can be found in Appendix A. Also [GP95] contains an SDF definition, but it differs from the BNF definition when it comes to process expressions. Our SDF definition matches the original BNF definition.

Our SDF definition does not have sort names for list constructs, since lists have been built into the ASF+SDF formalism. For example, a sequence of one or more $Names$ is not called a $Space-name-list$, but is written $Name+$. We also use ASF+SDF lists to denote that a $\mu$CRL specification consists of one or more smaller specifications. Therefore, we define sort, function, rewrite, action, communication, and process specifications to be $Single-specifications$. Next, we define a $Specification$ as a sequence of one or more $Single-specifications$:
3.2 Static semantics

In this section, we give an informal formulation of the constraints we pose on a μCRL specification. The list of constraints will be followed by a large example. In section 4, we will show how we can formalize these constraints.

Sorts
1. All sort names must be different.

Functions
1. No two functions can have the same name and the same input signature.
2. The input sorts and the output sort of every function must be declared.

Variables
1. All variable names must be different.
2. A variable can not have the same name as a function, action, or process without arguments.
3. The sort of each variable must be declared.

Rewrite rules
1. The data term at the left-hand side of the equation must be an application of a declared function.
2. The two sides of a rewrite rule should be correctly-typed data terms of the same sort.
3. The variables that appear on the right-hand side of the equation must also appear on the left-hand side of the equation.

Data terms
1. A data term must either be a declared variable, or an application of a declared function.
2. If a data term is a function application, its arguments, if any, must be correctly-typed data terms.

Actions
1. No two actions can have the same name and the same input signature.
2. An action can not have the same name and the same input signature as a function.
3. All sorts in an input signature must be declared.
Communications
1. At most one communication can be defined on each pair of action names.
2. For each action name in a communication declaration an action must be declared.
3. The actions declared for one of the action names on which a communication is defined must have the same input signatures as the actions declared for the other action name on which that communication is defined.
4. The actions declared for the action name that is the result of a communication must have all the input signatures of the actions declared for the action names on which that communication is defined.

Processes
In this paragraph, we say that process declarations have the form Process header = Process body.
1. No two process declarations can have the same header, i.e., the same name and the same input signature.
2. A process header can not have the same name and the same input signature as a function or an action.
3. The constraints on variables as mentioned above apply to the sequence of variable declarations (which can be empty) in a process header. These variables are local to the process declaration.
4. A process body must be a process expression of one of the following forms. Correctness of a process expression is always defined with respect to its local variables. The following priority exists between the process operators: "+" < "||" < "|" < "<" < "!" < "=" < "\".
   (a) $p_1 + p_2$, in which case $p_1$ and $p_2$ must be correctly-typed process expressions.
   (b) $p_1 \parallel p_2$, and see 4a.
   (c) $p_1 \downarrow p_2$, and see 4a.
   (d) $p_1 \mid p_2$, and see 4a.
   (e) $p_1 \cdot p_2$, and see 4a.
   (f) $p_1 \& t \& p_2$, in which case $t$ must be a correctly-typed data term of sort "Bool", and $p_1$ and $p_2$ must be correctly-typed process expressions.
   (g) $\delta$
   (h) $\tau$
   (i) $\vartheta\!\{n_1, \ldots, n_m\}, p$, in which case all names $n_1, \ldots, n_m$ must be different, for each of these names an action with that name must be declared, and $p$ must be a correctly-typed process expression.
   (j) $\tau\!\{n_1, \ldots, n_m\}, p$, and see 4i.
   (k) $\rho\!\{n_1 \rightarrow n'_1, \ldots, n_m \rightarrow n'_m\}, p$, in which case all names $n_1, \ldots, n_m$ must be different, for each name $n_1, \ldots, n_m, n'_1, \ldots, n'_m$ an action must be declared, the input signatures of the actions declared for a name $n_k$ must also be input signatures of the actions declared for the name $n'_k$, and $p$ must be a correctly-typed process expression.
   (l) $\Sigma\!(x : S, p)$, in which case the constraints on variables as mentioned above apply on the variable declaration $x : S$, and $p$ must be a correctly-typed process expression.
   (m) $n$, in which case an action or process $n$ without input must be declared.
(n) \( n(t_1, \ldots, t_m) \), in which case an action or process \( n \) with an input signature that matches the sorts of \( t_1, \ldots, t_m \) must be declared. Moreover, \( t_1, \ldots, t_m \) must be correctly-typed data terms.

(o) \( p \), in which case \( p \) must be a correctly-typed process expression.

Example. Below we give a specification that contains a lot of errors. We refer to the constraints mentioned above as follows: \( \mathbf{V2} \) refers to the second constraint for Variables.

**sort**

\[
\begin{align*}
\text{X} & \rightarrow \text{X} \\
\text{Y} & \rightarrow \text{Y} \\
\text{Z} & \rightarrow \text{Z}
\end{align*}
\]

**func**

\[
\begin{align*}
c & : \text{c} \rightarrow \text{X} \\
d & : \text{d} \rightarrow \text{Y} \\
f & : \text{f} : \text{X} \neq \text{Y} \rightarrow \text{X} \\
f & : \text{f} : \text{Y} \neq \text{X} \rightarrow \text{X} \\
f & : \text{f} : \text{Y} \neq \text{X} \rightarrow \text{Y} \\
g & : \text{g} : \text{X} \neq \text{Y} \rightarrow \text{Z}
\end{align*}
\]

**var**

\[
\begin{align*}
c, v1, v2 & : \text{X} \\
v1, v3 & : \text{Y} \\
v4 & : \text{Z}
\end{align*}
\]

**rew**

\[
\begin{align*}
v3 & = \text{d} \\
c & = \text{d} \\
c & = \text{f}(v2, v3) \\
f(v2, v3) & = \text{e} \\
f(v2, v3) & = \text{f}(v2, v4)
\end{align*}
\]

**act**

\[
\begin{align*}
a, b, e & : c \\
a, b, e & : \text{Y} \\
e & : \text{Z}
\end{align*}
\]

**comm**

\[
\begin{align*}
a & | b = e \\
b & | a = e \\
a & | b = k \\
b & | e = a
\end{align*}
\]

**proc**

\[
\begin{align*}
P & = a + b \\
P(v:X) & = a + b \\
P(w:X) & = a \cdot b \\
Q(v:X, v:Y) & = a + b \\
d & = a + b \\
b(v:X) & = a + b \\
P1 & = \partial(\{a, b, a\}, a + b) \\
P2 & = \tau(\{a, b, k\}, a + b) \\
P3 & = \rho(\{a \rightarrow b, a \rightarrow e\}, a + b) \\
P4 & = \rho(\{e \rightarrow a\}, a + b) \\
P5 & = \rho(\{a \rightarrow k\}, a + b) \\
P6 & = \sum(c : Y, a + b) \\
P7 & = k \\
P8 & = a(d)
\end{align*}
\]
3.3 Well-formedness

In the previous section, we defined under which conditions a μCRL specification is correct, i.e., when it is internally consistent.

There are, however, some extra properties one would like to impose on a specification. For instance, to give the if-then-else process construction a meaning, one would like to have exactly two boolean constants: true ("T") and false ("F"). Unfortunately, it is undecidable whether or not true can be rewritten into false (or vice versa) in a specification, or whether a specification has any other boolean normal forms. Therefore, the best we can do is to make sure that the sort "Bool" with constants "T" and "F" has been declared. Furthermore, we want the communication function not only to be commutative, but also associative, since this is the case in ACP. Finally, we do not want empty sorts, because this would make it very hard to provide a specification with semantics.

A μCRL specification fulfilling the following requirements is called well-formed:

1. The specification is statically semantically correct as described in the previous section.
2. The communication function is associative.
3. The specification has no empty sorts. This means that for each sort there exists a closed data term.
4. The sort "Bool" is declared with constant functions "T" and "F".

Example. Consider the following specification:

\[
\begin{align*}
\text{sort} & \quad X \\
\text{func} & \quad f: X \rightarrow X \\
\text{act} & \quad a \ b \ c \ d \ e \\
\text{comm} & \quad a \ | \ b = c \\
& \quad c \ | \ d = e
\end{align*}
\]

In this specification the communication function is not associative. We have that \((a | b) | d = e\)\), but \(a | (b | d) = e\) is not derivable. Communication declarations \(b | d = f\) and \(a | f = e\) should be added for some action \(f\). Furthermore, sort \(X\) is empty. There should at least be one constant function of sort \(X\). Clearly, the booleans have not been defined.

4 A well-formedness checker

In this section, we specify a function that checks whether or not a μCRL specification is well-formed. The function returns a list of errors when the specification is not well-formed. Since the first demand for well-formedness is that a specification is statically semantically correct, we start to specify this property.

We already mentioned that a μCRL specification consists of the declaration of sorts, functions over these sorts, variables, rewrite rules, actions, and communications and processes over these actions. The language μCRL has a few properties worth mentioning. First of all, it is not necessary that, for instance, all actions are declared in one action specification. A μCRL specification can contain several sort, function, variable, rewrite, action, communication, and process specifications. Also, μCRL has “application-before-definition”, which means that, for instance, one can declare a function
of which the output sort is declared after the function definition. Therefore, we first collect all sort, function, action, communication, and process declarations from a given specification. The five-tuple thus obtained is called the signature of the specification. Next, we examine whether the specification is statically semantically correct with respect to its signature.

The signature of the entire specification will be an argument of every higher-level check function, what makes it a kind of global variable or type environment. This approach allows us to check small pieces of the specification, while having access to relevant information about other parts of the specification.

The following variables are used throughout the paper:

<table>
<thead>
<tr>
<th>variables</th>
<th>variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$ [0-9]'* → Specification</td>
<td>$Fd$ [0-9]'* → Function-declaration</td>
</tr>
<tr>
<td>$SE$ [0-9]'* → Single-specification</td>
<td>$Fds$ [0-9]'* → Function-declaration+</td>
</tr>
<tr>
<td>$SEs$ [0-9]'* → Single-specification+</td>
<td>$OFds$ [0-9]'* → Function-declaration*</td>
</tr>
<tr>
<td>$Sig$ [0-9]'* → Signature</td>
<td>$Ads$ [0-9]'* → Action-declaration+</td>
</tr>
<tr>
<td>$Names$ [0-9]'* → Names</td>
<td>$OAds$ [0-9]'* → Action-declaration*</td>
</tr>
<tr>
<td>$Fun$ [0-9]'* → Fun</td>
<td>$Cds$ [0-9]'* → Communication-declaration+</td>
</tr>
<tr>
<td>$Act$ [0-9]'* → Act</td>
<td>$OCds$ [0-9]'* → Communication-declaration*</td>
</tr>
<tr>
<td>$Comm$ [0-9]'* → Comm</td>
<td>$Pds$ [0-9]'* → Process-declaration+</td>
</tr>
<tr>
<td>$Proc$ [0-9]'* → Proc</td>
<td>$Trm$ [0-9]'* → Data-term</td>
</tr>
<tr>
<td>$Var$ [0-9]'* → Var</td>
<td>$Trms$ [0-9]'* → {Data-term &quot;&quot;,&quot;}+</td>
</tr>
<tr>
<td>$Err$ [0-9]'* → Error</td>
<td>$Pre$ [0-9]'* → Process-expression</td>
</tr>
<tr>
<td>$Errs$ [0-9]'* → {Error &quot;:;&quot;,}+</td>
<td>$Rws$ [0-9]'* → Rewrite-rule+</td>
</tr>
<tr>
<td>$OErrs$ [0-9]'* → {Error &quot;:;&quot;}*</td>
<td>$Svds$ [0-9]'* → {Single-variable-declaration &quot;:&quot;,}+</td>
</tr>
<tr>
<td>$N$ [0-9]'* → Name</td>
<td>$Var-sec$ [0-9]'* → Variable-declaration-section</td>
</tr>
<tr>
<td>$Ns$ [0-9]'* → Name+</td>
<td>$Types$ [0-9]'* → Types</td>
</tr>
<tr>
<td>$ONs$ [0-9]'* → Name*</td>
<td></td>
</tr>
<tr>
<td>$Xns$ [0-9]'* → {Name &quot;:#&quot;}+</td>
<td></td>
</tr>
<tr>
<td>$OXns$ [0-9]'* → {Name &quot;:#&quot;}*</td>
<td></td>
</tr>
</tbody>
</table>

4.1 Signature

As stated above, the signature of a µCRL specification is a five-tuple, of which each component is a sequence of respectively sorts, functions, actions, communications, and processes declared.

In order to be able to define a signature, we first define these sequences.

context-free syntax

```
"[" Name* "]" → Names
"[" Function-declaration* "]" → Fun
"[" Action-declaration* "]" → Act
"[" Communication-declaration* "]" → Comm
"[" Process-declaration* "]" → Proc
```

As one can see, a sequence is just zero or more elements between square brackets. Since a sort is just a name, the sequence of sorts is a sequence of names, called Names. Furthermore, we define the concatenation of two sequences, and a function that checks whether a specific element is present in a sequence:
Naturally, the same functions have been defined for the other sequences as well. Now we can define a signature as follows:

context-free syntax

\[ "(\text{Names}, \text{Fun}, \text{Act}, \text{Comm}, \text{Proc})" \rightarrow \text{Signature} \]

We add a function \( \text{sig} \) that gives the signature of a specification, and functions to extract the different components from a signature.

context-free syntax

\[
\begin{align*}
\text{sig}(\text{Specification}) & \rightarrow \text{Signature} \\
\text{sigsort}(\text{Signature}) & \rightarrow \text{Names} \\
\text{sigfun}(\text{Signature}) & \rightarrow \text{Fun} \\
\text{sigact}(\text{Signature}) & \rightarrow \text{Act} \\
\text{sigcomm}(\text{Signature}) & \rightarrow \text{Comm} \\
\text{sigproc}(\text{Signature}) & \rightarrow \text{Proc}
\end{align*}
\]

These functions are defined by conditional equations. Below, we give a few examples.

\[
\begin{align*}
\text{sig}(\text{func } Fds) &= ([], [Fds], [], [], []) \quad \text{[sig-2]} \\
\text{sig}(SE) &= (\text{Names}_1, \text{Fun}_1, \text{Act}_1, \text{Comm}_1, \text{Proc}_1) \\
\text{sig}(SEs) &= (\text{Names}_2, \text{Fun}_2, \text{Act}_2, \text{Comm}_2, \text{Proc}_2) \\
\text{sig}(SE SEs) &= (\text{Names}_1 + \text{Names}_2, \\
& \quad \text{Fun}_1 + \text{Fun}_2, \\
& \quad \text{Act}_1 + \text{Act}_2, \\
& \quad \text{Comm}_1 + \text{Comm}_2, \\
& \quad \text{Proc}_1 + \text{Proc}_2) \quad \text{[sig-7]} \\
\text{sigsort}((\text{Names}, \text{Fun}, \text{Act}, \text{Comm}, \text{Proc})) &= \text{Names} \quad \text{[sigsort-1]}
\end{align*}
\]

4.2 Errors

Our top-level type-check function is defined as follows:

context-free syntax

\[ \text{ssc}(\text{Specification}) \rightarrow \text{Errors} \]

It maps a given specification onto a list of errors. Therefore, we have defined an error message for every constraint in Sections 3.2 and 3.3. When a constraint is violated a corresponding error message is generated. The error messages are mostly mix-fix functions that allow pieces of the specification to be given as arguments. In this way, the context of an error can be given with the error message. For example, the following error message is generated when some sort has been multiply declared:

context-free syntax

\[ "\text{Sort}" \text{ Name } "\text{has}" "\text{been}" "\text{defined}" "\text{more}" "\text{than}" "\text{once}" \rightarrow \text{Error} \]
Here, the Name argument is used for the sort name concerned. The other words in the error message have been quoted separately for ease of use. For instance, it is now allowed to split the message in two lines.

The function join concatenates two lists of errors. The message No errors were found is a special constant. It is returned by the type-check function if a specification (or a piece of it) is error-free. Since the entire specification is divided into smaller pieces for type checking, this message is generated for every piece that is error-free. Therefore, multiple appearances of the same error message have to be removed. Also, the special constant No errors were found has to be removed as soon as any other (real) errors are found. This is expressed by the following equations:

\[
\begin{align*}
\text{join}(\text{Errs}_1, \text{Errs}_2) &= \text{Errs}_1; \text{Errs}_2 & \text{(join-1)} \\
O\text{Errs}_1; \text{Err}; O\text{Errs}_2; \text{Err}; O\text{Errs}_3 &= O\text{Errs}_1; \text{Err}; O\text{Errs}_2; O\text{Errs}_3 & \text{(errs-1)} \\
O\text{Errs}; \text{No errors were found}; \text{Err} &= O\text{Errs}; \text{Err} & \text{(errs-2)} \\
\text{Errs}; \text{No errors were found}; O\text{Errs} &= \text{Errs}; O\text{Errs} & \text{(errs-3)}
\end{align*}
\]

### 4.3 Static semantics

As stated earlier, we will check a specification piece by piece, while we use the signature of the entire specification as a sort of global type environment. This is expressed by the top equation for the type-check function \( \text{ssc} \):

\[
\text{ssc}(E) = \text{ssc-spec}(\text{sig}(E), E) \tag{ssc-1}
\]

Next, we split the specification into single specifications:

\[
\begin{align*}
\text{ssc-spec}(\text{Sig}, SE SEs) &= \text{join}(\text{no-sort-overlap}(\text{sigsort}(\text{sig}(SE)), \text{sigsort}(\text{sig}(SEs))), \\
&\quad \text{join}(\text{no-fun-overlap}(\text{sigfun}(\text{sig}(SE)), \text{sigfun}(\text{sig}(SEs))), \\
&\quad \text{join}(\text{no-act-overlap}(\text{sigact}(\text{sig}(SE)), \text{sigact}(\text{sig}(SEs))), \\
&\quad \text{join}(\text{no-comm-overlap}(\text{sigcomm}(\text{sig}(SE)), \text{sigcomm}(\text{sig}(SEs))), \\
&\quad \text{join}(\text{no-proc-overlap}(\text{sigproc}(\text{sig}(SE)), \text{sigproc}(\text{sig}(SEs))), \\
&\quad \text{join}(\text{ssc-spec}(\text{Sig}, SE), \\
&\quad \text{ssc-spec}(\text{Sig}, SEs))))))
\end{align*}
\]

This equation shows that a sequence of two or more single specifications is correct if each of the single specifications is correct. Moreover, if a specification consists of two or more single specifications, we must make sure that each sort, function, action, communication, or process declaration appears in only one single specification. Although each single specification will later be checked for double declarations separately, it is still possible that the same declaration appears in two single specifications.

Now, each single specification can be checked separately, with respect to the signature of the entire specification:

\[
\begin{align*}
\text{ssc-spec}(\text{Sig}, \text{sort } Ns) &= \ldots & \text{(ssc-spec-1)} \\
\text{ssc-spec}(\text{Sig}, \text{func } Fds) &= \ldots & \text{(ssc-spec-2)} \\
\text{ssc-spec}(\text{Sig}, \text{Var-sec rew } Rws) &= \ldots & \text{(ssc-spec-3)} \\
\text{ssc-spec}(\text{Sig}, \text{act } Ads) &= \ldots & \text{(ssc-spec-4)} \\
\text{ssc-spec}(\text{Sig}, \text{comm } Cds) &= \ldots & \text{(ssc-spec-5)} \\
\text{ssc-spec}(\text{Sig}, \text{proc } Pds) &= \ldots & \text{(ssc-spec-6)}
\end{align*}
\]

We have divided the constraints formulated in Sections 3.2 and 3.3 in a number of categories. We shall now show how we have specified the necessary check functions for these categories.
4.3.1 Identification

The first property to check a specification for is whether it is possible to uniquely identify functions, actions, processes, and variables. For instance, an application of a function with a certain name and a certain input signature can not be identified if two or more functions with that name and that input signature have been declared.

First of all, each sort can only be declared once:

\[
\text{ssc-spec}(\text{Sig}, \text{sort } Ns) = \text{uniq-sorts}(\{Ns\})
\]

\[
\text{uniq-sorts}([]) = \text{No errors were found}
\]

\[
N \in [ONS] \neq \text{true} \\
\text{uniq-sorts}([N ONS]) = \text{uniq-sorts}([ONS])
\]

\[
N \in [ONS] = \text{true} \\
\text{uniq-sorts}([N ONS]) = \text{join}('\text{Sort } N \text{ has been defined more than once,} \text{ uniq-sorts}([ONS])')
\]

For each sort in the sequence of declared sorts we look whether it is also present in the rest of the sequence. If this is the case, that sort was declared more than once.

We can not handle the function declarations in exact the same way, because it is not enough to check that all function declarations are pairwise different. Two functions having the same name and the same input signature, but different output sorts, are also not allowed. Therefore, we will use the auxiliary function is-function. This function takes the name and arguments, if any, of a function declaration, and checks whether or not another function with that name and those arguments has been declared.

\[
\text{is-function}([OFds], N) \neq \text{true} \\
\text{uniq-fun}([N: \rightarrow N' OFds]) = \text{uniq-fun}([OFds])
\]

or, in case of a function with arguments:

\[
\text{is-function}([OFds], N: Xns) \neq \text{true} \\
\text{uniq-fun}([N: Xns \rightarrow N' OFds]) = \text{uniq-fun}([OFds])
\]

with is-function defined as follows:

\[
\text{is-function}([OFds], N: \rightarrow N' OFds2), N) = \text{true} \\
\text{is-function}([OFds], N: Xns \rightarrow N' OFds2), N: Xns) = \text{true}
\]

We explicitly keep different equations for functions with and without arguments, because is-function will also be called with action declarations later. Processes are dealt with in the same way.

With communications, we have to take care of the fact that the communication function is commutative:

\[
\text{is-comm}([OCds1, N1 | N2 = N3 OCds2], N1 | N2) = \text{true} \\
\text{is-comm}([OCds1, N1 = N3 OCds2], N1 | N2) = \text{true}
\]

Just as we do not want two functions with the same name and the same input signature, we do not want an action to have the same name and the same input signature as a function. In that case we would not be able to distinct that function from that action.
Therefore, one of the checks for actions is done by calling is-function:

\[
\text{is-function}(\text{sigfun}(\text{Sig}), \text{N}) \neq \text{true} \quad \text{[check-ad-1]}
\]

\[
\text{check-ad}(\text{Sig}, \text{N}) = \text{No errors were found}
\]

\[
\text{is-function}(\text{sigfun}(\text{Sig}), \text{N} : \text{Xns}) = \text{true} \quad \text{[check-ad-4]}
\]

\[
\text{check-ad}(\text{Sig}, \text{N} : \text{Xns}) = N(\text{Xns}) \text{ has been defined as both a function and an action}
\]

Here, \( N \) is the action declaration concerned, with or without input (\( \text{Xns} \)). Likewise, we do not want a process to have the same name and input signature as a function or an action:

\[
\text{Xns} = \text{var-sorts}([\text{Svds}])
\]

\[
\text{is-function}(\text{sigfun}(\text{Sig}), \text{N} : \text{Xns}) = \text{true} \quad \text{[check-pd-5]}
\]

\[
\text{check-pd}(\text{Sig}, N(\text{Svds}) = \text{Pre}) = N(\text{Xns}) \text{ has been defined as both a function and a process}
\]

\[
\text{is-action}(\text{sigact}(\text{Sig}), \text{N}) = \text{true} \quad \text{[check-pd-3]}
\]

\[
\text{check-pd}(\text{Sig}, \text{N} = \text{Pre}) = N \text{ has been defined as both an action and a process}
\]

Here, \( N = \text{Pre} \) (or \( N(\text{Svds}) = \text{Pre} \), in the case of a process with input) is the process declaration concerned. Because we do not care about the names of the variables in the process header, we use the auxiliary function \( \text{var-sorts} \) to remove these names. The result is returned as an X-name-list:

\[
\text{var-sorts}([N : \text{IV'}]) = \text{N} \quad \text{[var-sorts-1]}
\]

\[
\text{Xns} = \text{var-sorts}([\text{Svds}])
\]

\[
\text{var-sorts}([N : \text{N'}, \text{Svds}]) = \text{N'} \neq \text{Xns} \quad \text{[var-sorts-2]}
\]

### 4.3.2 Declaration of input and output sorts

Functions have an output sort and can have input sorts. Actions and processes can have data arguments. In all these cases the arguments are data terms of a certain sort. These sorts are specified in the declaration of functions, actions, and processes. We have to check whether these sorts are declared sorts, which should be the case. For processes, this is done by checking the variable declarations in the header. One of the constraints for variables is that their sort has been declared:

\[
\text{N in sigsort}(\text{Sig}) \neq \text{true} \quad \text{[check-vd-2]}
\]

\[
\text{check-vd}(\text{Sig}, \text{N} : \text{N'}) = \text{Unknown sort N' of variable N}
\]

For function and action declarations, we have functions that check whether the associated sorts have been declared. For functions, this is the function \( \text{sorts-defined-in-fun} \):

\[
\text{sorts-defined-in-fun}(\text{Names}, [\text{OXns} \neq \text{N'}], \text{N} : \text{OXns} \rightarrow \text{N'})
\]

Here, \( \text{Names} \) is a sequence containing all the sorts declared in the specification (the first element of the signature-tuple). The last argument is the function declaration, of which we want to check whether its input and output sorts have been declared. Its input sorts (\( \text{OXns} \)) and its output sort (\( \text{N'} \)) have been collected in the second argument. Note that the third argument is only used in the error message.
Now \texttt{sorts-defined-in-fun} checks for each sort name in the second argument whether or not it is contained in the sequence of all declared sorts in the first argument:

\[
\text{sorts-defined-in-fun}(\text{Names}, [], \text{Fd}) = \text{No errors were found}
\]

\[
N \in \text{Names} = \text{true}
\]

\[
\text{sorts-defined-in-fun}(\text{Names}, [N \# \text{OXns}], \text{Fd}) =
\text{sorts-defined-in-fun}(\text{Names}, [\text{OXns}], \text{Fd})
\]

\[
N \in \text{Names} \neq \text{true}
\]

\[
\text{join}([\text{Unknown sort } N \text{ in function declaration } \text{Fd},
\text{sorts-defined-in-fun}(\text{Names}, [\text{OXns}], \text{Fd})]
\]

For action declarations we have the function \texttt{sorts-defined-in-act}:

\[
\text{sorts-defined-in-act}(\text{sigsort}(\text{Sig}), [\text{Xns}], N : \text{Xns})
\]

with the action declaration to be investigated in the third argument, and its input sorts in the second argument. The function \texttt{sorts-defined-in-act} has exactly the same behaviour as \texttt{sorts-defined-in-fun}. In fact, if we had left out the third argument, we would not have needed two functions. The reason to use two separate functions (with three arguments) all the same is to be able to generate more specific error messages.

### 4.3.3 Declaration of applied items

Functions and variables that are used (called) in rewrite rules or process bodies must have been declared. The same holds for actions and processes called in a process body. Since variables can be used in rewrite rules as well as in process bodies, we define correctness of rewrite rules and process bodies not only with respect to the signature, but with respect to the signature and a set of variables. For rewrite rules, this set of variables (\texttt{Var}) is formed by the variables declared in a rewrite specification. In the case of a process body, \texttt{Var} is formed by the variables declared in the process header.

To start with, the left-hand side of a rewrite rule must be an application of a declared function:

\[
\text{Xns} = \text{sort-list}((\text{Sig}, \text{Var}, \text{Trms}),
\]

\[
\text{is-function}(\text{sigfun}(\text{Sig}), N : \text{Xns}) \neq \text{true}
\]

\[
\text{check-rw}(\text{Sig}, \text{Var}, N(\text{Trms}) = \text{Trm}) =
\]

\[
\text{Unknown function } N(\text{Xns}) \text{ on left-hand-side of rewrite-rule}
\]

Here, \texttt{sort-list} calculates the sort of each data term in a list of data terms and returns these sorts as an \texttt{X-name-list}.

Actions and processes called in process bodies must have been declared as well:

\[
\text{is-process}(\text{sigproc}(\text{Sig}), N) = \text{true}
\]

\[
\text{ssc-be}(\text{Sig}, \text{Var}, N) = \text{No errors were found}
\]

\[
\text{is-action}(\text{sigact}(\text{Sig}), N) = \text{true}
\]

\[
\text{ssc-be}(\text{Sig}, \text{Var}, N) = \text{No errors were found}
\]

\[
\text{Xns} = \text{sort-list}((\text{Sig}, \text{Var}, \text{Trms}),
\]

\[
\text{is-process}(\text{sigproc}(\text{Sig}), N(\text{Xns})) \neq \text{true},
\]

\[
\text{is-action}(\text{sigact}(\text{Sig}), N : \text{Xns}) \neq \text{true}
\]

\[
\text{ssc-be}(\text{Sig}, \text{Var}, N(\text{Trms})) = \text{Unknown process or action } N(\text{Xns}) \text{ in process expression}
\]

Here, \texttt{N} is the function, variable, action, or process called, with or without arguments (\texttt{Trms}).
4.3.4 Communications

An interesting aspect of μCRL is the way communication works. Communication takes place between actions. However, communications are defined on action names. If one defines communication between two action names, all actions defined with these names can communicate. Another example:

\[
\text{act } a, b, c \\
\text{a, b, c: Natural} \\
c: \text{Natural} \# \text{Natural}
\]

\[
\text{comm } a \mid b = c
\]

In this example, two pairs of actions can communicate: "a" can communicate with "b" into "c" and "a: Natural" can communicate with "b: Natural" into "c: Natural".

It will be checked that actions with the same input signatures have been defined for the action names in the communication declaration. In the above example these input signatures are the empty signature and "Natural". The actions resulting from the communication should also have been defined with these input signatures, although these actions can be defined with more input signatures. In the above example, "c" has also been defined with the input signature "Natural # Natural".

Because almost all the constraints on a communication declaration are defined in terms of the input signatures of action names, we will need the auxiliary function types. It returns a list of all the input signatures with which actions with a certain name have been declared. In the following equations, the input signatures of actions with name \( N \) are collected:

\[
\text{types}([], N) = [] \quad [\text{types-1}]
\]

\[
\text{types}([N \ OAds], N) = [\text{empty-pattern}] + \text{types}([OAds], N) \quad [\text{types-2}]
\]

\[
\text{types}([N : Xns \ OAds], N) = [Xns] + \text{types}([OAds], N) \quad [\text{types-3}]
\]

\[
N \neq N \quad [\text{types-4}]
\]

\[
\text{types}([N' \ OAds], N) = \text{types}([OAds], N) \\ 
N \neq N \\
\text{types}([N' : Xns \ OAds], N) = \text{types}([OAds], N) \quad [\text{types-5}]
\]

As one can see, \text{types} returns the special constant \text{empty-pattern} when an action has been declared without input. Therefore, \text{types} only returns an empty list if no actions at all have been declared for a certain name.

Now we can perform the necessary checks on a communication declaration:

\[
\text{Types}_1 = \text{types}(\text{Act}, N_1), \\
\text{Types}_2 \neq [] \\
\text{equal}(\text{Types}_1, \text{types}(\text{Act}, N_2)) = \text{true}, \\
\text{Types}_1 \text{ sub-of types}(\text{Act}, N_3) = \text{true} \\
\text{check-cd}(\text{Act}, N_1 \mid N_2 = N_3) = \text{No errors were found} \quad [\text{check-cd-1}]
\]

In this equation, we check the communication declaration \( N_1 \mid N_2 = N_3 \). In the third condition of this equation we check whether the actions declared with name \( N_1 \) have the same input signatures as the actions declared with name \( N_2 \) (the third constraint for communications of Section 3.2). In the fourth condition, we check the fourth constraint for communications of Section 3.2. Since we demand that
Types \(T\) is not empty, at least one action must have been declared for each action name (the second constraint for communications of Section 3.2).

We now show the equations expressing the cases in which a communication declaration is incorrect:

\[
\text{check-cd}(Act, N_1 | N_2 = N_3) =
\]

- Unknown action-name \(N_1\) in communication declaration \(N_1 | N_2 = N_3\)

\[
\text{equal}(\text{types}(Act, N_1), \text{types}(Act, N_2)) \neq \text{true}
\]

- Input-patterns of action-names in communication declaration \(N_1 | N_2 = N_3\) do not match

\[
\text{check-cd}(Act, N_1 | N_2 = N_3) =
\]

- Input-patterns of action-names in communication declaration \(N_1 | N_2 = N_3\) do not match

Some of the constraints on communication declarations apply in a similar way to other language constructs in \(\mu\)CRL. For instance, in a process with an encapsulation or a hiding construct, the set also contains only action names. So also in those cases we have to make sure that actions with these names have been declared.

### 4.4 Well-formedness

In this section, we will give functions that check for the conditions, as formulated in Section 3.3, under which a specification is well-formed.

#### 4.4.1 Declaration of "Bool", "T", and "F"

Checking whether a sort or a function has been declared is easy. We did that all the time when checking for static-semantical correctness. We take the sort or function element of the signature, and check whether the desired sort or function is present:

\[
\text{name}("B", "o", "o", "1") \text{ in sigsort}(\text{sig}(E)) = \text{true}
\]

\[
\text{bool-declared}(E) = \text{No errors were found}
\]

\[
F : \rightarrow \text{name}("B", "o", "o", "1") \text{ in sigfun}(\text{sig}(E)) \neq \text{true}
\]

\[
\text{false-declared}(E) = \text{Function } 'F: \rightarrow \text{Bool}' \text{ should be defined}
\]

#### 4.4.2 No empty sorts

To check for empty sorts, we have to make sure that there exists a data term without variables for each sort declared. This is done for a given sort name by check-sort:

\[
\text{check-sort}(\text{Fun}, N, \text{Fun})
\]
Here, $N$ is the sort name investigated, and $Fun$ the function element of the signature. Now check-sort will scan $Fun$ for a function with output sort $N$. If such a function does not exist, sort $N$ is empty. If such function does exist, we have to make sure that none of its input sorts are empty. From this follows that if there is a constant function with output sort $N$, sort $N$ is not empty.

\[
\text{check-sort}([], N, Fun) = \text{Sort } N \text{ is empty} \tag{check-sort-1}
\]

\[
N' \neq N \Rightarrow \text{check-sort}([N : \text{OXns} \rightarrow N' \text{OFds}], N, Fun) = \text{check-sort}([\text{OFds}], N, Fun) \tag{check-sort-2}
\]

\[
\text{scan-sorts}(\text{make-sort-sequence}([\text{OXns}]), \text{remove-fd}(Fun, N' : \text{OXns} \rightarrow N)) = \text{No errors were found} \tag{check-sort-3}
\]

\[
\text{scan-sorts}(\text{make-sort-sequence}([\text{OXns}]), \text{remove-fd}(Fun, N' : \text{OXns} \rightarrow N)) \neq \text{No errors were found} \tag{check-sort-4}
\]

To avoid loops in the last two equations, we remove the function of which we check the input sorts. The auxiliary function make-sort-sequence turns an X-name-list into a sequence of sort names, by removing the #s between the names in the X-name-list.

### 4.4.3 Associativity of communication

Associativity of the communication function means that $(N_1 | N_2 | N) = N_1 | (N_2 | N_3)$ equals $(N_1 | N_3 | N_2)$. Now we check the communication function for associativity in a couple of steps. First we make the commutativity of the communication function explicit, so that we do not have to work two-way all the time. This is done by the function comm-star:

\[
\text{comm-star}([N_1 | N_2 = N_3 \text{OCds}]) = [N_1 | N_2 = N_3, N_2 = N_1 = N_3] + \text{comm-star}([\text{OCds}]) \tag{comm-star-2}
\]

Next, thirds looks for third parties for each communication declaration, that is, for each communication $N_1 | N_2 = N$ we look for communications $N | N_3 = N'$:

\[
\text{thirds}(N_1 | N_2 = N, [N | N_3 = N' \text{OCds}]) = [N | N_3 = N'] + \text{thirds}(N_1 | N_2 = N, [\text{OCds}]) \tag{thirds-2}
\]

\[
N' \neq N \Rightarrow \text{thirds}(N_1 | N_2 = N, [N' | N_3 = N' \text{OCds}]) = \text{thirds}(N_1 | N_2 = N, [\text{OCds}]) \tag{thirds-3}
\]

Finally, check-third checks for each communication $N_1 | N_2 = N$ and a third party $N | N_3 = N'$ (so that $N_1 | N_2 | N_3 = N'$) whether communications $N_2 | N_3 = N'$ and $N_1 | N' = N'$ are also present (so that also $N_1 | (N_2 | N_3) = N'$). The actual check is performed in assoc:

\[
\text{Comm} = [\text{OCds}, N_2 | N_3 = N' \text{OCds}], \\
\text{Comm} = [\text{OCds}', N_1 | N' = N' \text{OCds}] \\
\text{assoc}(N_1 | N_2 = N, N | N_3 = N', \text{Comm}) = \text{true} \tag{assoc-1}
\]
\[
\text{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', \text{Comm}) = \text{true}
\]
\[
\text{check-third}(N_1 \mid N_2 = N, N \mid N_3 = N', \text{Comm}) = \text{No errors were found}
\]

\[
\text{assoc}(N_1 \mid N_2 = N, N \mid N_3 = N', \text{Comm}) \neq \text{true}
\]
\[
\text{check-third}(N_1 \mid N_2 = N, N \mid N_3 = N', \text{Comm}) = \text{Communication \( N_1 \), \( N_2 \), \( N_3 \) is not associative}
\]

4.5 The Alternating Bit Protocol

As a final example we give the specification of the alternating bit protocol. See figure 2. We made one mistake: in the process \( S \), the undefined process \( X \) is called. Of course this should be \( S \) again.

\[
\text{check-third}(N_1 \mid N_2 = N, N \mid N_3 = N', \text{Comm}) = \text{No errors were found}
\]

Figure 2: Specification of the alternating bit protocol with an error.

When using the ASF+SDF system on a standard workstation, it takes about 16 seconds to check this specification.

5 Related and future work

5.1 Related work

Algebraic specifications of type checkers exist for various languages. All these specifications have in common that they have to keep track of information concerning names and types of various elements.
of the language. Sometimes a special construct (environment) is defined for this purpose. In other specifications, the necessary information is extracted from the program itself.

In [Deu94], a specification for the static semantics of Pascal has been given. A context is defined, in which the names and types of variables, constants, functions and procedures with the types of their parameters are kept. The context may be different at different points in the program (scope rules).

A similar approach can be found in the specifications of type checkers for the toy languages PICO [BHK89c, HK89a] and ASPLE [vdM88]. Since these languages are very small, all that is being stored in the type environment (PICO) or mode environment (ASPLE) are declarations of variables.

In the specifications of type checkers for the object-oriented languages POOL [Wal89] and TKOL [Din92] the entire program is passed as an argument to the type-check function, as is the case in our specification. In the case of POOL, functions have been written to extract information about names and types from the program. In the case of TKOL, a local type environment for a specific class is generated when needed. This local type environment contains the names and types of variables and the signature of methods local to the class concerned.

All these type checkers have in common that some type-check function is applied to the abstract-syntax tree of the program. The program is therefore checked part by part, according to the syntactical structure of the language. As a reference, either some context or environment, or the entire program is passed as an argument, containing all information necessary for type checking. This is also the approach chosen in the current paper. A $\mu$CRL program (specification) is checked part by part; the signature of the specification is passed for reference. The signature is nothing more than the original program, where all declarations of the same kind have been put together. Functions have been written to extract specific information from the context. Because $\mu$CRL has application-before-definition, a specification can not be checked from top to bottom in one pass, as would, for instance, be possible in Pascal, where forward references must be explicit.

In [Vis92], a different approach is chosen for the specification for the static semantics of the object-oriented language Eiffel. An advanced, reusable, context is defined. Now an Eiffel program, which is a number of classes, is split-up in separate classes that are put in the context. Next, the program itself is not used anymore, because all further operations are performed on the context. These further operations are a number of passes through the context. In each pass, present information is checked on internal consistency. Also, finer information is distilled from the present information. This finer information is again stored in the context and is checked in a next pass. This approach follows from the circularity (mutual dependency) that is allowed at some levels in Eiffel programs.

5.2 Future work

In order to prove properties of a $\mu$CRL specification, a proof theory has been developed in [GP94]. This theory is implemented in Coq [DFH+93], which is a theorem prover based on type theory and inductive types. In [Sel94] it is described how to translate $\mu$CRL specifications and proofs into Coq code. The translation simply assumes that $\mu$CRL specifications are well-formed. At the moment the compilation is performed by hand. Our well-formedness checker can be used as a preprocessor of a future tool that translates $\mu$CRL into Coq fully automatically.

Another possible direction for future work is the extension of $\mu$CRL with higher-level process algebra features and updating the well-formedness checker accordingly. Current research topics, such as the ToolBus [BK94], depend heavily on process algebra. Application of the verification techniques of $\mu$CRL to these systems could therefore be valuable. One can imagine detailed implementations to be proven equal to some abstract process specification. The ToolBus, however, contains some high-level process algebra features that are currently not present in $\mu$CRL. These features include iteration, priorities, state operator, and process creation, and could be added to $\mu$CRL.
In [KPvW] it is described how processes with parallel inputs and outputs can be modelled using *early read* actions and *process prefixes*. Extending μCRL with these features could be a nice exercise.

When we extend μCRL with new process algebra features, we have to adapt the well-formedness checker accordingly. Due to the modular set-up of the specification, this process will not be difficult. Imagine that we add a new operator, which has higher priority than "||", "\"\"", and "\"\"". Now we have to change

\[
\text{Parallel-expression} \quad \rightarrow \quad \text{Process-expression}
\]
\[
\text{Parallel-expression "+" Process-expression} \quad \rightarrow \quad \text{Process-expression}
\]

into

\[
\text{New-expression} \quad \rightarrow \quad \text{Process-expression}
\]
\[
\text{New-expression "+" Process-expression} \quad \rightarrow \quad \text{Process-expression}
\]
\[
\text{Parallel-expression} \quad \rightarrow \quad \text{New-expression}
\]
\[
\text{Parallel-expression "new" New-expression} \quad \rightarrow \quad \text{New-expression}.
\]

Next, we also have to change the type of the arguments of "+" in the equations for "+", which comes down to changing the names of a couple of ASF+SDF variables. Finally, we have to make new equations that specify the demands on the new operator. We can leave the other equations unchanged, because each operator has its own set of equations which specify the necessary constraints.

**Origin tracking.** Our well-formedness checker takes a μCRL specification and returns a list of possible shortcomings. An attractive feature would be the possibility to relate each error to its source in the μCRL specification. This principle is called *origin tracking* and is described in [DKT93].

In [Din94], a method is offered for origin tracking, departing from non-reduced type-check terms. The idea is that the type-check function returns a boolean value that is either true, or a non-standard value (a non-reduced term). This can be obtained by not specifying the "false" cases. In case of an error, some function will not reduce to true, leaving a non-reduced term. The information in this term can then be used for origin tracking. It is shown in [Din94] that for this method to work properly, one should avoid the use of conditions in the specification of the type checker. A way to remove conditions from equations in a specification is offered in [Din94].

Finally, [Din94] shows that high-quality origins can be established for specifications from the class of *primitive recursive schemes*. These are specifications with a syntax-directed nature. This is typically the case in our ASF+SDF code where the type-check function is distributed syntax-directed over a μCRL specification.

**References**


[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In *Algebraic specification* [BHK89a], pages 1–66.


[HK89b] J. Heering and P. Klint. The syntax definition formalism SDF. In Bergstra et al. [BH89a], pages 283–297.


[Wal89] H.R. Walters. The static semantics of POOL. In Bergstra et al. [BH89a], pages 163–198.
A MuCRL

imports Layout

exports

sorts Name Sort-specification Function-specification Function-declaration
Rewrite-specification Variable-declaration-section
Variable-declaration Data-term Rewrite-rules-section Rewrite-rule
Parallel-expression Merge-parallel-expression Comm-parallel-expression
Dot-expression
Basic-expression Renaming-declaration Single-variable-declaration
Process-specification Action-declaration Communication-specification
Communication-declaration Single-specification

lexical syntax
[a-zA-Z0-9]+ → Name

context-free syntax

sort Name+ → Sort-specification

func Function-declaration+ → Function-specification

{Name","}+ ":" {Name "#"}+ ":" Name → Function-declaration

Variable-declaration-section
Rewrite-rules-section → Rewrite-specification

var Variable-declaration+ → Variable-declaration-section

{Name ","}+ ":" Name → Variable-declaration

Name → Data-term

Name "(" {Data-term ","}+ ")" → Data-term

rew Rewrite-rule+ → Rewrite-rules-section

Name "=" Data-term → Rewrite-rule

Name "(" {Data-term ","}+ ")" "=" Data-term → Rewrite-rule

Parallel-expression → Process-expression

Parallel-expression "+" Process-expression → Process-expression

Merge-parallel-expression → Parallel-expression

Comm-parallel-expression → Parallel-expression

Cond-expression → Parallel-expression

Condition-expression "||" Cond-expression → Parallel-expression

Condition-expression "||" Cond-expression → Parallel-expression

Comm-parallel-expression Comm-parallel-expression → Comm-parallel-expression

Cond-expression "||" Cond-expression → Comm-parallel-expression

Comm-parallel-expression "||" Cond-expression → Comm-parallel-expression

Dot-expression dot Data-term "b" → Cond-expression

Dot-expression "d" Data-term "b" → Cond-expression

Dot-expression → Dot-expression

Basic-expression → Dot-expression

Basic-expression "." Dot-expression → Dot-expression
delta
tau
encap "(" "{Name ","}+ ")" ;
Process-expression ")"
hide "(" "{Name ","}+ ")" ;
Process-expression ")"
rename "(" "{Renaming-declaration ","}+ ")" ;
Process-expression ")"
sum "(" Single-variable-declaration ","
Process-expression ")"
Name
Name "(" {Data-term ","}+ ")"
"(" Process-expression ")"
Name "\rightarrow" Name
proc Process-declaration+
Name "\leftarrow" Process-expression
Name "(" {Single-variable-declaration ","}+ ")" ;
"=" Process-expression
act Action-declaration+
Name
{Name ","}+ ";" {Name ",#"}+ 
comm Communication-declaration+
Name ";" Name "\rightarrow" Name
Sort-specification
Function-specification
Rewrite-specification
Action-specification
Communication-specification
Process-specification
Single-specification+
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Basic-expression
-> Renaming-declaration
-> Single-variable-declaration
-> Process-specification
-> Process-declaration
-> Action-specification
-> Action-declaration
-> Communication-specification
-> Communication-declaration
-> Single-specification
-> Single-specification
-> Single-specification
-> Single-specification
-> Single-specification
-> Specification
Formalizing Process Algebraic Verifications in Coq

Marc Bezem
Jan Friso Groote
Department of Philosophy
Utrecht University
Heidelberglaan 8
NL-3584 CS Utrecht
The Netherlands
Email: {Marc.Bezem.JanFriso.Groote}@phil.ruu.nl

Roland Bol
Department of Computer Systems
Uppsala University
P.O.Box 325
S-751 05 Uppsala
Sweden
Email: rolandb@docs.uu.se

Abstract
We show in this paper how process algebra can be translated to the language of a general purpose proof checker. This is a first step towards the formal verification of correctness proofs of real-life protocols in process algebra. The process algebra we use is $\mu$CRL, ACPT augmented with data, which is small enough to make the verification feasible, and at the same time expressive enough for the specification of real-life protocols. The proof checker we use is Coq, which is based on the Calculus of Constructions, an extension of simply typed lambda calculus. We focus on motivating the need for formal verification, the pitfalls that must be avoided, and the resulting considerations regarding the translation from $\mu$CRL to Coq.

Keywords: formal verification, process algebra, ACP, $\mu$CRL, Coq, Calculus of Constructions.

1 Introduction
This paper reports on the first steps towards the formal verification of correctness proofs of real-life protocols in process algebra. The process algebra we use is based on the Algebra of Communicating Processes (ACP) of Bergstra and Klop [BK86]. More precisely, we use $\mu$CRL, ACPT augmented with data. For a complete description of its syntax and semantics we refer to [GP94], for its proof theory to [GP93b]. $\mu$CRL is small enough to make verifications feasible, and at the same time expressive enough for the specification of real-life protocols. The proof checker we use is Coq, which is based on the Calculus of Constructions, an extension of simply typed lambda calculus.

We focus on motivating the need for formal verification, the pitfalls that must be avoided, and the resulting considerations regarding the translation from $\mu$CRL to Coq. Due to the lack of space, we cannot demonstrate in this paper that proving the correctness of real-life protocols using this translation is feasible. This is demonstrated to some extent in the full version of this paper [BBG95], where the Alternating Bit Protocol is formally verified.

The word 'verification' usually refers to a mathematical proof in a combination of natural language and formal or informal mathematical notation. Consider for example the correctness proof of the Alternating Bit Protocol given in Sections 4.7 and 5.7 of [BW90]. It consists of a series of steps so small that the reader is convinced of the correctness of each step. Indeed, the proof in [BW90] is more detailed than most other verifications, because the intended reader is an undergraduate student.

For centuries, this form of verification was the best there was. But, as both the writer and the reader of the proof are human, what guarantee does it give that a proof is indeed correct? After all, to err...

*While carrying out this research, this author was employed by Eindhoven University of Technology, P.O.Box 513, NL-5600 MB Eindhoven, The Netherlands.
human. In some cases, especially now that computer programs and protocols are being incorporated in vital control systems, there is so much at stake that such a verification of a program is simply not enough. Especially in concurrent systems, where the number of situations can be exponential in the number of components, it is not at all unlikely that an unfortunate conjunction of circumstances is overseen during its design, testing, and verification-by-hand.

Recently it has become possible to let a computer program take over the role of the reader, or even that of the writer of proofs. In the first case such a program is called a proof checker, in the second case a theorem prover. The Coq-system, on which we focus in this paper, is a proof checker equipped with very limited theorem proving capabilities.

In contrast to a 'classical' verification, a formal verification is a proof formulated completely in a formal language; each step in it consists of the application of a formal proof rule. Theoretically, a formal verification could be done completely by hand, but on the basis of our experience (e.g. [Kam93]) we claim that, for real-life protocols, it can only be done using a computer. Such a verification is, by the nature of computers, a formal verification. To stress these observations, and also because a great deal of human input is still needed, we avoid the phrase ‘automatic verification’.

If a proof checker is convinced of the correctness of a proof, should we be convinced too? One can never hope to achieve absolutely guaranteed correctness. But we claim that formal verification can provide a significant increase in the level of confidence in a protocol. In order to support this claim, we investigate which errors remain possible. We see the following types.

1. Errors of the computer system (hardware, operating system, etc.). These are relatively rare, and moreover usually result in error messages and/or sudden termination of the program, rather than in an erroneous proof being accepted by the proof checker.

2. Errors in the underlying theory of the proof checker. This theory should be stable and well-understood. For Coq, simply typed lambda calculus [Bar92] is basic and the Calculus of Constructions [CH88] is well-understood. The theory of inductive types ([CP90, PM93], see Section 2) requires more study.

3. Programming errors in the proof checker. Indeed, the correctness of the proof checker must be checked thoroughly. As the program is much smaller (and more modular) than the proofs we intend to verify, the level of confidence in large proofs is definitely raised, even if it is still not 100%.

4. The system we want to verify is usually formalized in a base theory different from the language of the proof checker. In this paper, the base theory is μCRL. This base theory might contain errors, or, less dramatically, axioms and proof rules that do not always apply (such as CFAR in μCRL). In this case the formal proof is correct, but it does not prove what we think it does.

5. The formalization of the system in the base theory might be incorrect. Again, the formal proof is correct. This error is more likely to occur than the previous one, because the base theory remains fixed, whereas we formalize a different system each time.

6. In order to use a proof checker, we translate the base theory and the theorem under consideration to the language of the proof checker. This translation can introduce errors. The probability of the first three classes of errors can be reduced by verifying the same protocol on various different proof checkers (and platforms). The fourth and fifth class are orthogonal to the use of a proof checker. In this paper we concentrate on the translation of μCRL itself and μCRL-specifications to Coq. Special care must be taken when the translation of a specification deviates from its formalization ‘because it is convenient in this particular proof checker’. Such errors can remain undiscovered much easier than the others, as the translation of a particular specification is used less often, and by less people, than the computer, the proof checker, and the translation of the base theory.

These considerations indicate that the focus of the sceptical reader must shift from proofs to axioms: a proof is the most likely place to find an error in an ordinary verification, but the proofs of a formal verification are most probably correct; for the axioms there is no such guarantee.
Formal verification is not limited to algebraic verification of protocols. In principle, it can be used for any formalism [Cou93], for example I/O-automata [LMWF94, HSV94] and temporal logic [MP82, OL82, Hoo91]. However, these formalisms are based on exploring the complete state space of a protocol; therefore they suffer from the state explosion problem. Recent experience shows that the algebraic method discussed in this paper can handle large protocols as well [BG94a, KS93, GP93a].

In the next section, we give an introduction to Coq. Section 3 is the core of the paper: it discusses how μCRL was translated to Coq, and which problems arose. It also outlines the corresponding translation of μCRL-specifications. The research on the topic of this paper is only just beginning; therefore we conclude the paper with a list of directions for future research.

2 The Coq Proof Checker

For a complete overview of the Coq proof checker, we refer to [DFH+93]. It is based on the Calculus of Constructions, an extension of simply typed lambda calculus, but a deep understanding of that formalism, in particular of the identification of propositions and types, is not necessary for understanding the use we make of Coq (propositions are of type Prop and sorts of type Set). One can declare sorts, and state the existence of (constructor) functions with their types, including constants. One can express quantification and higher order logic. The implication and negation behave constructively.

Coq extends the Calculus of Constructions by inductive definitions of sorts and propositions. A sort is defined inductively by listing its constructors. Such a definition of an Inductive Set yields an induction principle and a Match-function, which enables the definition of (primitive recursive) functions by induction on the constructors. Together, they imply that every term of that sort is equal to a constructor term, and that all constructor terms are different. For example, the booleans can be defined in Coq as

```
Inductive Set bool = true : bool | false : bool.
```

Equality in Coq is a ternary polymorphic function `<_>_=_` (see below). It has a so-called dependent type: `(D:Set)D->D->Prop`. That is, for each sort D, `<_>_=_` is a function of type D->D->Prop. A simpler example of a dependent type is the type of the function `(D:Set) [D:D]d`, the polymorphic identity function (square brackets denote lambda-abstraction in Coq). Its type is `(D:Set)D->D`. In fact, the notation `P->Q` is an abbreviation of `(x:P)Q` when x does not occur in Q.

From the above inductive definition of bool, one can prove -(<bool>true=false) (true and false are not equal) and (b:bool) <bool>b=true /
<bool>b=false (for all b of type bool, b is either true or false). A disadvantage of inductively defined sorts is that the axioms that come with them remain hidden. This can result in a seemingly reasonable specification which is nevertheless incorrect, perhaps even inconsistent. For this reason and others, explained later, we shall not use this translation. It would certainly not be a good idea to define processes inductively, as there is no assumption in the semantics of μCRL that all processes can be built from the given actions and operators.

By the propositions-as-types paradigm, propositions can also be defined inductively. An inductively defined type is the least set that is closed under the constructors (such that all constructor terms differ); an inductively defined proposition is the least proposition that is closed under the rules given for it. Some basic notions in Coq, such as truth, falsity, and equality, are inductively defined.

```
Inductive Definition True : Prop = I: True.
Inductive Definition False : Prop = .
Syntax eq "<_>=_".
```

I is by this definition the proof of the nullary relation True; the induction principle for True is a tautology. False is the empty nullary relation; with this definition comes the induction principle False.ind: (P:Prop)False->P, the ex-falso rule, which reflects the minimality property for False. Finally, equality on a set A is defined through the statement 'for x:A, the unary relation "being equal to x" contains only x'. This definition gives the induction principle (A:Set)(x:A)(P:A->Prop)(P x)->(a:A)<A>x=a)->(P a).
Thus the effect of eliminating $\langle A \rangle b = a$ is that (usually all) occurrences of $a$ are replaced by $b$. Equations can be used as term rewrite rules from right to left in this way. Conjunction and disjunction are also inductively defined. Eliminating a conjunctive hypothesis $A \land B$ yields two hypotheses $A$ and $B$; eliminating $A \lor B$ yields two new proof obligations, one with hypothesis $A$ and one with $B$.

3 The Translation of $\mu$CRL into Coq

In this section, we discuss and motivate how we translate $\mu$CRL into Coq. We also outline the corresponding translation of $\mu$CRL-specifications.

3.1 $\mu$CRL versus Coq

$\mu$CRL and its proof theory share a significant number of concepts with Coq; we name (data)types, equality, implication, axioms, and derivability. The most formal way to proceed is to ignore these similarities, and to encode each $\mu$CRL-concept in Coq. That is, to define a sort $\mu$CRL_Prop of $\mu$CRL propositions and to encode $\mu$CRL-derivability inductively as the least relation $Dv : \mu$CRL_Prop$\to$Prop that contains all axioms and is closed under all inference rules of $\mu$CRL:

Inductive Definition $Dv : \mu$CRL_Prop$\to$Prop =

Reflexivity: $(D :$sorts $)$ $(\text{has_sort t D}) \to (Dv \ (\text{equal D t t}))$ |

Replacement: $(\Phi : \mu$CRL_Prop $(D :$sorts $)$ $(Dv \ (\text{subst D t x Phi})) \to (Dv \ (\text{equal D t u})) \to (Dv \ (\text{subst D u x Phi}))$ |

ACP_A1: $(p,q :$proc $)$ $(Dv \ (\text{equal proc (alt p q) (alt q p)}))$ |

Implication: $(\Phi,\Psi : \mu$CRL_Prop $)$ $(Dv \ (\text{Phi} \to (Dv \ (\text{Psi}))) \to (Dv \ (\text{implies Phi Psi}))$ |

In this example, equal encodes the equality predicate of $\mu$CRL, subst encodes substitution, sorts the declaration of sorts, has_sort the declaration of variables, alt the + on processes, implies implication between $\mu$CRL propositions, and so on.

Translating $\mu$CRL to Coq in this way is possible, but cumbersome: it gives rise to unreadable Coq texts and makes it impossible to automate the bulk of the proof. Namely, proofs in process algebra typically use a subset of the axioms (and derived equations) as a term rewriting system, computing normal forms for process terms (modulo associativity and commutativity of $\oplus$). Hand-written, such a part of the proof appears as $term = term = \ldots = term$; formally each step is an application of Replacement. In the above translation, the intermediate terms cannot be found by Coq; the user must provide them. This makes it effectively impossible to find even the most trivial proof automatically. In other words, with this translation we cannot hope to achieve a granularity of Coq proofs that comes anywhere near the granularity of hand-written proofs. Consequently, this approach is not (yet) scalable to real-life protocols.

Therefore we take another approach: rather than encoding $\mu$CRL in Coq, we embed $\mu$CRL in Coq, that is, we map $\mu$CRL-concepts to the 'same' concepts in Coq as much as possible. Such a translation renders Coq texts that are relatively easy to read, and intuitive proofs. The obvious problem with this approach is of course its soundness (and completeness). However, the soundness of the encoding approach is also not immediate, as it is not even proved yet that Coq is consistent [CP90, PM93], i.e., False might be derivable. In fact, the problem lies in the inductive sets and definitions, on which the encoding relies much more than our embedding approach. Clearly, any such soundness result lies beyond the scope of process algebra as long as the consistency of Coq is not proved.

\footnote{By eliminating $H$, we mean applying the induction principle for the main constructor of $H$.}
So the axioms of \( \mu \text{CRL} \) are translated to axioms in Coq; inference rules become implications (see Section 3.4 for the details). Also the rewrite rules of a \( \mu \text{CRL} \)-specification are translated to axioms. There are some obvious mismatches between Coq and \( \mu \text{CRL} \) to take care of. The most obvious mismatch occurs between the classical implication of \( \mu \text{CRL} \) and the constructive implication of Coq. In this case the rules of \( \mu \text{CRL} \) are stronger than those of Coq, so soundness is not at stake. We could have added the axiom 
\[
(P : \text{Prop}) \Rightarrow \text{P} \Rightarrow \text{P},
\]
but it turned out that we did not need it.

Another potential source of problems is equality. The equality \(<_\rightarrow=_\>_\_) of Coq has the Leibniz property, i.e., two terms are equal if and only if they can be substituted for each other in every context of type Prop. This is a strong requirement, as these contexts are built from the expressive language of Coq. Whether \( = \) in \( \mu \text{CRL} \) can be interpreted conservatively as Leibniz equality in Coq is a subject for specialized study.

Finally, \( \mu \text{CRL} \) has no explicit quantification, but instead the substitution rule (allowing the substitution of a term for a free variable, respecting sorts). This rule entails that all variables are implicitly universally quantified. These quantifiers are made explicit in our translation. Yet not all variables in \( \mu \text{CRL} \) are bound in this way: the sum operator \( \sum_{d:D}(x) \) binds the variable \( d \) of sort \( D \) in \( x \). We translate this binding to lambda abstraction, see Section 3.4 for the details.

### 3.2 Data

A significant part of the proof theory of \( \mu \text{CRL} \) can be translated to Coq independently of a particular \( \mu \text{CRL} \)-specification. Only the set of action names, the communication function \( \gamma \), and the set of sorts from a particular \( \mu \text{CRL} \)-specification parameterize this translation. The two sets are finite; therefore we define them as Inductive Sets\(^2\), simply enumerating their members. These are the only Inductive Sets we use. From these definitions it is easy to prove that all actions, respectively sorts, are different.

For simplicity, we allow actions to have precisely one data argument. For actions that have more than one parameter in the specification, pairing can be used. Actions without parameter get the dummy argument \( i \), which is the only element of the trivial sort \( \text{one} \). So we define

\[
\text{Inductive Set types} = \text{onetype:types} | \text{booltype:types} | \text{nattype:types} | ... \]

Why the sort \( \text{nat} \) of naturals is needed is explained in Section 3.7. In fact, this declaration gives us sort names. The sorts themselves are created through the declaration of a function \( \text{type : types->Set} \). (These definitions are part of the translation of \( \mu \text{CRL} \), similar definitions must be given for all other sorts in a specification.)

\[
\text{Parameter type : types->Set. } \quad \text{Definition bool} = (\text{type booltype}).
\]
\[
\text{Definition one } = (\text{type onetype}). \quad \text{Definition nat } = (\text{type nattype}).
\]

A consequence of this approach is that we cannot define these sorts inductively. Thus we must declare their constructors and induction principles explicitly. We can also not use the Match-function, therefore we must axiomatize the functions \( \text{zero} \) and \( \text{pred} \), which allow us to prove that naturals of the form \( S^n(0) \) differ for different \( n \).\(^3\)

\[
\text{Parameter i } : \text{one}. \quad \text{Parameter 0 } : \text{nat}.
\]
\[
\text{Parameter true,false} : \text{bool}. \quad \text{Parameter S} : \text{nat->nat}.
\]
\[
\text{Axiom I1} : (j : \text{one}) \Rightarrow \text{one} \rightarrow j = i.
\]
\[
\text{Axiom B1} : \Rightarrow \text{false}.
\]
\[
\text{Axiom B2} : (b : \text{bool}) \Rightarrow \text{false} \rightarrow \text{false} \land \Rightarrow \text{false} \rightarrow \text{false}.
\]
\[
\text{Axiom nat_ind} : (P : \text{nat->Prop})(n : \text{nat}) \Rightarrow (P 0) \Rightarrow ((y : \text{nat})(P y) \Rightarrow (P (S y))) \Rightarrow (P n).
\]
\[
\text{Parameter zero} : \text{nat->bool}. \quad \text{Parameter pred} : \text{nat->nat}.
\]

\(^2\) In Section 3.4 we explain why we cannot identify sorts from \( \mu \text{CRL} \) with sorts in Coq.

\(^3\) Alternatively, we could postulate a bijection between the sort \( \text{nat} \) as defined here and inductively defined naturals. Proofs using CFAR might be simplified by the resulting ability to use the Match-function.
Axiom zero0: \( \text{bool}(\text{zero} \ 0) = \text{true} \).
Axiom zeroS: \( \text{nat} \ (\text{zero} \ (\text{S} \ n)) = \text{false} \).
Axiom pred0: \( \text{nat} \ (\text{pred} \ 0) = 0 \).
Axiom predS: \( \text{nat} \ (\text{pred} \ (\text{S} \ n)) = n \).

If \( d \) and \( e \) have the same sort, then the equality \( d = e \) is a \( \mu \text{CRL} \) proposition. It is not a boolean term. If we need a function \( \text{eq}_d : D \rightarrow D \rightarrow \text{bool} \) for a sort \( D \), denoting equality, then we must define it in the specification. This will almost always be the case. It is indeed possible, although not straightforward, to define \( \text{eq}_d \) such that \( \text{eq}_d(d, e) = T \) iff \( d = e \). Instead of forcing the user to translate for each sort \( D \) the tricky definition of \( \text{eq}_d \) into Coq, we translate \( \text{eq}_d \) to the polymorphic function \( \text{eql} \), which we define by the axiom def_eql.

Parameter eql: \( (T:\text{types}) (\text{type} \ T) \rightarrow (\text{type} \ T) \rightarrow \text{bool} \).
Axiom def_eql: \( (T:\text{types}) (d, e: (\text{type} \ T)) \text{bool} (\text{eql} T d e) = \text{true} \leftrightarrow (\text{type} \ T) d = e \).

3.3 Actions and Communication

Actions in \( \mu \text{CRL} \) are declared with their respective sorts, but overloading of action names is allowed: one may declare an action \( r \) with sort \( D \) and another action \( r \) with a different sort \( E \). In the translation into Coq, actions are declared without their sorts (in other words: action names are declared). Thus there can be actions in the translation that are not present in the original specification. As these actions will not occur in the processes, this mismatch is harmless.

The communication function (\( \text{comm} \) in \( \mu \text{CRL} \), \( \gamma \) in \( \text{ACP}^\tau \)), is translated to the function \( \gamma \) in Coq. Communication in \( \mu \text{CRL} \) is defined on action names only, that is, if two actions (of different sort) have the same name, then they must communicate in the same way. This facilitates a correct translation into Coq: \( \gamma \) is specified only for the action name \( r \), not for \('r:D' \) and \('r:E' \) separately. It is not easy to specify partial functions in Coq, therefore when \( \gamma(a, b) \) is undefined, its translation \( (\gamma \ a \ b) \) returns the special action name \( \text{delta} \). The process \( r \) in \( \mu \text{CRL} \) behaves similarly to an atomic action, so a second special action name \( \tau \) is introduced.

\( \gamma \) must have certain properties, which are stated as five proof obligations (goals) in Coq. We must prove these goals in order to show that \( \gamma \) satisfies the desired properties. Some of these properties are also useful as lemmas when proving the correctness of protocols. The first two properties are that \( \text{delta} \) and \( \tau \) do not communicate. The third is that the communication of two actions is not \( r \) (allowing this would complicate defining guardedness, see Section 3.6). The fourth is that \( \gamma \) is commutative, as is required in [BW90]. It is also required there that \( \gamma \) is associative, but we assumed handshaking (communication between two components only), the fifth property, which is stronger.

Goal \( (a: \text{act}) \ (\gamma \delta \ a) = \delta \).
Goal \( (a: \text{act}) \ (\gamma \tau \ a) = \tau \).
Goal \( (a, b: \text{act}) \ (\gamma \ a \ b) = (\gamma \ b \ a) \).
Goal \( (a, b, c: \text{act}) \ (\gamma \ a \ (\gamma \ b \ c)) = \delta \).

Proving these goals is automatic: the text of the proofs does not depend on \( \gamma \): it is always a straightforward case analysis (thanks to the fact that actions are defined inductively, and \( \gamma \) is defined by the \( \text{Match} \) function.

3.4 Processes and Axioms

The distinction between the action \( a \) and the process \( a \) is not always obvious in process algebra. In the current setting, it is obvious that a process is formed from an action name, its sort, and an element of that sort. However, there is only one process \( \delta \) and one process \( r \). Thus we declare
Parameter proc : Set. (* the type of processes *)
Parameter ia : (T:types) act->(type T)->proc. (* interpret an action and *)

Definition Delta = (ia onetype delta i).
Definition Tau = (ia onetype tau i).

Axiom Delta_Data : (T:types)(t:(type T)) <proc>Delta=(ia T delta t).
Axiom Tau_Data : (T:types)(t:(type T)) <proc>Tau=(ia T tau t).

It remains to model sets of actions (for hiding and encapsulation), before we declare the operators on processes. We model such sets by their characteristic function act->Prop. A small complication is that we have added delta and tau to the set of actions, and that these cannot be encapsulated, nor hidden. Thus we define the function goodset, which, given a set of actions, returns the same set without delta and tau.

Definition goodset ehset = act->Prop. (* encapsulation and hiding sets *)
Definition goodset ehset->ehset = [L:ehset]
[a:act] ((<act>a=delta))/\((<act>a=tau))/\(L a).

Parameter alt, seq, mer, Lmer, comm : proc->proc->proc.
Parameter cond : proc->bool->proc->proc.
Parameter sum : (T:types) ((type T)->proc)->proc.

These operators denote the following ones in μCRL respectively: +, ||, , < . , , Σ, δ, and τ. Note the type of the sum operator. \( \sum_{t \in T}(x) \) is translated to \( \text{sum } T \{ [d:(type T)]x \} \), thus sum has the polymorphic type \( (T:\text{types})((type T)->\text{proc})->\text{proc} \). Now it is clear why we cannot identify μCRL-sorts with sorts in Coq: proc could then be used as a μCRL-sort. This would allow for example the process definition \( P = (a \triangleleft P=\delta \triangleright \delta) \), which implies \( a = \delta \), and \( \sum_x \text{proc } x \), the sum of all processes.

Most of the axioms of μCRL translate directly into Coq, as they are simply equations between processes; variables are universally quantified. For example, the axiom A1, \( x + y = y + x \), translates to

Axiom A1: (x,y:proc)<proc>(alt x y)=(alt y x).

Table 1: The communication function axioms of μCRL.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Equation</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF1</td>
<td>( a(t_1, \ldots, t_m) \mid b(t_1, \ldots, t_m) = c(t_1, \ldots, t_m) ) if ( \gamma(a,b) = c, m \geq 0 ), ( \delta ) if ( \gamma(a,b) ) is undefined.</td>
<td>( i \leq m ), in particular, if ( a ) or ( b ) is ( \delta ) or ( \tau ),</td>
</tr>
<tr>
<td>CF2</td>
<td>( a(t_1, \ldots, t_m) \mid b(t'_1, \ldots, t'_m) = \delta )</td>
<td>( \delta ) if ( a ) and ( b ) have different sorts, in particular, if ( m \neq m' ).</td>
</tr>
<tr>
<td>CF2'</td>
<td>( a(t_1, \ldots, t_m) \mid b(t'_1, \ldots, t'_m) = \delta )</td>
<td>( 1 \leq i \leq m ),</td>
</tr>
<tr>
<td>CF2''</td>
<td>( a(t_1, \ldots, t_m) \mid b(t'_1, \ldots, t'_m) = \delta )</td>
<td></td>
</tr>
</tbody>
</table>

Only the translation of the axioms regarding the communication function, listed in Table 1, is not so straightforward. We translate them to

Axiom CF1: \(<\text{proc}>(\text{comm } (\text{ia } T \text{ a t}) \text{ (ia T b t'))})=\text{(cond } (\text{ia } T \text{ (gamma a b) t}) \text{ (eql T t t') Delta}) \).
Axiom CF2: "<types>T=U -> <\text{proc}>(\text{comm } (\text{ia } T \text{ a t}) \text{ (ia U b u'))}=\text{Delta} \).
CF1 covers not only the case of actual communication (CF1 in Table I), but also the case where communication fails because the actions do not communicate or the data is not the same (CF2 and CF2'). The axiom deLeql justifies this formulation, which effectively replaces the premiss \( \neg(t_i = t'_i) \) of CF2 by \( eq_T(t_i, t'_i) = F \). The only remaining case is that of CF2": actions with different sorts (and hence incomparable data), which is covered by CF2.

| SUM1 | \( \sum_{d \in D} p = p \) if \( d \) not free in \( p \) |
| SUM2 | \( \sum_{d \in D} p = \sum_{e \in D} (p[e/d]) \) if \( e \) not free in \( p \) |
| \( \ldots \) |
| SUM11 | \( \sum_{d \in D} p_1 = \sum_{d \in D} p_2 \) |

Table 2: Some axioms for summation in \( \mu\text{CRL} \).

The axiom SUM2 of \( \mu\text{CRL} \) is recognised as \( \alpha \)-conversion, and can therefore be omitted in the translation. The requirement that \( d \) is not free in \( p \) in SUM1 (and other axioms not listed) is verified automatically: if it is not satisfied, then an unbound variable occurs. The premiss of SUM11 refers to the equality of two processes with a free variable \( d : D \); it is translated to \( \forall d \in D : p_1(d) = p_2(d) \).

### 3.5 Recursive Specifications and RDP

Informally, RDP states that a recursive specification has at least one solution. Thus we need to translate what is a recursive specification, and what is a solution of it. First, we consider the case of a single recursive equation. Such an equation, written as \( X(d) = G(X, d) \), can be seen as the definition of the process operator \( G \) of type \((D\rightarrow\text{proc})\rightarrow(D\rightarrow\text{proc})\). (This is a generalization of the linear process operators of [BG94b], where \( G \) must be in a particular normal form.) A solution of the recursive equation is then a fixed point of \( G \), and has type \( D\rightarrow\text{proc} \).

In the general case, we have a set of process variables ProcVar and a function Typ from ProcVar to types giving their associated sorts (similar to actions, we let process variables have exactly one data parameter). A solution of a system of recursive equations is now a function that interprets each process variable as a function from its data parameter to a process, thus the type of a solution (in fact, of any such interpretation) is \( \text{Inttype} = (X: \text{ProcVar}) (\text{type} (\text{Typ} X) \rightarrow \text{proc} \). The system of recursive equations DefEq itself is then a process operator \( \text{Inttype} \rightarrow \text{Inttype} \) (similar to \( G \) above). The solution is its fixed point.

For example, the system \( \{ X = a \cdot Y(T) ; \ Y(b : \text{Bool}) = X + a \cdot Y(\text{not}(b)) \} \) is defined as follows (note that DefEq needs the "old" interpretation of process variables \( \text{ipV} \) to interpret the occurrence of a process variable in the body of an equation as a process).

**Inductive Definition** ProcVar = \( X : \text{ProcVar} \mid Y : \text{ProcVar} \).

**Definition** Typ = \( [P : \text{ProcVar}] (\text{type}(\text{Typ} P)) \rightarrow \text{proc} \).

**Definition** Inttype = \( (P : \text{ProcVar}) (\text{type}(\text{Typ} P)) \rightarrow \text{proc} \).

**Definition** DefEq = \( [\text{ipV} : \text{Inttype}] (P : \text{ProcVar}) (\text{type}(\text{Typ} P)) \rightarrow \text{proc} \)

\( \text{Match P with} \)
\( \text{(*) X (*) [j:one] (seq (ia onetype a i) (ipV Y true))} \)
\( \text{(*) Y (*) [b:bool] (alt (ipV X i) (seq (ia onetype a i) (ipV Y (neg b))))} \).

RDP states that a system of recursive equations has a solution, i.e., that a process operator has a fixed point. Thus we declare the solution function Sol: \( (\text{Inttype} \rightarrow \text{Inttype}) \rightarrow \text{Inttype} \) giving a solution for each system of equations (think of it as the least fixpoint operator). That (Sol DefEq) is indeed a solution for DefEq is stated in axiom RDP.
Section RDP.

Variable ProcVar : Set.

Variable Typ : ProcVar->types.

Local Inttype = (X: ProcVar)(type (Typ X))->proc.

Variable DefEq : Inttype->Inttype.

Parameter Sol : (Inttype->Inttype)->Inttype.

Axiom RDP : <Inttype>(Sol DefEq)=(DefEq (Sol DefEq)).

End RDP.

The process $X$ defined by the above system is now referred to as $\text{Sol ProcVar Typ DefEq X}$. By RDP, we can prove that this term is equal to $(\text{seq (ia onetype a i) (Sol ProcVar Typ DefEq Y true)})$.

3.6 RSP

RSP states that guarded systems of equations have unique solutions. So we must define guardedness in Coq. A single recursive equation is guarded if we can determine for all $n$ the first $n$ visible actions of its solution by repeatedly unfolding the equation. For example, if we have $X(b: \text{Bool}) = (\text{tau a}) \cdot X(\text{not}(b))$, then $X(T) = \tau \cdot X(F) = \tau \cdot a \cdot X(T)$, so we can determine the first visible action $(a)$ of $X(T)$ by unfolding the equation twice. Further applications of the equation give us further visible actions: the equation is guarded.

In contrast, if we have $Y = a \cdot Z$, then this equation gives us the first visible action, but a second unfolding yields $Y = a \cdot \tau(a \cdot \tau(X(Y)) = a \cdot \tau(a)(Y)) = a \cdot \tau(a)(Y)$. Clearly, further unfoldings do not yield further visible actions for $Y$, so this equation is unguarded. Indeed, both $a$ and $a \cdot \tau$ are solutions for this equation, thus RSP should not be applicable. In view of this second example, we will simply consider every recursive equation in which the hiding operator $\tau$ occurs unguarded (unless of course we can remove the hiding operator by rewriting the system using the axioms).

Now we return to the first example. We note that when we unfold $X(T)$, we obtain $X(F)$ without a visible action (guard) in front. We say that $X(T)$ depends unguarded on $X(F)$. On the other hand, unfolding $X(F)$ yields $X(T)$ only behind a guard, so $X(F)$ does not depend unguarded on $X(T)$. We can have the same notion in a system of equations: if we replace $X(T)$ by $Y$ and $X(F)$ by $Z$ then we obtain the system $\{Z = \tau \cdot Y, Y = a \cdot Z\}$ in which $Z$ depends unguarded on $Y$, but $Y$ does not depend unguarded on $Z$.

We conclude that 'depends unguarded on' is a binary relation $R$ on pairs of the form $(X, e)$, where $X$ is a process variable and $e$ is data of the correct sort for $X$. $R$ must be well-founded for the system to be guarded. Rather than writing an axiomatization that tries to compute $R$, we let the user provide $R$. Then we check that $R$ is well-founded (see also [BG94c]) and that for all process variables $X$ and data $e$ of the sort for $X$, the body of the equation for $X(e)$ is safe, that is, if $Y(f)$ occurs in this body, either it occurs behind a guard, or $R(X, e, Y, f)$ holds. What follows is the translation of this into Coq; the details are explained thereafter.

Parameter Safe : (ProcVar:Set)

(ProcVar->types)
(1PV: (X:ProcVar)(type (Typ X))->proc)
(X:ProcVar)
(e:(type (Typ X)))->
(X:ProcVar)(type (Typ X))->(Y:ProcVar)(type (Typ Y))->Prop)->
proc->Prop.

---

4 Allowing $\gamma(a, b) = \tau$ would give similar problems for $| |$ and $||$, consider e.g., $Z = a \cdot (\tau | Z)$.

5 The meaning of the word 'occurs' in this section is semantical; the hiding operator does not 'occur' in $\delta \cdot \tau f(X)$, as this term is equal to $\delta$. Because we have Leibniz-equality, we cannot even use a syntactical notion of occurrence.

6 Apart from cyclic ones, this also excludes unfounded specifications like $X(n: \text{nat}) = X(S(n))$. 

113
Section RSP.

Variable ProcVar : Set.
Variable Typ : ProcVar -> types.
Local typ = [X:ProcVar](type (Typ X)).
Local Inttype = (X:ProcVar)(type X)->proc.
Local RT = (X:ProcVar)(type X) -> (Y:ProcVar)(type Y) -> Prop.
Variable iPV : Inttype.
Variable DefEq : Inttype->Inttype.
Variable X : ProcVar.
Variable e : (type X).
Variable R : RT.

Local RSafe : proc->Prop = (Safe ProcVar Typ iPV X e R).
Local TSafe : proc->Prop = (Safe ProcVar Typ iPV X e R)
[X:ProcVar][e:(typ X)][Y:ProcVar][f:(typ Y)]True).

Variable x,y : proc.
Variable T : types.

Axiom S0: (Y:ProcVar)(f:(typ Y)) (R X e Y f) -> (RSafe (iPV Y f) ).
Axiom S1: ((a:act)(t:(type T)) (RSafe x) -> (RSafe y) -> (RSafe (seq x y)).
Axiom S2: ((a:act)(t:(type T)) (RSafe x) -> (RSafe y) -> (RSafe (alt x y)).
Axiom S3: ((a:act)(t:(type T)) (RSafe x) -> (RSafe y) -> (RSafe (mer x y)).
Axiom S4: ((a:act)(t:(type T)) (RSafe x) -> (RSafe y) -> (RSafe (comm x y)).
Axiom S5: ((a:act)(t:(type T)) (RSafe x) -> (RSafe y) -> (RSafe (sum T p)).
Axiom S6: ((a:act)(t:(type T)) (RSafe x) -> (RSafe y) -> (RSafe (enc L x)).

Variable ProcVar' : Set.
Variable Typ' : ProcVar' -> types.
Local typ' = [X':ProcVar'](type (Typ' X')).
Local Inttype' = (X':ProcVar')(type X') ->proc.

Local TSafe' = [iPV':Inttype']
(Safe ProcVar' Typ' iPV' X' e'
[X':ProcVar'][e':(typ X')][Y':ProcVar'][f':(typ Y')])True).

Axiom S10: (DefEq' :Inttype'->Inttype')(X':ProcVar')(e':(typ X'))
((iPV':Inttype')(X':ProcVar')(d':(typ X'))
(TSafe' iPV' (DefEq' iPV' X' d')) )->
(RSafe (Sol ProcVar' Typ' DefEq' X' e')).

S0 states that Y(f) can occur unguarded in the defining equation of X(e), provided R(X, e, Y, f) holds.
S2 states that all process variables may occur after a guard; the effect is obtained by replacing R by the relation that is always true. The premiss (TSafe y) serves to check that no hiding operator occurs in y.
S10 states that the system may refer to another system of equations. This other system must be proved safe w.r.t. the relation that is always true, i.e. it must not contain hiding and, more importantly, it must not contain variables of the current system (technically: the defining equations DefEq' of this new system must not depend on the iPV of the current one). For example, following the notation of [BW90], we could have E = \{X = a \cdot (X' | E_X')\}, with E_X' = \{X' = X + b \cdot X'\}. Notice that in μCRL we cannot distinguish

\footnote{We need not prove that this other system is guarded! If it is not, then it will not have a unique solution, but the unique solution of the current system will contain the (not uniquely determined) term (Sol ProcVar' Typ' DefEq' X' e').}
this combination from the flattened system \( \{ X = a \cdot X', \ X' = X + b \cdot X' \} \), but that we need the distinction to modularize proofs.

One can observe that the above combination of \( E \) and \( E_X' \) is in fact safe, because the flattened system is. Indeed, we can allow the stronger variant of axiom \( S10 \) below, which allows the occurrence of those variables \( Y(f) \) that were allowed to occur unguarded anyway in the equation for \( X(e) \), because \( R(X,e,Y,f) \) holds. It does however not change \( R \) to the relation that is always true after encountering a guard. Anyway, we do not need this stronger version of \( S10 \) if we only build one system on top of the other, instead of mutually recursive ones.

\[
\text{Axiom } S10: (\text{DefEq'}:\text{Inttype'}->\text{Inttype'})(X':\text{ProcVar'})(e':(\text{typ'} X'))
\]

\[
( (iPV':\text{Inttype'})(X':\text{ProcVar'})(d':(\text{typ'} X'))
\]

\[
( (Y:\text{ProcVar})(f:(\text{typ Y})) (R X e Y f)->(\text{T Safe'} iPV' (iPV Y f)) )->
\]

\[
(\text{T Safe'} iPV' (\text{DefEq'} iPV' X' d')) )->
\]

\[
(\text{RSafe (Sol ProcVar Typ DefEq'} X' e')).
\]

Finally, we can state the axiom \( RSP \). Given are an interpretation of process variables \( iPV \), the system of equations \( \text{DefEq} \) and the relation \( R \). The system is guarded if \( R \) is well-founded and all bodies are safe (for no \( X \) and \( d \), there is an infinite descending chain from \( X \) and \( d \), and the body of the equation for \( X \) and \( d \) is safe). If the system is guarded and \( iPV \) is indeed a solution\(^{6}\), then \( iPV \) equals the canonical solution \( \text{(Sol ProcVar Typ DefEq)} \) of the system.

\[
(\text{Inductive Definition } WF : (X:ProcVar)(\text{typ X})->\text{Prop} =
\]

\[
WF1: (X:ProcVar)(d:(\text{typ X}))
\]

\[
((Y:ProcVar)(e:(\text{typ Y}))(R X d Y e)->(WF Y e)) \to (WF X d).
\]

\[
(\text{Definition Guarded } = (X:ProcVar)(d:(\text{typ X}))(iPV:\text{Inttype})
\]

\[
(WF X d) \land (\text{Safe ProcVar Typ iPV X d R (DefEq iPV X d)}).
\]

\[
(\text{Axiom } RSP: \text{Guarded}->
\]

\[
((X:ProcVar)(d:(\text{typ X}))<\text{proc}>(iPV X d) = (\text{DefEq iPV X d}))->
\]

\[
<\text{Inttype} iPV=(\text{Sol ProcVar Typ DefEq}).
\]

\[
\text{End } RSP.
\]

### 3.7 Fair Abstraction

A fairness assumption is often necessary for proving a protocol correct. Such assumptions were translated into process algebra in various ways, most notably in the form of fair abstraction rules. For an overview we refer to Section 5.6 of [BW90]. We chose to translate CFAR\(^{6}\) into Coq (Cluster Fair Abstraction Rule for branching bisimulation, we omit the superscript \( b \) further on). Informally, a cluster is a (maximal) set of states of a process such that each state in it can reach each other in it by taking only hidden steps.

We have adapted CFAR to the presence of data as follows. Instead of a single cluster, we like to coHaps a number of clusters at the same time. For example, if we have a process definition

\[
X(n: \text{nat}) = b(n) + i \cdot (X(n + 9) \oplus (n \mod 10) = 0 \oplus X(n - 1)),
\]

then we want to infer

\[
X(n: \text{nat}) = b(n) + i \cdot (X(n + 9) \oplus (n \mod 10) = 0 \oplus X(n - 1)),
\]

for all \( n: \text{nat} \): \( r \cdot \tau_{\{i\}}[X(n)] = r \cdot \tau_{\{i\}}[X(n + 9) \oplus \ldots \oplus X(n + 9)] \).

\[
\text{There are infinitely many clusters, therefore we cannot collaps each cluster separately. One way to proceed would be to fix a } k: \text{nat} \text{ and to define}
\]

\[
Y_k(m: [0..9]) = b(10k + m) + i \cdot (Y_k(9) \oplus m = 0 \oplus Y_k(m - 1)).
\]

\(^{6}\)That \( iPV \) is a solution must be written as \((X:ProcVar)(d:(\text{typ X}))<\text{proc}>(iPV X d)=(\text{DefEq iPV X d})\), rather than \(<\text{Inttype} iPV=(\text{DefEq iPV})\), because the latter equality does not follow from the former in Coq.
Then we prove by CFAR

\[ \tau \cdot \tau_{\tau_1}(Y_k(m)) = \tau \cdot (b(k) + \ldots + b(k + 9)). \]

Finally we prove by RSP \( X(n) = Y_{n \mod 10}. \) We cannot formalize this approach in \( \mu \text{CRL}, \) because there \( k \) should be a formal parameter of \( Y, \) leaving us with many clusters again. However, our translation of recursive specifications into Coq does not prevent parameterized specifications such as the one of \( Y_k: \) we can encode this approach in Coq, albeit clumsily (we must add a new datatype with ten elements and a function interpreting them as \( 0..9 \)).

Therefore we chose a formulation of CFAR that collapses multiple clusters explicitly. First we number the different clusters. Then we number the different pairs \((X, d)\) within each cluster, where \( X \) is a process variable and \( d \) a data parameter of the type of \( X.\) That is, we assume having the following functions.

- \( \text{cluster}(X, d) \) gives the number of the cluster to which the pair \((X, d)\) belongs.
- \( \text{element}(X, d) \) gives the order number of \((X, d)\) within its cluster.
- \( \text{process}(n, m) \) \((n, m \in \text{nat})\) returns \( X(d) \) such that \( \text{cluster}(X, d) = n \) and \( \text{element}(X, d) = m. \) It returns \( \delta \) if \( n \geq \) the number of clusters or \( m \geq \) the number of processes in the cluster.
- \( \text{Exit}(n, m) \) \((n, m \in \text{nat})\) returns the exit process of the \( m \)th item in the \( n \)th cluster. Again it is \( \delta \) if \( n \) or \( m \) are too large.
- \( a(X, d, m) \) is the action (including data) that leads from \( X(d) \) to the \( m \)th item in the cluster of \( X(d). \) It is \( \delta \) if there is no such action.

In our translation into Coq, the user must provide these functions for each application of CFAR, and show that they have the following properties (let \( L \) be the set of actions going to be hidden).

1. For all \( X \) and \( d: \) \( X(d) = \text{process}(\text{cluster}(X, d), \text{element}(X, d)). \)
2. For all \( n \) and \( m: \) if for no \( X \) and \( d: \) \((\text{cluster}(X, d), \text{element}(X, d))\),
   then \( \text{Exit}(n, m) = \text{process}(n, m) = \delta. \)
3. The system of equations can be written in the form
   \[ X(d) = \sum_{m:\text{nat}} a(X, d, m) \cdot \text{process}(\text{cluster}(X, d), m) + \text{Exit}(\text{cluster}(X, d), \text{element}(X, d)). \]
4. Each \( a(X, d, m) \) is either \( \delta, \tau, \) or its action name is in \( L.\)
5. All clusters are connected: we can step from \( X(d) \) to \( Y(e) \) iff \( a(X, d, \text{element}(Y, e)) \neq \delta; \) a cluster is connected if for all \( X(d) \) and \( Y(e) \) in it, we can go from \( X(d) \) to \( Y(e) \) in one or more steps.
6. The system is guarded.

Given definitions satisfying these properties, CFAR concludes for all \( X \) and \( d: \)

\[ \tau \cdot \tau_L(X(d)) = \tau \cdot \tau_L(\sum_{m:\text{nat}} \text{Exit}(\text{cluster}(X, d), m)). \]

In our example, we could use the following functions.

\[
\begin{align*}
\text{cluster}(X, n) &= n \text{ div } 10 \\
\text{element}(X, n) &= n \mod 10 \\
\text{process}(k, m) &= X(10k + m) \text{ if } m \leq 9, \quad \delta \text{ otherwise} \\
\text{Exit}(k, m) &= b(10k + m) \text{ if } m \leq 9, \quad \delta \text{ otherwise} \\
a(X, n, m) &= i \text{ if } m = (n - 1) \mod 10, \quad \delta \text{ otherwise.}
\end{align*}
\]

\[9\] Here we see a summation over the natural numbers. Since we have only summation over sorts, we need \textit{nat} as a built-in sort.
We finally provide the representation of CFAR in Coq. Notice that process needs an interpretation of process variables, and that the definition of a(X,d,m) is split in three parts: sort, action name, and data. The definition of conn is optimized by putting Z and f first: these variables must be provided in a proof, the others are inferred from the goal.

Section CFAR.
Variable ProcVar : Set.
Variable Typ : ProcVar -> types.
Local typ = [X:ProcVar](type (Typ X)).
Local Inttype = (X:ProcVar)(typ X)->proc.
Variable DefEq : Inttype->Inttype.
Variable L : ehset.
Variable cluster : (X:ProcVar)(typ X) -> nat.
Variable element : (X:ProcVar)(typ X) -> nat -> act.
Variable Exit : nat -> nat -> proc.
Variable D' : (X:ProcVar)(typ X) -> nat -> types.
Variable a : (X:ProcVar)(typ X) -> nat -> a.
Variable d' : (X:ProcVar)(d:(typ X))(n:nat) (type (D' X d n)).

Definition CheckInside = (X:ProcVar)(d:(typ X))(iPV:Inttype)
(*1*) <proc>(process iPV (cluster X d) (element X d)) = (iPV X d).

Definition CheckOutside = (n,m:nat)(iPV:Inttype)
(*2*) ((X:ProcVar)(d:(typ X)) "(<nat>n=(cluster X d) /
<nat>m=(element X d) ) ) ->
<proc>(process iPV n m)Delta /
<proc>(Exit n m)Delta.

Definition CheckDef = (X:ProcVar)(d:(typ X))(iPV:Inttype)
(*3*) <proc>(DefEq iPV X d);
(alt (sum nattype [n:nat](seq (ia (D' X d n) (a X d n) (d' X d n))
(process iPV (cluster X d) n)))
(Exit (cluster X d) (element X d))).

Definition Checka = (X:ProcVar)(d:(typ X))(n:nat)
(*4*) <act>(a X d n)=delta / <act>(a X d n)=tau / (goodset L (a X d n)).

Inductive Definition Conn : (X,Y:ProcVar)(typ X)->(typ Y)->Prop
conn1: (X,Y:ProcVar)(d:(typ X))(e:(typ Y))
"<act>(a X d (element Y e))=delta -> (Conn X Y d e)
(Conn X Z d f) -> (Conn Z Y f e) " -> (Conn X Y d e).

Definition CheckConn = (X,Y:ProcVar)(d:(typ X))(e:(typ Y))
(*5*) <nat>(cluster X d) (cluster Y e) -> (Conn X Y d e).

Axiom CFAR: (X:ProcVar)(d:(typ X))
CheckInside -> CheckOutside -> CheckDef -> Checka -> CheckConn ->
(*6*) (Guarded ProcVar Typ DefEq R) ->
<proc>(seq Tau (hide L (Sol ProcVar Typ DefEq X d))=
(seq Tau (hide L (sum nattype [n:nat](Exit (cluster X d) n)))).

End CFAR.
4 Conclusions and Future Work

We have presented a translation from $\mu$CRL to Coq. The aim of this translation is to allow formal algebraic verification of protocols, formulated in $\mu$CRL, to be carried out in Coq. By verifying the Alternating Bit Protocol on the basis of this translation, we have shown in [BBG95] that such a verification is at least possible for small protocols. As a side-effect, we began to develop a library of lemmas of general use, such as lemmas about the standard sorts, ‘derived axioms’ for ACP’, and expansion theorems.

In order to verify larger protocols, we must extend this library of lemmas, and improve proof techniques at the level of Coq as well as at the level of $\mu$CRL, see e.g. [BG94b]. Work in this direction was reported in [BG94a, KS93, GP93a]. Some protocols involve aspects that cannot be modeled in $\mu$CRL, in particular the aspect of time. It appears worthwhile to extend $\mu$CRL with (discrete) real time [BB92] and by translating the resulting formalism to Coq arriving at formal verification of timed protocols [KP93, Klu91].

Proving the soundness of our translation w.r.t. $\mu$CRL is a moving target, as changes to Coq are still being made (version 5.10 recently appeared), and changes to $\mu$CRL are proposed, e.g. in [GW94]. Moreover, it requires the consistency of Coq, a result which is outside the scope of process algebra. The same problem arises for any other proof checker or theorem prover we might use. Still, other tools must be evaluated to see if they allow us to verify protocols more easily; algebraic verifications consist for a significant part of term rewriting, which is not easy to automate in Coq. We wish to encourage other groups to join us in this effort, as getting acquainted with a tool to the extent that one can translate a sizeable theory like $\mu$CRL into it requires a significant investment of time.

Acknowledgments
We thank Jaco van de Pol, Jan Springintveld, Alex Sellink, Erik Poll, Jos Baeten, and Jan Bergstra for some valuable discussions.

References


Two Finite Specifications of a Queue

Marc Bezem\(^1\) Alban Ponse\(^2\)

\(^1\)Utrecht University, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

\(^2\)University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

E-mail: bezem@phil.ruu.nl alban@fwi.uva.nl

Abstract. Two finite specifications of a queue in ACP with abstraction are proven correct relative to a standard specification of a queue that employs an infinite data type for representing its contents. The proofs are given in the proof theory of I'CLRL, and the only 'r-laws' used are \(xr = x\) and \(x(r(y+z)+y) = x(y+z)\). Therefore the proofs are adequate for both 'branching bisimilarity' and 'observation equivalence'. Additionally, it is shown that standard concurrency follows from RSP for a class of processes guardedly specifiable in ACP with abstraction.

Key words & Phrases: process algebra, queue, guardedness.


1 Introduction

The purpose of this paper is to record correctness proofs of two finite specifications of a queue, introduced below. Both these specifications are already known for some time. However, of the first one no proof has been published yet (as far as we are aware), and only slight variants of the second one were proven correct. Furthermore, the specifications and proofs are given in the proof theory of \(\mu\)CRL [GP90, GP91, GP94], and can in that form be used as samples for proof checking.

Additionally, the paper offers a small theoretical result on RSP (Recursive Specification Principle), a fixed point rule mostly adopted in proofs on recursively specified processes in the setting of ACP with abstraction (see [BK85, BW90] for a general introduction to ACP with abstraction and RSP). This result states that RSP implies the standard concurrency identities for processes specified by linear equations (and therewith commutativity and associativity of \(\parallel\)). This is remarkable, because axioms for standard concurrency are often explicitly adopted in correctness proofs that already use RSP.

Queues and correctness. Proving that some specification indeed defines a queue presupposes a standard definition of a queue. Correctness then boils down to proving equality with this standard definition using specific axioms and rules. Given that a queue is a process that receives data of type \(D\) along some in-port, and can send data in the same order in which they were received along some out-port (FIFO-like, i.e., First In, First Out), we use sequences of \(D\)-elements, typed Sequence, to represent the contents of a queue at any moment in its execution. For a queue with in-port \(i\) and out-port \(j\), this standard specification is given in \(\mu\)CRL by the following recursion equation (cf.

121
\[ Q^{ij}(q : \text{Sequence}) = \sum_{d \in D}(r_i(d) \cdot Q^{ij}(d.q)) + s_j(\text{right-elt}(q)) \cdot Q^{ij}(\text{left-rm}(q)) \triangleq \text{non-empty}(q) \triangleright \delta \]

where: \( . \) in \( d.q \) is the function that inserts a new \( D \) element at the left of a sequence;

\( P \triangleq b \triangleright Q \) abbreviates if \( b \) then \( P \) else \( Q \) (notation from [HHJ+87]);

\( \text{non-empty}(q) \) is the Boolean expressing whether \( q \) does not equal \( e \), the empty sequence;

\( \text{right-elt} \) extracts the right element of a sequence, e.g. \( \text{right-elt}(d.(e.f)) = e \);

\( \text{left-rm} \) extracts the remainder of a \( \text{Sequence} \) element, e.g. \( \text{left-rm}(d.(e.f)) = d.e \).

The functions \( \text{left-rm} \) and \( \text{right-elt} \) are by definition total: \( \text{left-rm}(e) = E \) and \( \text{right-elt}(E) = d.e \).

The parameter \( q \) in \( Q^{ij}(q) \) represents the execution state of the queue. For example, one can depict \( Q_{11}(d.(e.f)) \), a state that can be reached by having received first \( e \) and then \( d \) along port \( i \) (actions \( r_i(e), r_i(d) \)), as follows:

Now \( Q^{ij}(d.(e.f)) \) can either receive some new datum \( e' \) along port \( i \) (action \( r_i(e') \)) and evolve into \( Q^{ij}(e'.(d.(e.f))) \), or can send \( e \) along port \( j \) (action \( s_j(e) \)) and evolve into \( Q^{ij}(d.f) \). In Section 3, we provide a detailed specification of the data type \( \text{Sequence} \) in \( \mu\text{CRL} \) style.

Starting point for this paper is to adopt the process \( Q^{ij}(e) \) as the standard definition of the (empty) queue with in-port \( i \) and out-port \( j \).

**Two finite queue specifications in ACP with abstraction.** Staying in the realm of ACP with abstraction, one can specify the queue above by using only the process operations of \( ACP_\tau(A, \gamma) \), instead of basing it on data type specification and data/process interaction. Two such specifications of a queue will be proven correct, i.e., equal to \( Q^{ij}(e) \). The only '\( \tau \)-laws' used in this proof are

\[ \begin{align*}
\text{(B1)} & \quad x\tau = x \\
\text{(B2)} & \quad x(\tau(y + z) + y) = x(y + z)
\end{align*} \]

which were defined in the setting of 'branching bisimulation' [GW89, GW89a]. (These laws are implied by the \( \tau \)-laws for 'observation equivalence' [Mil89].) In the following we introduce two finite queue specifications.

In the paper [BK86a], Bergstra and Klop presented the following intricate \( ACP_\tau(A, \gamma) \) specification of a queue using six recursion equations and one auxiliary (hidden) port:

\[ Q^{ij} = \sum_{d \in D}(r_i(d) \cdot (Q^{ik} || s_j(d) \cdot Q^{kj})) \text{ for all } \{i, j, k\} = \{1, 2, 3\}, \]

where: \( Q^{ij} \) represents a queue with in-port \( i \) and out-port \( j \);

\( z || y \) is short for \( \tau_{(c_4(d) | d \in D)} \circ \delta_{(t_4(d), s_4(d) | d \in D)}(x || y) \);

\( r_i(d)|s_i(d) = c_i(d) \) for \( i \in \{1, 2, 3\} \) and \( d \in D \) (and no other communications are defined).
Result 1.1. In $\mu$CRL proof theory with (B1) and (B2), $Q^{ij} = Q^{ij}(e)$ for $i,j \in \{1,2,3\}, i \neq j$.

In the same style as the specification of $Q^{ij}$ above, one can specify a queue with in-port 1 and out-port $j$ by using only four equations, employing an auxiliary port $i$ and linked one element buffers. The one element buffer $B^{ij}$ is a process that can receive some datum $d$ along port $i$, after which it can only send $d$ along port $j$, after which this behaviour is repeated. This specification is the following:

$$
QB^{1j} = \sum_{d \in D} (r_i(d) \cdot (QB^{1i} || s_j(d) \cdot B^{ij}))
$$

for all $\{i,j\} = \{2,3\}$,

with the communications and $||$ as defined above.

In [Hoa85], Hoare already proved that queues can be specified as 'linked' one element buffers. Similar results were proved by Van Glabbeek and Vaandrager [GV89], and Brinksma and Kars [BK91]. Most close to pure ACP $\tau(A, \gamma)$ is the specification from [GV89] that employs chaining operators $\gg$ and $\gg\gg$.

Result 1.2. In $\mu$CRL proof theory with (B1) and (B2), $QB^{1j} = Q^{1j}(e)$ for $j \in \{2,3\}$.

Note on RSP. In the set-up of $\mu$CRL, i.e. processes that may interact with data, RSP applications are based on systems of equations that are guarded and that go with a substitution mechanism that allows data modification (see Appendix). In this respect, the $\mu$CRL version of RSP differs from the usual ACP version and all such applications are therefore displayed in a very detailed way.

Plan of this paper. In the next section we prove that the two finite queue specifications introduced above are guarded, by which each identifier $Q^{ij}$, $QB^{1j}$ uniquely determines a process. In Section 3, a proof of Result 1.1 is provided and in Section 4, Result 1.2 is proved. In Section 5 attention is given to standard concurrency in the presence of RSP. An appendix on RSP in the setting of $\mu$CRL completes the paper.

Acknowledgement. Jos van Wamel is thanked for his careful proof reading.

2 Guardedness

Our first task is to show that each of the two specifications under consideration indeed specifies some process, i.e. has a unique solution for each identifier. This is only the case if these specifications are 'guarded'. For example, $X = a \cdot \tau_{\{a\}}(X)$ clearly does not specify a process: any process $aP$ with $P$ not containing $a$ is a solution for this specification. There are various definitions for guardedness of recursive specifications. Here we use a liberal version, defined in two stages (taken from [BBP94b]).

We employ the standard operational semantics of $\mu$CRL (see [GP90]), and speak of steps that can be performed by a declared, closed process expression.

Definition 2.1. Let $P$ be an expression containing $X$. An occurrence of $X$ in $P$ is $\tau$-guarded if $P$ has a subexpression $a \cdot Q$, where $a \in A \cup \{\tau\}$ and $Q$ contains this occurrence of $X$.

We call a recursive specification $E = \{X_j = T_j \mid j \in J\}$ $\tau$-guarded if by substituting expressions $T_j$ for occurrences $X_j$ in the right-hand sides ('unfolding') a finite number of times, one can obtain the situation that every occurrence of every $X_j$ in the unfolded right-hand sides is $\tau$-guarded.

The specification $E$ is $\tau$-founded if none of the $X_j$ gives rise to an infinite number of consecutive $\tau$ steps.

The specification $E$ is guarded if it is both $\tau$-guarded and $\tau$-founded.
Though $\tau$-foundedness can easily be rephrased in a more formal way, we refrain here from doing this and just give a few examples. The earlier mentioned specification $X = a \cdot \tau_{(a)}(X)$ is $\tau$-guarded, but not $\tau$-founded: $X$ can do an $a$ step to $\tau_{(a)}(X)$, which can only perform an infinite number of consecutive $\tau$ steps. On the other hand, the specification $X = X$ is $\tau$-founded, but not $\tau$-guarded.

We recall the queue specification of Bergstra and Klop:

**Definition 2.2.** For all $\{i, j, k\} = \{1, 2, 3\}$ we define

$$Q^{ij} = \sum_{d \in D} r_i(d)(Q^{ik} \| k s_j(d)Q^{kj})$$

where $x \| y$ is short for $\tau_{(c_d(d))d \in D} \circ \theta_{(r_k(d), s_k(d))d \in D}(x \| y)$.

We shall prove that these six defining equations form a guarded system, thus ensuring that the $Q^{ij}$s are uniquely determined by RSP. Observe that the data parameter in $r_i(d)$ and $s_j(d)$ is irrelevant for this analysis. Therefore we shall omit it, as well as the $\sum$ operator. Accessible states of $Q^{ij}$ will be parsed on their $\|$, structure. To this end the notion of $(i,j)$-tree is defined.

**Definition 2.3.** For all $\{i, j, k\} = \{1, 2, 3\}$ we define by simultaneous induction:

1. Every $Q^{ij}$ is an $(i,j)$-tree;

2. If $T$ is an $(i,j)$-tree and $T'$ an $(j,k)$-tree, then $T \cdot T'$ is an $(i,k)$-tree;

3. If $T$ is an $(i,j)$-tree, then $\tau k Q^{ij}$ is an $(i,k)$-tree.

Obviously, leaves of an $(i,j)$-tree are either of the form $Q^{i'j'}$ or of the form $s_{j'}Q^{i'j'}$. The left-right ordering of an $(i,j)$-tree induces an ordering of its leaves. From now on we speak about leftmost, rightmost and adjacent leaves with respect to this ordering.

**Lemma 2.4.** For every $(i,j)$-tree $T$ the following holds:

1. $T$ can do a step $r_i$ originating from its leftmost leaf, which is of the form $Q^{i'j'}$ for some $j' \neq i$;

2. $T$ can do a step $s_j$ originating from its rightmost leaf if and only if this rightmost leaf is of the form $s_{j'}Q^{i'j'}$ for some $i' \neq j$;

3. $T$ can do a step $\tau$ originating from two adjacent leaves of the form $s_{j'}Q^{i'j'}$ and $Q^{i'k'}$ ($i' \neq j'$, $j' \neq k'$) if and only if $T$ has two such adjacent leaves;

4. $T$ can do no other steps than those enlisted above.

**Proof.** By simultaneous induction on the structure of the $(i,j)$-tree $T$.

Case $T = Q^{ij}$: obvious.

Case $T = T' \cdot T''$ with $(i,k)$-tree $T'$ and $(k,j)$-tree $T''$. By the induction hypothesis the lemma holds for $T'$ and $T''$.

As for 1: $T$ can do the step $r_i$ of $T'$ since $i \neq k$.

As for 2: $T$ can do a possible step $s_j$ of $T''$ since $j \neq k$, and the condition is equivalent for $T$ and $T''$ since they share their rightmost leaf.

---

1A precise definition of $\tau$-foundedness can be found in [BBP94a]. In fact, the definition of guardedness can be further relaxed to $\tau$-guardedness and semi-$\tau$-foundedness, but we do not need this here. In [BG94] a related definition of guardedness is introduced.
As for 3: $T$ can do all steps $\tau$ that $T'$ and $T''$ can do, when the adjacent leaves are either both leaves of $T'$, or of $T''$. Moreover, if the rightmost leaf of $T'$ is of the form $s_k Q^{i,k}$, then, by the induction hypothesis $2$, $T'$ can do a step $s_k$. Also, by induction hypothesis $1$, $T''$ can do a step $r_k$ originating from its leftmost leaf. Now observe that the rightmost leaf of $T'$ and the leftmost leaf of $T''$ are adjacent, so we are happy to see that $s_k$ and $r_k$ communicate into $c_k$, which results in a $\tau$ step of $T$ due to the definition of $\|k$.

As for 4: follows from the induction hypotheses $1-4$ for $T'$ and $T''$ plus the observation that $s_k$ and $r_k$ above are encapsulated due to the definition of $\|k$.

Case $T \equiv T' s_j Q^{k,j}$ with $(i, k)$-tree $T'$: similar but simpler than the previous case (only $\tau$ steps from $T''$).

**Lemma 2.5.** If $T$ is an $(i, j)$-tree, then all accessible states of $T$ are $(i, j)$-trees.

**Proof.** By induction on the number of steps, it suffices to verify that each of the steps $1-3$ from the previous lemma results in an $(i, j)$-tree.

As for 1: The leftmost leaf of $T$, which is of the form $Q^{ij}$, is replaced by the $(i, j')$-tree $Q^{ik'} s_j Q^{k', j'}$ with $(i, j', k') = \{1, 2, 3\}$. The result is again an $(i, j)$-tree (to be proved by induction on $T$).

As for 2: The rightmost leaf of $T$, which is of the form $s_j Q^{i,j}$, is replaced by the $(i', j)$-tree $Q^{i', j}$. (Note that $s_j Q^{i,j}$ itself is not an $(i', j)$-tree, but this doesn't matter.) The result is again an $(i, j)$-tree (to be proved by induction on $T$).

As for 3: Combines 2 and 1 on adjacent leaves. The result is again an $(i, j)$-tree.

**Corollary 2.6.** All accessible states of $Q^{ij}$ are $(i, j)$-trees.

The converse of this corollary is not true:

\[ Q^{ij} \text{ is an $(i, k)$-tree which is not accessible from $Q^{ik}$.} \]

**Corollary 2.7.** The operational behaviour of an $(i, j)$-tree is completely determined by its list of leaves.

**Proof.** By Lemma 2.5 and Lemma 2.4. The conditions on the clauses $1-3$ of Lemma 2.4 are expressed in terms of the leaves only.

By the corollary above we can restrict attention to the leaves of an $(i, j)$-tree. We distinguish between $Q$-leaves of the form $Q^{i', j'}$ and $Q_s$-leaves of the form $s_j Q^{i', j'}$. We represent sequences of leaves by sequences of natural numbers in the following way:

\[ (n_1, ..., n_k) \equiv Q^{n_1-1} Q_s Q^{n_2-1} Q_s ... Q^{n_{k-1}-1} Q_s Q^{n_k}, \]

where $k \geq 1$, $1 \leq i < k \rightarrow n_i \geq 1$, and $Q^p$ represents a subsequence of $p$ $Q$-leaves $(p \geq 0)$ and $Q_s$ represents a $Q_s$-leaf.
By inspection of the steps 1 – 3 form Lemma 2.4 and the corresponding steps of the proof of Lemma 2.5 one gets the following transition system for \((i,j)\)-trees:

\[
\begin{align*}
(n_l + 1, n_2, \ldots, n_k) &\xrightarrow{r_i} (2, n_1, n_2, \ldots, n_k) \\
(n_1, \ldots, n_{k-1}, 0) &\xrightarrow{s_j} (n_1, \ldots, n_{k-1}) \\
(n_1, \ldots, n_i + 2, \ldots, n_k) &\xrightarrow{s_j} (n_1, \ldots, n_{i-1} + 2, n_i + 1, \ldots, n_k) \\
(n_1, \ldots, n_k + 1) &\xrightarrow{r_i} (n_1, \ldots, n_{k-1} + 2, n_k)
\end{align*}
\]

provided \(k > 1\)

This transition system allows us to prove that the defining equations of \(Q_{ij}\) are \(r\)-founded. Let \((n_1, \ldots, n_k)\), with \(k \geq 1\), represent such a state. Define:

\[t_k = n_k\]

\[t_i = n_i - 1 + 2t_{i+1} \quad \text{for all } 1 \leq i < k.\]

Then we have \(t_i = (n_i - 1)2^0 + (n_{i+1} - 1)2^1 + \ldots + (n_{k-1} - 1)2^{k-1-i} + n_k2^{k-i}\) for \(1 \leq i \leq k\). It is easily seen that for \(k > 1\) the maximum number of consecutive \(r\) steps from the state \((n_1, \ldots, n_k)\) is given by \(t_2 + \ldots + t_k\), ending always in a state \((t_1 + 1, 1, 1, \ldots, 1, 0)\). If \(k = 1\), then the state is \((n_k)\) and there are no \(r\) steps possible. In both states \((t_1 + 1, 1, 1, \ldots, 1, 0)\) and \((n_k)\) the process can only continue by an \(r_i\) or \(s_j\) step. We conclude that the defining equations of \(Q_{ij}\) are \(r\)-founded.

Observe that the defining equations of \(Q_{ij}\) are trivially \(r\)-guarded. Now we have the following theorem.

**Theorem 2.8.** The defining equations of \(Q_{ij}\) are guarded.

**Remark 2.9.** Theorem 2.8 justifies an application of RSP in an algebraic proof that \(Q_{ij}\) is a 'real' queue. This justification requires an analysis of the operational semantics of \(Q_{ij}\). It is therefore questionable whether the algebraic proof is to be preferred over an argument showing that the operational semantics of \(Q_{ij}\) is weakly bisimilar to that of the 'real' queue. Such an argument can be obtained without much difficulty from the above analysis, by giving the atomic actions their data parameters back.

We now recall the queue specification inspired by [Hoa85]:

**Definition 2.10.** For all \(\{i,j\} = \{2,3\}\) we define

\[
QB^i_j = \sum_{d \in D}(r_i(d) \cdot (QB^{il}_j \parallel s_j(d) \cdot B^{ij}))
\]

\[
B^{ij} = \sum_{d \in D}(r_i(d) \cdot s_j(d) \cdot B^{ij})
\]

where \(x \parallel y\) is short for \(\tau_{\{x \cdot d \mid d \in D\}} \circ \partial_{\{x(d), s \cdot d \mid d \in D\}}(x \parallel y)\).

In a similar way as Theorem 2.8 is proved, one can argue that this specification is guarded. Here one obtains a tree structure that is much simpler: \(QB^i_{1j}\) for \(i \in \{2,3\}\) is a \((1,i)\)-tree, and if \(T\) is a \((1,i)\)-tree with leftmost leaf \(QB^i_{1k}\), then replacing this leaf by the tree

\[
QB^i_{1j} \parallel j\]

or by the tree \(QB^i_{1j} \parallel k\) (with \(\{k,j\} = \{2,3\}\)) again yields a \((1,i)\)-tree. Without further proof we claim:

**Theorem 2.11.** The defining equations of \(QB^i_{1j}\) are guarded.
Table 1: The data type Sequence specified in μCRL.

<table>
<thead>
<tr>
<th>sort</th>
<th>Bool</th>
<th>var</th>
<th>d, e, f : D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>q, r : Sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>rew</td>
<td>right-elt(ε) = d₀</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>right-elt(q, d) = d</td>
</tr>
<tr>
<td>func</td>
<td>t, f :→ Bool</td>
<td></td>
<td>left-rm(ε) = ε</td>
</tr>
<tr>
<td></td>
<td>d₀ :→ D</td>
<td></td>
<td>left-rm(q, d) = q</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>non-empty(ε) = f</td>
</tr>
<tr>
<td></td>
<td>ε :→ Sequence</td>
<td></td>
<td>non-empty(q, d) = t</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>d.ε = ε.d</td>
</tr>
<tr>
<td></td>
<td>. : Sequence × D → Sequence</td>
<td></td>
<td>d.(q.ε) = (d.q).ε</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>right-elt : Sequence → D</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>left-rm : Sequence → Sequence</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-empty : Sequence → Bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>. : D × Sequence → Sequence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3 The Bergstra-Klop Specification defines a Queue

In this section we prove that the specification of Bergstra and Klop discussed earlier indeed defines a queue (Result 1.1). We give this proof in an algebraic fashion, in the proof theory of μCRL.

The data type Sequence. In Table 1 we (partly) specify the data involved. The data type D, the elements of which are stored in the queue, is left unspecified apart from an arbitrary constant d₀. Notably D is not assumed to be finite. The data type Sequence has two constructors, ε for the empty sequence, and a binary (infix written) function . : Sequence × D → Sequence for inserting a D-element to the right of a sequence. Though this function is not used in the standard specification, it is convenient in our proofs. The function right-elt extracts the right element of a sequence, and gives the arbitrary element d₀ ∈ D in case of emptiness, left-rm gives the left remainder of a sequence, and non-empty tests on inequality with ε. The standard queue specification employs the function . : D × Sequence → Sequence modelling insertion of a D-element at the left side of a sequence, which is specified in the last two lines.

As a consequence of our choice of constructors, the induction principle for the data type Sequence reads as follows:

(Induction Principle) \( \phi(ε) \land \forall d \in D \forall q \in \text{Sequence} \left[ \phi(q) \rightarrow \phi(q, d) \right] \rightarrow \forall q \left[ \phi(q) \right] \).

Lemma 3.1. Let d : D and q : Sequence.
1. non-empty(q) = t → right-elt(d.q) = right-elt(q),
2. non-empty(q) = t → left-rm(d.q) = d.left-rm(q), and
3. non-empty(q) = t → left-rm(q).right-elt(q) = q.
**Proof.** By induction. By way of example we give a proof of clause 3:
If \( q = \varepsilon \), then 3 follows trivially (from \( f = t \) anything can be derived). In case \( q = q'.d \), clause 3 follows also trivially from the equations defined in Table 1 and \( \rightarrow \) introduction.

**Proving Result 1.1.** Recall that for \( i, j \in \{1, 2, 3\}, i \neq j \) our standard definition of a queue with in-port \( i \) and out-port \( j \) is \( Q^{ij}(\varepsilon) \) with specification

\[
Q^{ij}(q : \text{Sequence}) = \sum_{d: D} (r_i(d) \cdot Q^{ij}(d.q)) + s_j(\text{right-elt}(q)) \cdot Q^{ij}(\text{left-qrt}(q)) < \text{non-empty}(q) \rightarrow \delta.
\]

We prove that

\[
Q^{ij}(\varepsilon) = \sum_{d: D} (r_i(d) \cdot (Q^{ik}(\varepsilon) || s_j(d) \cdot Q^{kj}(\varepsilon))) \quad \text{for all } \{i, j, k\} = \{1, 2, 3\}. \tag{1}
\]

Because \( Q^{ij} \) (see Definition 2.2) has a guarded specification of exactly the same form (see Theorem 2.8), it follows by a trivial RSP application that

\[Q^{ij} = Q^{ij}(\varepsilon).\]

Hence \( Q^{ij} \) also defines a queue with in-port \( i \) and out-port \( j \). The rest of this section is devoted to this proof.

**Identity (1).** By symmetry, it suffices to prove (1) for \( Q^{13}(\varepsilon) \). For readability, assume for the remainder of this section that the following variables are globally declared:

\[
d, e, f : D \\
q, r : \text{Sequence}.
\]

Define the auxiliary processes

\[
X(q, r, d) = \sum_{e: D} (r_i(e) \cdot X(e.q, r, d)) + s_j(d) \cdot Y(q, r)
\]

\[
Y(q, r) = \sum_{e: D} (r_i(e) \cdot Y(e.q, r)) + s_j(\text{right-elt}(r)) \cdot Y(q, \text{left-qrt}(r)) < \text{non-empty}(r) \rightarrow \delta + \tau \cdot Y(\text{left-qrt}(q), \text{right-elt}(q), r) < \text{non-empty}(q) \rightarrow \delta.
\]

Note that this specification is guarded: the summand starting with \( \tau \cdot Y(\ldots) \) does not give rise to an infinite \( \tau \) trace, since the length of the first \( \text{Sequence} \) argument in \( Y(\ldots) \) decreases and the condition tests on emptiness.

The process \( X(q, r, d) \) represents \( Q^{13} \) 'in state qrd' with its contents split up in a '\( Q^{12}(q) \) part', a '\( Q^{23}(r) \) part' and a 's3(d) part'. The process \( X(q, r, d) \) can receive data and store these in its leftmost argument, or send the \( d \) in its rightmost argument and evolve into \( Y(q', r) \). See also Figure 1.

The process \( Y(q, r) \) is meant to represent \( Q^{12}(q) ||_2 Q^{23}(r) \), and apart from performing \( r_i \) and \( s_3 \) steps, it explicitly 'transfers' \( q \) to \( r \) by performing \( \tau \) steps until \( q \) is \( \varepsilon \). The process \( Y(\varepsilon, \varepsilon) \) externally behaves as \( Q^{13}(\varepsilon) \). The process \( Y(q, r) \) is depicted in Figure 2.

The following lemma, which plays an important role in our proof, states that these (internal) \( \tau \) steps do not change the external behaviour of \( \tau \cdot Y(q, r) \).
Figure 1: The auxiliary process $X(q,r,d)$.

Figure 2: The auxiliary process $Y(q,r)$.

Lemma 3.2. For all $q,d,r$ it holds that $\tau \cdot Y(q,d,r) = \tau \cdot Y(q,d,r)$.

Proof. By double expansion, $x < t > y = x$, the last equation in Table 3.1 and Lemma 3.1,1 and 2, derive

$$\tau \cdot Y(q,d,r) = \tau \cdot \left( \sum_{e:D}(r_1(e) \cdot Y(e,(q,d),r)) + s_3(right-elt(r)) \cdot Y(q,d,left-rm(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + \tau \cdot Y(q,d,r) \right)$$

$$= \tau \cdot \left( \sum_{e:D}(r_1(e) \cdot Y((e,q),d,r)) + s_3(right-elt(r)) \cdot Y(q,d,left-rm(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + \right. \left. \begin{bmatrix} \sum_{e:D}(r_1(e) \cdot Y(e,(q,d),r)) + s_3(right-elt(r)) \cdot Y(q,d,left-rm(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + \\ \tau \cdot Y((e,q),right-elt(r),(d,r)) \triangleleft \text{non-empty}(r) \triangleright \delta \end{bmatrix} \right).$$

Regarding $d$ as the left element of $r$ gives a similar identity, obtained with expansion and Lemma 3.1,1 and 2, followed by application of the following consequence of B2:

$$\tau \cdot (x + y + z + z') = \tau \cdot (x + y + \tau \cdot (x + y + z + z'))$$

129
(conversely, this identity implies B2, using BPA and B1).

\[
\tau \cdot Y(q,d,r) = \tau \cdot \left( \sum_{e \in E} (r_1(e) \cdot Y(e,q,d,r)) + s_3(\text{right-elt}(r)) \cdot Y(q,d,\text{left-\text{rm}}(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + s_3(d) \cdot Y(q,e) \triangleleft \text{not}(\text{non-empty}(r)) \triangleright \delta \right) \\ + \tau \cdot (\sum_{e \in E} (Y(e,q,d,r)) + s_3(\text{right-elt}(r)) \cdot Y(q,d,\text{left-\text{rm}}(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + s_3(d) \cdot Y(q,e) \triangleleft \text{not}(\text{non-empty}(r)) \triangleright \delta + \tau \cdot Y(\text{left-\text{rm}}(q),\text{right-elt}(q),(d,r)) \triangleleft \text{non-empty}(q) \triangleright \delta \right). 
\]

Now consider the guarded equation \( N_1 \) defined by

\[
n(q,d,r) = \tau \cdot \left( \sum_{e \in E} (r_1(e) \cdot n(e,q,d,r)) + s_3(\text{right-elt}(r)) \cdot n(q,d,\text{left-\text{rm}}(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + s_3(d) \cdot n(e,q,d,r) + s_3(\text{right-elt}(r)) \cdot Y(q,d,\text{left-\text{rm}}(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + s_3(d) \cdot Y(q,e) \triangleleft \text{not}(\text{non-empty}(r)) \triangleright \delta + \tau \cdot Y(\text{left-\text{rm}}(q),\text{right-elt}(q),(d,r)) \triangleleft \text{non-empty}(q) \triangleright \delta \right). 
\]

Applying \( \tau \)-law B1 in the derivations above, it follows that both equations \( N_1/\lambda q d r . \tau \cdot Y(q,d,r)/n \) and \( N_1/\lambda q d r . \tau \cdot Y(q,d,r)/n \) are derivable. Hence RSP implies that \( \tau \cdot Y(q,d,r) = \tau \cdot Y(q,d,r) \).

The structure of the proof of Identity (1) is as follows:

\[
Q^{13}(e) = \sum_{d \in D} (r_1(d) \cdot Q^{13}(d,e)) \\
(\overset{2}{\overset{\otimes}{\overset{3}{\otimes}}} \overset{4}{\overset{\otimes}{\overset{\otimes}{\otimes}}}) \sum_{d \in D} (r_1(d) \cdot Y(e,d,e)) \\
(\overset{2}{\overset{\otimes}{\overset{3}{\otimes}}} \overset{4}{\overset{\otimes}{\overset{\otimes}{\otimes}}}) \sum_{d \in D} (r_1(d) \cdot X(e,e,d)) \\
(\overset{2}{\overset{\otimes}{\overset{3}{\otimes}}} \overset{4}{\overset{\otimes}{\overset{\otimes}{\otimes}}}) \sum_{d \in D} (r_1(d) \cdot (Q^{12}(e) \| s_3(d) \cdot Q^{23}(e)))
\]

showing which identities remain to be proved. The proofs given below all concern generalizations of these identities.

**Identity (2).** From Lemma 3.2 it follows that

\[
\tau \cdot Y(d,e,q) = \tau \cdot Y(e,d,q) = \tau \cdot Y(e,d,q). 
\]

By expansion, B1 and the axioms for conditionals this leads to

\[
Y(e,q) = \sum_{d \in D} (r_1(d) \cdot Y(d,e,q)) + s_3(\text{right-elt}(q)) \cdot Y(e,\text{left-\text{rm}}(q)) \triangleleft \text{non-empty}(q) \triangleright \delta \\
= \sum_{d \in D} (r_1(d) \cdot Y(e,d,q)) + s_3(\text{right-elt}(q)) \cdot Y(e,\text{left-\text{rm}}(q)) \triangleleft \text{non-empty}(q) \triangleright \delta. 
\]

By a trivial application of RSP we obtain:

\[
Q^{13}(q) = Y(e,q). 
\]
Identity (3). First observe that

\[ X(r,e,q,d) = X(r,e,q,d) \]

by RSP: consider the guarded equation \( N_2 \) defined by

\[
n(r,e,q,d) = \sum_{f:D}(r_1(f) \cdot n(f,r,e,q,d)) + s_3(d) \cdot Y(r,e,q).
\]

Now \( N_2[X(r,e,q,d)/n] \) is derivable (use Lemma 3.2), and so is \( N_2[X(r,e,q,d)/n] \) (immediate).

With the above observation, further derive

\[
X(e,q,d) = \sum_{e:D}(r_1(e) \cdot X(e,e,q,d)) + s_3(d) \cdot Y(e,q).
\]

Because

\[
Y(e,q,d) = \sum_{e:D}(r_1(e) \cdot Y(e,e,q,d)) + s_3(d) \cdot Y(e,q)
\]

we can apply RSP on the guarded equation \( M_2 \) defined by

\[
n(q,d) = \sum_{e:D}(r_1(e) \cdot n(e,q,d)) + s_3(d) \cdot Y(e,q)
\]

because both \( M_2[X(r,e,q,d)/n] \) and \( M_2[Y(e,q,d)/n] \) are derivable, whence this result follows with RSP.

Identity (4). First a trivial result on the commutativity of \( \parallel \), namely \( Q^{12}(q) \parallel Q^{23}(r) = Q^{23}(r) \parallel Q^{12}(q) \). Consider the guarded equation \( N_3 \) with only two occurrences of the identifier \( n \):

\[
n(q,r) = \sum_{d:D}(r_1(d) \cdot (Q^{12}(d,q) \parallel Q^{23}(r))) + s_2(right-elt(q)) \cdot (Q^{12}(left-rm(q)) \parallel Q^{23}(r)) \parallel non-empty(q) \parallel \delta + \sum_{d:D}(r_2(d) \cdot (Q^{23}(d,r) \parallel Q^{12}(q))) + s_3(right-elt(r)) \cdot (Q^{23}(left-rm(r)) \parallel Q^{12}(q)) \parallel non-empty(r) \parallel \delta + c_2(right-elt(q)) \cdot n(left-rm(q), right-elt(q), r) \parallel non-empty(r) \parallel \delta.
\]

It follows easily that both \( N_3[X(q,r,q,d)/n] \) and \( N_3[X(q,r,q,d)/n] \) are derivable, whence this result follows with RSP. This trivial type of identities also follows from a general result on 'linearly specified' processes proven in Section 5.

With RSP and the identity above one proves

\[
X(q,r,d) = Q^{12}(q) \parallel_2 s_3(d) \cdot Q^{23}(r)
\]

\[
Y(q,r) = Q^{12}(q) \parallel_2 Q^{23}(r)
\]
simply by one expansion of all processes involved: for the system of guarded equations $M_3$ defined by

$$
n(q,r,d) = \sum_{e : D} (r_1(e) \cdot n(e,q,r,d)) + s_3(d) \cdot m(q,r)
$$

$$
m(q,r) = \sum_{e : D} (r_1(e) \cdot m(e,q,r)) + s_3(right-elt(r)) \cdot m(q, left-rm(r)) \triangleleft \text{non-empty}(r) \triangleright \delta + \tau \cdot m(left-rm(q), right-elt(q), r) \triangleleft \text{non-empty}(q) \triangleright \delta
$$

the derivability of

$$
M_3[\lambda qrd . X(q,r,d)/n , \lambda qr . Y(q,r)/m] \quad \text{and}
M_3[\lambda qrd . (Q^{12}(q) \parallel_2 s_3(d) \cdot Q^{23}(r))/n , \lambda qr . (Q^{12}(q) \parallel_2 Q^{23}(r))/m]
$$

follows immediately. This finishes the proof of Identity (1), and hence the proof of Result 1.1.

From the proof above it follows that two queues in parallel and connected via a hidden port externally behave as a single queue (cf. [BBP94a, Kor94]):

**Corollary 3.3.** For all $\{i,j,k\} = \{1,2,3\}$ it holds that $Q^{ij}(e) = Q^{ik}(e) \parallel_k Q^{kj}(e)$.

**Proof.** $Q^{ij}(e) \triangleq Y(e,e) \triangleq Q^{ik}(e) \parallel_k Q^{kj}(e)$. $\square$

## 4 Queues as Linked One Element Buffers

In this section we prove for our standard queue specification that for all $\{i,j\} = \{2,3\}$

$$
Q^{1i}(e) = \sum_{d : D} (r_i(d) \cdot (Q^{1j}(e) \parallel_j s_i(d) \cdot B^{ij}))
$$

(6)

where $B^{ij}$ is specified by the guarded equation $B^{ij} = \sum_{d : D} (r_i(d) \cdot s_j(d) \cdot B^{ij})$. By RSP it then follows immediately that $QB^{1i}$ (see Definition 2.10, and for its guardedness Theorem 2.11) also defines a queue with in-port 1 and out-port $i$. This proves Result 1.2.

By symmetry, it suffices to prove (6) for $Q^{12}(e)$. The proof is in the same style as that of the previous result, and is given with less detail (though we maintain the claim on exactness).

Let $d$ be a variable of type $D$ and $q$ a variable of type $Sequence$. We define the following auxiliary processes:

$$
\overline{x}(q,d) = \sum_{e : D} (r_1(e) \cdot \overline{x}(e,q,d)) + s_3(d) \cdot \overline{y}(q)
$$

$$
\overline{y}(q) = \sum_{e : D} (r_1(e) \cdot \overline{y}(e,q)) + \tau \cdot \overline{x}(left-rm(q), right-elt(q)) \triangleleft \text{non-empty}(q) \triangleright \delta.
$$

Both these are specified such that

$$
\overline{x}(q,d) = (Q^{12}(q) \parallel_2 s_3(d) \cdot B^{23})
$$

$$
\overline{y}(q) = (Q^{12}(q) \parallel_2 B^{23})
$$
follow trivially (cf. identity (4) in the preceding proof). The structure of the proof of Identity (6) is as follows:

\[
Q^{12}(\epsilon) = \sum_{d.D}(r_1(d) \cdot Q^{12}(d,\epsilon))
\]
\[
\overset{(8)}{=} \sum_{d.D}(r_1(d) \cdot \overline{Y}(d,\epsilon))
\]
\[
= \sum_{d.D}(r_1(d) \cdot \overline{V}(e,\epsilon))
\]
\[
\overset{(7)}{=} \sum_{d.D}(r_1(d) \cdot \overline{X}(\epsilon, d))
\]
\[
= \sum_{d.D}(r_1(d) \cdot (Q^{12}(\epsilon) \parallel_2 s_3(d) \cdot B^{23})).
\]

In the following, generalizations of the two marked identities are proved.

**Identity (7).** Consider the guarded equation \(M_4\) defined by

\[
n(q,d) = \tau \cdot (\sum_{e.D}(r_1(e) \cdot n(e,q,d)) + \tau \cdot (\sum_{e.D}(r_1(e) \cdot \overline{X}(e,q,d)) + s_3(d) \cdot \overline{Y}(q,\epsilon))).
\]

From B1, B2 and the last equation in Table 3.1, it follows that both \(M_4[\lambda qd \cdot \tau \cdot \overline{X}(q,d)/n] \) and \(M_4[\lambda qd \cdot \tau \cdot \overline{Y}(q,d)/n] \) are derivable. Hence by RSP we find

\[
\tau \cdot \overline{Y}(q,d) = \tau \cdot \overline{X}(q,d).
\]  

**Identity (8).** By expansion, B1 and Lemma 3.1.3 derive

\[
\tau \cdot \overline{Y}(q) = \tau \cdot (\sum_{e.D}(r_1(e) \cdot \overline{Y}(e,q)) + \tau \cdot \overline{X}(\text{left-elm}(q), \text{right-elt}(q)) \triangleleft \text{non-empty}(q) \triangleright \delta)
\]
\[
= \tau \cdot [\sum_{e.D}(r_1(e) \cdot \overline{X}(e,\text{left-elm}(q), \text{right-elt}(q))) + s_3(\text{right-elt}(q)) \cdot \overline{Y}(\text{left-elm}(q))]
\]
\[
\overset{(7)}{=} \tau \cdot [\sum_{e.D}(r_1(e) \cdot \overline{X}(e,q)) + s_3(\text{right-elt}(q)) \cdot \overline{Y}(\text{left-elm}(q))]
\]
\[
\overset{(3,1,3)}{=} \tau \cdot [\sum_{e.D}(r_1(e) \cdot \overline{Y}(\text{left-elm}(q))) + s_3(\text{right-elt}(q)) \cdot \overline{Y}(\text{left-elm}(q))]
\]
\[
\overset{\triangleleft \text{non-empty}(q) \triangleright \delta.}{=} \tau \cdot Q^{13}(q) = \tau \cdot \overline{Y}(q).
\]

This finishes the proof of Identity (6), and hence the proof of Result 1.2.

\[\text{Derive the law } \tau \cdot (x + y \triangleright \delta) \equiv \tau \cdot (x + \tau(x + y) \triangleright \delta) \text{ by case distinction on } \delta.\]
Table 2: Axioms of Standard Concurrency (SC), with $a \in A$.

\begin{align*}
\text{SC1} & \quad (x \parallel y) \parallel z = x \parallel (y \parallel z) \\
\text{SC2} & \quad x \parallel \delta = x\delta \\
\text{SC3} & \quad x \uparrow y = y \uparrow x \\
\text{SC4} & \quad (x \uparrow y) \uparrow z = x \uparrow (y \uparrow z) \\
\text{SC5} & \quad x \uparrow (y \parallel z) = (x \uparrow y) \parallel z \\
\text{SC5-a} & \quad x \uparrow (ay \parallel z) = (x \uparrow a) \parallel z
\end{align*}

5 RSP and Standard Concurrency

**Standard Concurrency** (SC) is the axiomatic support for the identities

\[
\begin{align*}
    x \parallel y &= y \parallel x \\
    (x \parallel y) \parallel z &= x \parallel (y \parallel z)
\end{align*}
\]

expressing commutativity and associativity of $\parallel$ (see [BK85, BW90, GP91]). The axioms given in Table 2 and those of ACP allow one to derive these identities. All axioms in Table 2 are valid for closed (recursion-free) terms, to be proved by structural induction using the commutativity and associativity of the communication function on atomic actions.

The notion SC is a relative one. In the setting of observation congruence, the axiom SC5 is not sound: for example

\[
a \uparrow (\tau b \parallel c) \neq (a \uparrow \tau b) \parallel c
\]

because the left-hand term has a summand $(a|\emptyset)b$ which need not be equal to $\delta$ and which is absent in the right-hand term. Therefore, the alternative SC5-a is used in the setting of observation equivalence.

**ACP and standard concurrency.** Linear systems of recursion equations concern a syntactical characterization of certain processes. A system $(X_i)_{i=1}^n$ of recursion equations is linear if it consists of equations of the form

\[
X_i = \sum_{j=1}^n (\alpha_{ij} X_j) + \beta_i
\]

for $i = 1, \ldots, n$, where $\alpha_{ij}, \beta_i$ are sums of atomic actions or equal $\delta$. A process is regular if it satisfies a (finite) linear system of recursion equations.

First observe that regular processes are closed under $\parallel, \uparrow, \mid, \partial_H$, and that this can be proved with RSP: expansion with linear systems as arguments always yields a linear system. RSP can also be used to prove commutativity and associativity of $\parallel$ for regular processes (and therewith the identities characterized by the standard concurrency axioms). This is because linearly specified processes expand recurrently in the scope of the parallel operator, giving way to RSP applications, as we show below.

Let for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ the regular processes $X_i$ and $Y_j$ be defined by

\[
X_i = \sum_{k=1}^n (\alpha_{ik} X_k) + \beta_i
\]
\begin{equation}
Y_j = \sum_{l=1}^{m} (\bar{\alpha}_{jl}Y_l) + \bar{\beta}_j.
\end{equation}

The following guarded system, containing \(nm\) equations, serves to prove that \(X_i \parallel Y_j = Y_j \parallel X_i\) for all appropriate \(i, j\).

\begin{equation}
n_{ij} = \sum_{k=1}^{n} (\alpha_{ik} \cdot (X_k \parallel Y_j)) + \sum_{l=1}^{m} (\bar{\alpha}_{jl} \cdot (Y_l \parallel X_i)) + \\
\beta_iY_j + \bar{\beta}_jX_i + \\
\sum_{k=1}^{n} \sum_{l=1}^{m} ((\alpha_{ik}\bar{\alpha}_{jl}n_{kl})) + \\
\sum_{k=1}^{n} ((\alpha_{ik}\bar{\beta}_j)X_k) + \sum_{l=1}^{m} ((\bar{\alpha}_{jl}\beta_l)Y_l) + (\beta_l\bar{\beta}_j).
\end{equation}

Observe that both \(X_i \parallel Y_j\) and \(Y_j \parallel X_i\) are solutions for \(n_{ij}\). This is a consequence of the commutativity of the communication function and the communication axioms CM 5.6 and CM 8.9. By RSP we conclude that \(X_i \parallel Y_j = Y_j \parallel X_i\) for all appropriate \(i, j\).

As to the associativity of \(\parallel\), taking a third linear system \(\langle Z_p \rangle_{p=1}^{r}\), the expansions of both \((X_i \parallel Y_j) \parallel Z_p\) and \(X_i \parallel (Y_j \parallel Z_p)\) give identical patterns, apart from associative and commutative variants of the three identifiers and communications. It follows immediately that \(\parallel\) also is an associative operation for regular processes.

Finally, given the commutativity and associativity of \(\parallel\), the SC identities follow easily for linearly specified processes. As each recursion free process can be specified linearly, one obtains as a corollary that the SC axioms are valid for this class of processes.

This leads to the following result.

**Theorem 5.1.** For the class of processes definable in ACP with linear recursive specifications, RSP implies that \(\parallel\) is a commutative and associative operation, and that all axioms in Table 2 are valid.

**ACP with abstraction.** In the setting with \(\tau\), linear systems may have \(\tau\) as a constant. Adopting branching bisimilarity (i.e. the \(\tau\)-laws B1 and B2), one can prove in the same way as above that RSP implies standard concurrency for all processes definable by guarded linear systems. In observation equivalence this requires more work, due to the additional \(\tau\)-laws: \(\tau \cdot x = y \cdot x\) \(x\) and \(x \cdot \tau \cdot y = x \cdot y\). In this case, a solution is to simultaneously prove the commutativity (associativity) of \(\parallel\) and \(\cdot\).

**Up to \(\mu\)CRL.** It is not hard to see that the results of this section can be generalized to \(\mu\)CRL, though a general formulation of the appropriate type of specifications is cumbersome. By way of example, regard the proof of \(Q^{12}(q) \parallel Q^{23}(r) = Q^{23}(r) \parallel Q^{12}(q)\) given in Section 3. Because the possible infinity of such systems is in this case captured by data, and because the possible \(\tau\) occurrences do not affect the arguments above in a setting with data and conditionals, it follows immediately that the standard concurrency result is preserved in the \(\mu\)CRL setting. Finally, in [Pon92] it is shown that each recursive (finitely branching) transition system has a canonical \(\mu\)CRL-linear specification modulo strong bisimilarity. Hence, all processes that can be associated with recursive transition systems satisfy the standard concurrency identities.

**References**

Appendix: RSP in $\mu$CRL

In this appendix we discuss the use of RSP in $\mu$CRL. In order to derive identities between $\mu$CRL-processes, an extended version of the Recursive Specification Principle (RSP, see e.g., [BW90]) has been introduced in [GP91, GP94]. This was done to handle data parametric, conditional processes. Given a $\mu$CRL specification $E$, the (extended) rule RSP employs a system of process equations that is 'fresh' with respect to $E$ and that must be 'guarded'.

Let a $\mu$CRL specification $E$ be given and let $n_1, \ldots, n_m$ be $m$ different fresh process identifiers. We call a system $G$ of $m$ equations $G_1, \ldots, G_m$ a system of process-equations over $E$ if

1. each equation $G_i$ has at its left-hand side an expression of the form $n_i$ or $n_i(x_{i1}, \ldots, x_{im_i})$ where each $x_{ij}$ is a data variable over data declared in $E$, and
2. the right hand side of each equation $G_i$ is a (well-formed) process expression over $E$ that may contain the new, properly typed identifiers $n_i$ or $n_i(t_{i1}, \ldots, t_{im_i})$ for $1 \leq i \leq m$.

The rule RSP in the setting of $\mu$CRL is restricted to systems of process-equations that are guarded. Though various characterizations of guardedness occur in the ACP/$\mu$CRL literature [BW90, GP90, GP91, BG94, BBP94b], we here stick to Definition 2.1 and add the following comments:

1. The guardedness of $G$ refers to the specification of all process identifiers occurring in $G$;
2. $\tau$-guardedness is a special case of the definition of guardedness in [GP91, GP94];
3. $\tau$-foundedness can be rephrased in a formal way using the standard operational semantics for $\mu$CRL defined in [GP90].

Next we recall the substitution mechanism for a system $G = G_1, \ldots, G_m$ of process-equations over $E$ defined in [GP91, GP94]. Abbreviating the (possible) variables of $n_i$ by $\bar{x}_i$,

$$G_i[\lambda \bar{x}_i . p(\bar{x}_i)/n_i]$$

is defined as the equation obtained by substituting $\lambda \bar{x}_i . p(\bar{x}_i)$ for the $n_i$-occurrences in $G_i$, and then repeatedly performing $\beta$-conversion on the respective arguments of the process identifier $n_i$. For any identifier without arguments only the substitution of $p$ is performed.

Given a guarded system $G_1, \ldots, G_m$ of $m$ process-equations over $E$, the $\mu$CRL version of the rule RSP is as follows:

$$
\frac{G_i[\lambda \bar{x}_j . p_j(\bar{x}_j)/n_j]_{j=1}^m}{G_i[\lambda \bar{x}_j . q_j(\bar{x}_j)/n_j]_{j=1}^m} \quad \text{for } 1 \leq i \leq m
$$

where

- for $1 \leq i \leq m$ the $p_i(\bar{x}_i)$ and $q_i(\bar{x}_i)$ are well-formed process terms over $E$,
- the notation $[\ldots]_{j=1}^m$ abbreviates the $m$ given, simultaneous substitutions.

Further information on $\mu$CRL syntax, semantics and proof theory can be found in [GP90, GP91, BG94, GP94].
Prefix Iteration in Basic Process Algebra: applying termination techniques

Wan Fokkink & Hans Zantema

Utrecht University

Departments of Philosophy and of Computer Science

Utrecht, The Netherlands

fokkink@phil.ruu.nl hansz@cs.ruu.nl

Abstract

Prefix iteration $\mu^* x$ is added to Basic Process Algebra with deadlock and empty process. We present a finite equational axiomatization for this process algebra, and we prove that this axiomatization is complete with respect to strong bisimulation equivalence. This result is a mild generalization of a similar result in the setting of basic CCS in Fokkink (1994b).

To obtain this completeness result, we set up a rewrite system based on the axioms. In order to prove that this rewrite system is normalizing, we apply the technique of dummy elimination from Ferreira and Zantema (1994). Finally, we present an involved proof that this rewrite system is terminating. For this purpose, we generalize a termination theorem from Zantema and Geser (1995) to the setting of rewriting modulo equations, since process terms are considered modulo AC of the $+$. 

1 Introduction

Kleene (1956) defined a binary operator $\cdot^*$ in the context of finite automata, called Kleene star or iteration. Intuitively, the expression $p^* q$ yields a solution for the recursive equation $X = p \cdot X + q$. In other words, $p^* q$ can choose to execute either $p$, after which it evolves into $p^* q$ again, or $q$, after which it terminates.

In this paper, we study the prefix variant of iteration in bisimulation equivalence, in the setting of Basic Process Algebra (BPA) together with the deadlock $\delta$ from Bergstra and Klop (1984) and the empty process $\epsilon$ from Koymans and Vrancken (1985).

Milner (1984) proposed an axiomatization for the Kleene star in a process algebra equivalent to BPA$_\delta$\&, including a conditional axiom for iteration from Salomaa (1966), and he raised the question whether his axiomatization is complete with respect to bisimulation equivalence. This question is, to our knowledge, still open.

Bergstra, Bethke and Ponse (1994) considered the Kleene star in BPA, and they suggested a finite equational axiomatization for this algebra. Fokkink and Zantema (1994) (see also Fokkink (1994a)) proved that this axiomatization is complete with respect to strong bisimulation equivalence.

139
Sewell (1994) proved that there does not exist a complete finite equational axiomatization for the Kleene star in $\text{BPA}_S$ modulo bisimulation, due to equivalences such as $(x^n)^* \delta \Leftrightarrow x^* \delta$ for $n \geq 1$. In order to obtain a complete finite equational axiomatization nevertheless, we replace the binary iteration operator $x^* y$ by its unary prefix version $\mu x$, where the argument at the left is restricted to atomic actions and deadlock and empty process. The resulting algebra is denoted by $\text{BPA}_{\mu}^\star$.

We propose five equational axioms for prefix iteration. Three of these axioms are actually instantiations of the three axioms for the Kleene star. We prove that these five axioms, together with the nine standard axioms of $\text{BPA}_{\mu}^\star$, are a complete axiomatization for $\text{BPA}_{\mu}^\star$ with respect to bisimulation. This result has been inspired by the need for a complete axiomatization for prefix iteration in an extension of $\text{BPA}_S$ with discrete time, in a revision of Baeten and Bergstra (1992).

Our result is a mild generalization of a similar completeness result for basic CCS extended with prefix iteration in Fokkink (1994b) (see also Fokkink (1994a)), where multiplication is restricted to its prefix counterpart $a \cdot x$, and the empty process and single atomic actions in $A$ are not used in the construction of the syntax. Besides the five extra axioms for $\text{BPA}_{\mu}^\star$, compared with basic CCS, and two extra axioms to deal with terms of the form $\delta p$ and $\epsilon p$, we only need one extra axiom for prefix iteration, namely

$$(a^\ast x)y = a^\ast(xy).$$

The strategy of the completeness proof that is presented here is fully different from the one in Fokkink (1994b). Process terms are modified by means of a rewrite system. The proof that this rewrite system is normalizing, which means that each process term can be reduced to a ground normal form, uses the technique of dummy elimination from Ferreira and Zantema (1994). The completeness theorem is proved by showing that bisimilar ground normal forms of the rewrite system are provably equal.

Next, we present an involved proof that the rewrite system is terminating, based on an abstract commutation technique from Zantema and Geser (1995). This technique is closely related to an earlier technique from Bellegarde and Lescanne (1990). In order to apply this technique, first we have to generalize it to the setting of rewriting modulo equations since we take $+$ modulo $\mathcal{AC}$. This generalization is presented in the Sections 4.1 and 4.2. It can be considered as a general applicable technique for proving termination of rewriting modulo equations. This is of interest itself, independent from the field of process algebra. Basically, termination of a rewrite system $R$ is proved by means of termination of a simplified rewrite system $S$ and an auxiliary rewrite system $A$ connecting $R$ and $S$. Surprisingly, for extending this framework to the setting of rewriting modulo a set of equations $E$, no cooperation between $R$ and $E$ is required, only between $A$ and $E$.

Aceto and Ingólfsdóttir (1995) study prefix iteration together with the silent step in observation congruence. They prove that in basic CCS, three equational axioms suffice to describe the interplay between the silent step and prefix iteration modulo observation congruence.

\[ \text{Conjecture: neither does there exist a complete finite equational axiomatization for the Kleene star in } \text{BPA}_S \text{ modulo bisimulation, due to equivalences such as } ((x + e)^n)^* y \Leftrightarrow x^* y \text{ for } n \geq 1. \]
Fokkink (1995) studies prefix iteration together with the silent step in branching bisimulation. In BPA$_{\delta e}$, two equational axioms suffice to describe the relation between the silent step and prefix iteration modulo branching bisimulation.

Acknowledgements. Jos Baeten initiated this research. Luca Aceto, Alfons Geser and Rob van Glabbeek provided helpful comments.

2 BPA$_{\delta e}$ with Prefix Iteration

2.1 The syntax and semantics

We assume an alphabet $A$ of atomic actions, and two special constants $\delta$ and $\epsilon$, which represents deadlock and empty process respectively. We use $a$ to range over $A$ and $\mu$ to range over $A \cup \{\delta, \epsilon\}$. The signature of the process algebra BPA$_{\delta e}$ is built from the constants in $A \cup \{\delta, \epsilon\}$, alternative composition $x + y$, sequential composition $x \cdot y$, and prefix iteration $\mu^*x$.

Table 1 presents an operational semantics for BPA$_{\delta e}$ in the style of Plotkin (1981). Prefix iteration $a^*x$ can choose to execute either $a$, after which it evolves into $a^*x$ again, or $x$. The expression $x \downarrow$ denotes successful termination of $x$.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon \downarrow$</td>
<td>$a \xrightarrow{a} \epsilon$</td>
</tr>
<tr>
<td>$x \downarrow$</td>
<td>$a \xrightarrow{a} x'$</td>
</tr>
<tr>
<td>$(x + y) \downarrow$</td>
<td>$(y + x) \downarrow$</td>
</tr>
<tr>
<td>$x + y \xrightarrow{a} x'$</td>
<td>$y + x \xrightarrow{a} x'$</td>
</tr>
<tr>
<td>$x \downarrow$</td>
<td>$y \downarrow$</td>
</tr>
<tr>
<td>$(x \cdot y) \downarrow$</td>
<td>$x \cdot y \xrightarrow{a} y'$</td>
</tr>
<tr>
<td>$x \cdot y \xrightarrow{a} x'$</td>
<td>$x \cdot y \xrightarrow{a} x' \cdot y$</td>
</tr>
<tr>
<td>$x \downarrow$</td>
<td>$\mu^*x \xrightarrow{a} \mu^*x \xrightarrow{a} x'$</td>
</tr>
<tr>
<td>$(\mu^*x) \downarrow$</td>
<td>$a^*x \xrightarrow{a} a^*x$</td>
</tr>
</tbody>
</table>

Table 1: Action rules for BPA$_{\delta e}$

Our model for BPA$_{\delta e}$ consists of all the closed terms that can be constructed from the constants in $A \cup \{\delta, \epsilon\}$ together with the three operators. That is, the BNF grammar for the collection of process terms is as follows:

$$p ::= \mu \mid p + p \mid p \cdot p \mid \mu^*p.$$ 

As binding convention, $^*$ binds stronger than $\cdot$, which binds stronger than $+$. Often, $p \cdot q$ will be abbreviated to $pq$.

Process terms are considered modulo (strong) bisimulation equivalence from Park (1981). Intuitively, two process terms are bisimilar if they have the same branching structure.
Definition 2.1 Two processes \( p \) and \( q \) are bisimilar, denoted by \( p \leftrightarrow q \), if there exists a symmetric binary relation \( \mathcal{B} \) on processes, which relates \( p \) and \( q \), such that

1. if \( r \xrightarrow{a} r' \) and \( r \mathcal{B}s \), then there is a transition \( s \xrightarrow{a} s' \) with \( r' \mathcal{B}s' \),
2. if \( r \downarrow \) and \( r \mathcal{B}s \), then \( s \downarrow \).

The action rules in Table 1 are in the 'path' format of Baeten and Verhoef (1993). Hence, bisimulation equivalence is a congruence with respect to all the operators, i.e. if \( p \leftrightarrow p' \) and \( q \leftrightarrow q' \), then \( p + q \leftrightarrow p' + q' \) and \( p \cdot q \leftrightarrow p' \cdot q' \) and \( \mu^*p \leftrightarrow \mu^*p' \). See Baeten and Verhoef (1993) for the definition of the path format, and for a proof of this congruence result. This proof uses the extra assumption that the action rules are 'well-founded'; in Fokkink and Van Glabbeek (1995) (see also Fokkink (1994a)) it has been shown that this requirement can be dropped.

Furthermore, the action rules for \( \text{BPA}_d \) are 'pure', which is a syntactic criterion from Groote and Vaandrager (1992), and the three action rules for prefix iteration incorporate the Kleene star in the left-hand side of their conclusions. Hence, \( \text{BPA}^*_d \) is an operationally conservative extension of \( \text{BPA}_d \), i.e. the action rules for prefix iteration do not influence the transition systems of \( \text{BPA}_d \) terms. See Verhoef (1994) for a proof of this conservativity result.

2.2 The axioms
Table 2 contains an axiom system for \( \text{BPA}^*_d \), which consists of the nine axioms from \( \text{BPA}_d \) together with five axioms for prefix iteration. The axioms MI1,3 already appeared in Hennessy (1981), as axioms (in CCS) for the delay operator, which is an instance of prefix iteration.

In the sequel, \( p = q \) will mean that this equality can be derived from the axioms. The axiomatization A1-9+MI1-5 is sound with respect to bisimulation equivalence, i.e. if \( p = q \) then \( p \leftrightarrow q \). Since bisimulation is a congruence, this can be verified by checking soundness for each axiom separately, which is left to the reader. In this paper it is proved that the axiomatization is complete with respect to bisimulation, i.e. if \( p \leftrightarrow q \) then \( p = q \).

2.3 Completeness of the axioms
We prove that the axioms of \( \text{BPA}^*_d \) are complete with respect to bisimulation equivalence. From now on, process terms are considered modulo \( AC \) of the +, that is, modulo associativity and commutativity of the +. In the sequel, \( p =_{AC} q \) denotes that \( p \) and \( q \) are equal modulo \( AC \) of the +, and we say that \( p \) and \( q \) are of the same form.

Table 3 contains a rewrite system \( R_0 \), which reduces sequential composition to its prefix counterpart (rules 1-5), and which eliminates expressions of the form \( \delta^*x \) and \( \epsilon^*x \) (rules 6 and 7), and which reduces occurrences of prefix iteration in the context of alternative composition and of prefix iteration (rules 8 and 9). The rewrite rules are to be interpreted modulo \( AC \) of the +.

In the completeness proof it will be shown that if two ground normal forms of \( R_0 \), i.e. two process terms that cannot be reduced by the rewrite rules in \( R_0 \), are
Table 2: Axioms for $BPA_{5e}^*$

| A1 | $x + y = y + x$ |
| A2 | $(x + y) + z = x + (y + z)$ |
| A3 | $x + x = x$ |
| A4 | $(x + y)z = xz + yz$ |
| A5 | $(xy)z = x(yz)$ |
| A6 | $x + \delta = x$ |
| A7 | $\delta x = \delta$ |
| A8 | $xe = x$ |
| A9 | $\epsilon x = x$ |
| M11 | $a \cdot a^* x + x = a^* x$ |
| M12 | $(a^* x)y = a^*(xy)$ |
| M13 | $a^*(a^* x) = a^* x$ |
| M14 | $\delta^* x = x$ |
| M15 | $\epsilon^* x = x$ |

bisimilar, then they are provably equal. Hence, in order to obtain completeness for the full class of process terms, we desire to know that each process term is provably equal to a ground normal form of $R_0$.

**Theorem 2.2** Each process term is provably equal to a ground normal form of $R_0$.

Since the rewrite rules in $R_0$ can all be deduced from the axioms, this theorem follows from the fact that $R_0$ is normalizing, which means that each process term can be reduced to a ground normal form by means of the rewrite rules in $R_0$. Normalization of $R_0$ will be deduced in Section 3, using the technique of dummy elimination from Ferreira and Zantema (1994).

Independently, in Section 4 we shall prove a stronger fact than normalization of $R_0$, namely that $R_0$ is terminating, also called strongly normalizing, which means that $R_0$ does not allow any infinite reductions. Normalization of the rewrite system $R_0$ provides an algorithm to decide whether or not two terms are bisimilar: find their normal forms by breadth-first search; two terms are bisimilar if and only if their normal forms are equal. Termination implies that this algorithm can be simplified: no back-tracking or breadth-first search is needed for finding normal forms.

Process terms are considered modulo associativity and commutativity of the $\cdot$. Therefore, in order to solve the termination problem for $R_0$, we will have to generalize a termination result from Zantema and Geser (1995) to the setting of rewriting
1. \((x + y)z \rightarrow xz + yz\)
2. \((xy)z \rightarrow x(yz)\)
3. \(\delta x \rightarrow \delta\)
4. \(\epsilon x \rightarrow x\)
5. \((a^*x)y \rightarrow a^*(xy)\)
6. \(\delta^*x \rightarrow x\)
7. \(\epsilon^*x \rightarrow x\)
8. \(a^*x + y \rightarrow a \cdot a^*x + x + y\)
9. \(a^*(b^*x) \rightarrow a^*(b \cdot b^*x + x)\)

Table 3: The rewrite system \(R_0\)

modulo equations. This generalization and its application to \(R_0\) constitute termination techniques that are of interest independent from the field of process algebra.

For convenience, we adapt ground normal forms a bit further to basic terms. In the completeness proof it will be shown that bisimilar basic terms are provably equal.

In the sequel, \(\xi\) ranges over the set \(\{\delta, \epsilon\}\).

**Definition 2.3** A basic term is of the form either \(\sum_{i=1}^{n} a_ip_i + \xi\) or \(a^*(\sum_{i=1}^{n} a_ip_i + \xi)\), where the terms \(p_i\) are basic.

**Corollary 2.4** Each process term is provably equal to a basic term.

**Proof.** Assume a ground normal form \(q\) of \(R_0\). It follows from the rewrite rules in \(R_0\) that \(q\) is a sum of terms of the form \(a\mu\) or \(aq'\) or \(a^*q'\), with \(q'\) a ground normal form, where \(a^*q'\) does not occur as an argument of alternative composition nor of prefix iteration. In other words, \(q\) is of the form either \(\sum_{i=1}^{k} a_iq_i + \sum_{j=1}^{l} \mu_j\) or \(a^*(\sum_{i=1}^{k} a_iq_i + \sum_{j=1}^{l} \mu_j)\), where the terms \(q_i\) are ground normal forms. By induction on size, i.e. on the number of function symbols in \(q\), we may assume that the ground normal forms \(q_i\) are provably equal to basic terms.

The ground normal form \(q\) can be adapted to a basic term by means of the axioms as follows. Remove all occurrences of summands \(\delta\) (axiom A6), remove double occurrences of summands \(\epsilon\) (axiom A3), add one summand \(\delta\) if there is no summand \(\epsilon\) present (axiom A6), and replace summands \(a\) by \(a\epsilon\) (axiom A8).

According to Theorem 2.2 each process term is provably equal to a ground normal form of \(R_0\). Since each ground normal form of \(R_0\) is provably equal to a basic term, it follows that each process term is provably equal to a basic term. \(\square\)

The following lemma stems from Aceto and Ingólfsdóttir (in the setting of CCS).

**Lemma 2.5** If \(a^*p \leftrightarrow b^*q\), then \(a = b\).

**Proof sketch.** If \(a^*p \leftrightarrow b^*q\), then it follows that \(a^*p\) exhibits the infinite trace of actions \((ab)^\omega\). Thus, this lemma is an immediate consequence of the following fact.
• If \( p_n \xrightarrow{a_n} p_{n+1} \) for \( n = 0, 1, 2, \ldots \), then there is an \( N \) such that \( a_n = a_N \) for \( n > N \).

The proof of this fact is an easy exercise by structural induction on terms, which is left to the reader. \( \square \)

**Theorem 2.6** The axiomatization A1-9 + M1-5 for BPA\(^{\pi}\) is complete with respect to bisimulation equivalence.

**Proof.** According to Corollary 2.4, each process term is provably equal to a basic term. Therefore it is sufficient to show that bisimilar basic terms are provably equal. We deduce the following statement \( I_n \) by induction on \( n \).

\( I_n \) If \( p \) and \( q \) are bisimilar basic terms with \( \text{size}(p) + \text{size}(q) = n \), then \( p = q \).

Suppose that we have already proved \( I_n \) for \( n < N \), and let \( p \) and \( q \) are bisimilar basic terms with \( \text{size}(p) + \text{size}(q) = N \). We prove that \( p = q \), in three distinct cases, which distinguish the possible forms of \( p \) and \( q \).

1. \( p = \text{AC} \sum_i a_i p_i + \xi \) and \( q = \text{AC} \sum_j b_j q_j + \xi' \).

\( p \leftrightarrow q \), so \( p \downarrow \) if and only if \( q \downarrow \). Hence, \( \xi \) represents \( \epsilon \) if and only if \( \xi' \) represents \( \epsilon \), so \( \xi = \xi' \).

Since \( p \xrightarrow{a_i} p_i \), and since \( p \leftrightarrow q \), there is a transition \( q \xrightarrow{b_k} q' \) where \( p \leftrightarrow q' \).

Thus, \( b_k = a \) and \( p \leftrightarrow q_k \) for some \( k \). Since \( \text{size}(q_k) < \text{size}(q) \), the induction hypothesis \( I_{N-1} \) yields \( p = q_k \). Hence, \( ap = bkq_k \).

Since \( p \xrightarrow{a_i} p_i \), and since \( p \leftrightarrow q \), there is a transition \( q \xrightarrow{a_i} q' \) where \( p_i \leftrightarrow q' \).

Hence, \( b_k = a_i \) and \( p_i \leftrightarrow q_k \) for some \( k \). By induction \( p_i = q_k \), so \( a_i p_i = b_k q_k \).

Thus, each summand of \( ap + \sum_i a_i p_i + \xi \) is provably equal to a summand of \( q \). By the symmetric argument, we find that each summand of \( q \) is provably equal to a summand of \( ap + \sum_i a_i p_i + \xi \). Hence, \( p \equiv_{\text{M1}} ap + \sum_i a_i p_i + \xi = q \).

2. \( p = \text{AC} \sum_i a_i p_i + \xi \) and \( q = \text{AC} \sum_j b_j q_j + \xi' \).

In this case, we can repeat the argument of the previous case to find that each summand of \( p \) is provably equal to a summand of \( q \), and vice versa. Hence, \( p = q \).

3. \( p = \text{AC} \sum_i a_i p_i + \xi \) and \( q = \text{AC} b^* (\sum_j b_j q_j + \xi') \).

By symmetry, we may assume that \( \text{size}(p) \geq \text{size}(q) \). Since \( p \leftrightarrow q \), Lemma 2.5 yields \( a = b \).

\( p \leftrightarrow q \), so \( p \downarrow \) if and only if \( q \downarrow \). Hence, \( \xi \) represents \( \epsilon \) if and only if \( \xi' \) represents \( \epsilon \), so \( \xi = \xi' \).

We distinguish three cases.

• \( q_j \leftrightarrow p \) for some \( j \in J \).

Induction yields \( q_j = p \). Moreover, \( q_j \leftrightarrow p \leftrightarrow q \) and \( \text{size}(q) \leq \text{size}(p) \), so induction yields \( q_j = q \). Hence, \( p = q_j = q \).
• For each $i \in I$, $a_i \neq a$ or $p_i \not\leftrightarrow q$, and for each $j \in J$, $q_j \not\leftrightarrow p$.

In this case, each transition $p \xrightarrow{a_i} p_i$ of $p$, for $i \in I$, can only be mimicked by a transition $q \xrightarrow{b_j} q_j$ of $q$ for some $j \in J$, and vice versa. So by induction each summand $a_i p_i$ for $i \in I$ is provably equal to a summand $b_jq_j$ for $j \in J$, and vice versa. Hence, $p =_{AC} a^*(\sum_{i \in I} a_i p_i + \xi) =_{AC} a^* q$.

• $p_i \leftrightarrow q$ and $a_i = a$ for some $i \in I$, and for each $j \in J$, $q_j \not\leftrightarrow p$.

Abbreviate $\sum_{j \in J} b_jq_j + \xi'$ to $q'$. 

Let $I_0$ be the non-empty subset of elements $i \in I$ for which $p_i \leftrightarrow q$ and $a_i = a$. For $i \in I_0$, induction yields $p_i = q$, so $a_i p_i = a q =_{AC} a \cdot a^* q'$. Hence, $\sum_{i \in I} a_i p_i = a \cdot a^* q' + \sum_{i \in I \setminus I_0} a_i p_i$.

Each transition $p \xrightarrow{a_i} p_i$ of $p$ for $i \in \setminus I_0$ can only be mimicked by a transition $q \xrightarrow{b_j} q_j$ of $q$ for some $j \in J$, and vice versa. So by induction each summand $a_i p_i$ for $i \in \setminus I_0$ is provably equal to a summand $b_jq_j$ for $j \in J$, and vice versa. Hence, $p =_{AC} a^* (\sum_{i \in I_0} a_i p_i + \sum_{i \in \setminus I_0} a_i p_i + \xi) =_{AC} a^*(a \cdot a^* q' + \xi') =_{AC} a^*(a^* q') =_{AC} a^* q'$.

2.4 Complete axiomatizations for subalgebras

With a similar proof scheme as has been used for the completeness proof for $BPA^*_{\delta_\varepsilon}$, with the cases for $\delta$ and/or $\varepsilon$ omitted, we obtained the following results.

**Theorem 2.7** The axiomatization $A1-5 + MI1-3$ for $BPA^*_{\delta_\varepsilon}$ is complete with respect to bisimulation equivalence.

**Theorem 2.8** The axiomatization $A1-7 + MI1-4$ for $BPA^*_{\delta}$ is complete with respect to bisimulation equivalence.

**Theorem 2.9** The axiomatization $A1-5,8,9 + MI1-3,5$ for $BPA^*_{\delta_\varepsilon}$ is complete with respect to bisimulation equivalence.

3 Normalization of the Rewrite System $R_0$

In this section we deduce normalization of the rewrite system $R_0$ in Table 3. In the proof of this result we will use termination modulo $AC$ of the + of the rewrite system that consists of the rules 8 and 9 in $R_0$. This fact can be established by the technique of dummy elimination from Ferreira and Zantema (1994), which enables to remove a function symbol from a rewrite rule if it does not occur at the left-hand side of the rewrite rules. In the case of rules 8 and 9, dummy elimination can be applied with respect to sequential composition.

3.1 Dummy elimination

Suppose that a function symbol $f$ occurs only at the right-hand side of the rewrite rules in a rewrite system $R$. Dummy elimination enables to remove $f$ from $R$.
as follows. First, introduce a new constant $\Delta$. If a rule in $R$ is of the form $t \rightarrow C[f(t_1, \ldots, t_n)]$, where $C[]$ is some context, then replace this rule by the following collection of rules:

\[
\begin{align*}
  t &\rightarrow C[\Delta] \\
  t &\rightarrow t_1 \\
  \vdots \\
  t &\rightarrow t_n.
\end{align*}
\]

Repeat this procedure until all occurrences of $f$ in $R$ have been removed. Let $E(R)$ denote the resulting rewrite system. The main theorem of Ferreira and Zantema (1994) says that termination of $E(R)$ implies termination of $R$.

**Definition 3.1** An equation $s = t$ is called **linear** if both $s$ and $t$ do not contain double occurrences of variables.

An equation $s = t$ is called **non-erasing** if both $s$ and $t$ contain exactly the same variables.

A set of equations is called linear or non-erasing respectively if all its equations are so. Dummy elimination applies to rewriting modulo a set of equations, under the condition that this set of equations is linear and non-erasing, see Ferreira (1995). Note that the equations for associativity and commutativity of the $+$ are indeed both linear and non-erasing.

### 3.2 Termination of rules 8 and 9

\[
\begin{align*}
  a\star x + y &\rightarrow \Delta + x + y \\
  a\star x + y &\rightarrow a \\
  a\star x + y &\rightarrow a\star x \\
  b\star (a\star x) &\rightarrow b\star (\Delta + x) \\
  b\star (a\star x) &\rightarrow a \\
  b\star (a\star x) &\rightarrow a\star x
\end{align*}
\]

Table 4: Rules 8 and 9 after dummy elimination

In the case of rules 8 and 9, dummy elimination can be applied to remove the sequential composition. The rewrite system which results after dummy elimination is presented in Table 4. Define a weight function as follows.

\[
\begin{align*}
  w(\Delta) &= 1 \\
  w(\mu) &= 1 \\
  w(p + q) &= w(p) + w(q) \\
  w(pq) &= w(p) + w(q) \\
  w(p\star p) &= w(p) + 2.
\end{align*}
\]

The weight of terms strictly decreases under application of the rewrite rules. Note that terms which are equal modulo AC of the $+$ have the same weight. Hence, the
rewrite system in Table 4 is terminating modulo AC of the +. So rules 8 and 9 are also terminating modulo AC of the +.

3.3 Normalization of \( R_0 \)

**Lemma 3.2** The rewrite system \( R_0 \) is normalizing.

**Proof.** We split up \( R_0 \), namely we separate rules 8 and 9 from rules 1-7. In the previous section we proved termination of rules 8 and 9. Termination of rules 1-7 follows trivially by defining an appropriate weight function:

\[
\begin{align*}
    w(\mu) &= 2 \\
    w(p + q) &= w(p) + w(q) \\
    w(pq) &= w(p)^2 w(q) \\
    w(\mu^* p) &= w(p) + 1.
\end{align*}
\]

Consider a process term \( p \). Since rules 1-7 are terminating, they reduce \( p \) to a process term \( p' \) which does not reduce any further. Since rules 8,9 are terminating, they reduce \( p' \) to a process term \( p'' \) which does not reduce any further. It is easy to see that \( p'' \) is a ground normal form of \( R_0 \). \( \square \)

4 Termination of the Rewrite System \( R_0 \)

In this section we prove that the rewrite system \( R_0 \) in Table 3 is terminating modulo AC of the +, which means that \( R_0 \) does not allow any infinite reductions. Note that the rewrite rules 8 and 9 are self-embedding: their left-hand sides can be embedded in the corresponding right-hand sides. Hence, it is not possible to prove termination of \( R_0 \) neither by means of a recursive path ordering nor by a compositional weight function in the naturals, see e.g. Zantema (1994).

In order to prove termination of \( R_0 \) modulo AC of the +, we apply a technique from Zantema and Geser (1995), based on an abstract commutation criterion. Since we will need that result in the more general setting of rewriting modulo equations, in Section 4.1 we generalize the abstract commutation criterion accordingly. Next, in Section 4.2 we describe how this applies to term rewriting. Finally, in Section 4.3 we use this technique to prove termination of \( R_0 \) modulo AC of the +.

4.1 Abstract termination

The proof that the rewrite system \( R_0 \) is terminating modulo AC of the +, which is presented in Section 4.3, is based on a theorem that can be given in a very general abstract setting. This theorem, which is a generalization of a result from Zantema and Geser (1995) to the setting of rewriting modulo equations, will be presented and proved in this section.

Let \( R, S, T, E \) denote binary relations on a fixed set \( V \). We write a dot symbol for relational composition, i.e. one has \( t(R.S)t' \) if and only if there exists \( t'' \) such that \( tRt'' \) and \( t''St' \). We write \( R^+ \) for the transitive closure of \( R \) and \( R^* \) for the reflexive
transitive closure of $R$. Further we write $R \subseteq S$ if $tRt'$ implies $tSt'$. Clearly, if $R \subseteq S$ then $RT \subseteq ST$ and $TR \subseteq TS$.

We write $\infty(t, R)$ if there exists an infinite sequence $t_1, t_2, t_3, \ldots$ such that $t = t_1$ and $t_iRt_{i+1}$ for all $i = 1, 2, 3, \ldots$. A relation $R$ is called terminating if there does not exist any term $t$ satisfying $\infty(t, R)$.

In the following lemma we collect some standard properties for relations, which are easy to check.

**Lemma 4.1**

1. If $R.S \subseteq S*.R$, then $R.S^* \subseteq S*.R$.
2. If $R.S \subseteq S.R^*$, then $R^*.S \subseteq S.R^*$.
3. If $R.S \subseteq T^+.R$ and $t'Rt$ and $\infty(t, S)$, then $\infty(t', T)$.

For relations $R, E$ we write $R/E$ for $E^*.R.E^*$. The intuition here is that the reduction relation $R$ is taken modulo equations $E$. However, we do not need that $E$ is symmetric, hence our theorem is even on relative termination which is more general than modulo an equivalence. Now we state and prove our abstract termination theorem.

**Theorem 4.2** Let $R, S, T, E$ be binary relations satisfying

1. $S/E$ is terminating,
2. $R \subseteq S^+.T^*$,
3. $T.R \subseteq R^+.T^*$,
4. $T.E \subseteq E^*.T$.

Then $R/E$ is terminating.

**Proof.** From condition 4 and the first item of Lemma 4.1 we conclude $T.E^* \subseteq E^*.T$.

From this and condition 3 we conclude:

$$(T/E).(R/E) = E^*.T.E^*.R.E^*$$

$${} \subseteq E^*.E^*.T.R.E^*$$

$${} = E^*.T.R.E^*$$

$${} \subseteq E^*.R^+.T^*.E^*$$

$${} \subseteq E^*.R^+.R.(R \cup T)^*.E^*$$

$${} = E^*.R.(R \cup T)^*.E^*$$

$${} \subseteq (R/E).((R \cup T)/E)^*.$$

Since also $(R/E).(R/E) \subseteq (R/E).((R \cup T)/E)^*$, we obtain

$$(R \cup T)/E).(R/E) = (R/E).(R/E) \cup (T/E).(R/E) \subseteq (R/E).((R \cup T)/E)^*.$$
From the second item of Lemma 4.1 and condition 2 we conclude

\[
((R \cup T)/E)^*.(R/E) \subseteq (R/E).((R \cup T)/E)^* \\
= E^*.R.E^*.((R \cup T)/E)^* \\
\subseteq E^*.S^+.T^*.E^*.((R \cup T)/E)^* \\
= E^*.S^+.((R \cup T)/E)^* \\
\subseteq (S/E)^+.((R \cup T)/E)^*
\]

Assume that \(R/E\) does not terminate. Then there exists an element \(t\) with \(\infty(t.R/E)\). Clearly \(t((R \cup T)/E)^*t\), hence the third item of Lemma 4.1 yields \(\infty(t.S/E)\). This contradicts condition 1. \(\square\)

To stress the subtlety of this theorem, we show that condition 4 may not be weakened to \(T.E \subseteq E^*.T^+\).

**Example 4.3** Let \(V = \{1, 2, 3, 4\}\) and

\[
\begin{align*}
1R3, \\
1S4, \\
4T3 & 3T2 & 1T1 & 2T2 & 3T3, \\
1E2 & 2E1 & 2E3 & 3E2.
\end{align*}
\]

\(S/E\) is terminating, because \(S\) consists only of \(1S4\), and 4 cannot be reduced by \(S\) nor by \(E\). The relation inclusions in conditions 2 and 3 in Theorem 4.2 and \(T.E \subseteq E^*.T^+\) are easily checked.

\[
\begin{align*}
R & \subseteq S^+.T^* : 1R3 & 1S4T3. \\
T.R & \subseteq R^+.T^* : 1T1R3 & 1R3. \\
T.E & \subseteq E^*.T^+ : 4T3E2 & 4T3T2 \\
 & 3T2E1 & 3E2E1T1 \\
 & 3T2E3 & 3T3 \\
 & 1T1E2 & 1E2T2 \\
 & 2T2E1 & 2E1T1 \\
 & 2T2E3 & 2E3T3 \\
 & 3T3E2 & 3T2.
\end{align*}
\]

However, \(R/E\) is not terminating: \(1R3E2E1R3 \ldots\).

### 4.2 Application to rewrite systems

Before applying Theorem 4.2 to rewrite systems, first we recall some standard terminology from term rewriting. See e.g. Klop (1992) for an overview of the field of term rewriting.

**Definition 4.4**

- A rewrite rule \(l \rightarrow r\) is called left-linear if each variable occurs at most once in \(l\).
A rewrite rule \( l \rightarrow r \) is called non-erasing if each variable in \( l \) also occurs in \( r \).

A rewrite system is called left-linear or non-erasing respectively if all its rules are so.

Recall that an equation \( s = t \) is called linear if both \( s \rightarrow t \) and \( t \rightarrow s \) are left-linear, and that \( s = t \) is called non-erasing if both \( s \rightarrow t \) and \( t \rightarrow s \) are non-erasing (see Definition 3.1). In particular, both associativity and commutativity are linear and non-erasing.

Theorem 4.2 can be applied to prove termination of rewrite systems modulo equations. Then for \( R \) one chooses the rewrite relation of a rewrite system (also called \( R \)) for which termination has to be proved modulo some equations. For \( E \) one chooses the 'one-step equalities' corresponding to these equations. For \( S \) one chooses an adaptation of \( R \) for which termination modulo the equations can be proved. Then condition 1 of Theorem 4.2 is fulfilled. For each rule \( l \rightarrow r \) of \( R \) there is to be a rule \( l \rightarrow r' \) in \( S \) such that \( r \rightarrow_A r' \) for some auxiliary rewrite system \( A \). For the relation \( T \) one chooses the inverse of the rewrite relation of \( A \), so that condition 2 of Theorem 4.2 is also fulfilled. Now condition 3 reads:

\[
\text{if } t \rightarrow_A t' \text{ and } t \rightarrow_R t'', \text{ then there exists a } u \text{ for which } t' \rightarrow_R^+ u \text{ and } t'' \rightarrow_A^* u.
\]

If \( A \) is left-linear and non-erasing, and if \( R \) is left-linear, then this requirement is always fulfilled for non-overlapping redexes. Hence, this condition can be verified by a finite analysis of overlapping redexes. In the typical case, in the first attempt for \( A \) the condition does not hold, and \( A \) has to be extended a number of times to obtain condition 3. This is a kind of completion, similar to what is done in Bellegarde and Lescanne (1990). In the application of Theorem 4.2 in this paper, the rewrite systems \( R \) and \( S \) are also extended during this completion, and the final auxiliary rewrite system \( A \) has infinitely many rules.

Finally, for the system \( E \) of equations, condition 4 of Theorem 4.2 reads:

\[
\text{if } t \rightarrow_A t' \text{ and } t \leftrightarrow_E t'', \text{ then there exists a } u \text{ for which } t' =_E u \text{ and } t'' \rightarrow_A u.
\]

Here \( \leftrightarrow_E \) denotes 'one-step equalities' corresponding to \( E \), while \( =_E \) denotes \( \leftrightarrow_E^* \), being the generated congruence. If \( E \) is linear and non-erasing, and if \( A \) is left-linear, then this condition can be verified by a finite analysis of overlapping redexes, similar as for condition 3.

### 4.3 Termination of \( R_0 \)

The intuition behind the termination proof of \( R_0 \) is that the expansion from a pattern \( a^*p \) to \( a \cdot a^*p + p \), as is done by rules 8 and 9, can occur at most only once for every occurrence of a prefix iteration symbol. We formalize this as follows. Extend the signature with unary function symbols \( a^#_a \) for \( a \in A \). Intuitively, these new function symbols will be used to register that the expansion from \( a^*p \) to
a·a*p + p has been done. In a first attempt to apply Theorem 4.2, we choose R to be $R_0$, and S to be the simplified variant of $R$ in which the patterns $a·a*p + p$ in the right-hand sides of rules 8 and 9 have been replaced by $a·a#p + p$. Furthermore, we choose $E$ to be the AC rules for $+$. It is not difficult to see that $S/E$ is terminating. As a first obvious try, we choose $A$ to consist of the rule

$$a(a^*x) \rightarrow a(a#x).$$

Now condition 2 is easily checked, but condition 3 does not yet hold. Therefore we extend the systems $A$, $R$ and $S$ with some new rules, triggered by the desired validity of condition 3. This process of completion ends in the following choices for the systems $A$, $R$ and $S$.

For the rewrite system $R$ we choose the original system $R_0$ extended by the following two new rules:

10. $(a#x)y \rightarrow a#(xy)$
11. $a#(b^*x) \rightarrow a#(b·b#x + x)$.

Since rewrite rules are applied modulo AC of the $+$, we take $E = \{x + y = y + x, (x + y) + z = x + (y + z)\}$. We shall apply Theorem 4.2 yielding termination of $R/E$, which immediately implies termination of $R_0$ modulo AC of the $+$.

The rewrite system $S$ is obtained by a slight modification of $R$ in which the patterns $a·a*p + p$ as they appear in the right-hand sides of rules 8, 9 and 11 have been replaced by $a·a#p + p$. That is, $S$ consists of the rules 1-7, 10 and

9'. $a^*(b^*x) \rightarrow a^*(b·b#x + x)$
11'. $a#(b^*x) \rightarrow a#(b·b#x + x)$.

Finally, we define the rewrite system $A$ to be the following infinite collection of rewrite rules:

$$\begin{align*}
    r_0 & : a(a^*x) \rightarrow a(a#x) \\
    r_1 & : a((a^*x)y_0) \rightarrow a((a#x)y_0) \\
    r_2 & : a(((a^*x)y_0)y_1) \rightarrow a(((a#x)y_0)y_1) \\
    & \vdots
\end{align*}$$

More precisely, $A$ consists of rewrite rules $r_i$ of the form $a·C_i[a^*x] \rightarrow a·C_i[a#x]$ for $i \geq 0$, where the contexts $C_i[]$ are defined inductively by

$$C_0[] = [], \quad C_{i+1}[] = C_i[]·y_i,$$

with $y_i$ a fresh variable. Equivalently, one can say that $r_i$ is of the form $a·D_i[a^*x] \rightarrow a·D_i[a#x]$, where the contexts $D_i[]$ are defined inductively by

$$D_0[] = [], \quad D_{i+1}[] = D_i[]·z_i,$$

with $z_i$ a fresh variable. We will need both representations of $r_i$ later on.

**Theorem 4.5** The rewrite system $R_0$ is terminating modulo AC of the $+$. 
Proof. We prove that the rewrite system $R$ is terminating modulo AC of the $+$, by verifying the four conditions of Theorem 4.2. Since $R$ incorporates $R_0$, this result yields that $R_0$ is terminating modulo AC of the $+$.

1. $S/E$ is terminating.

Define the following weight function on terms.

\[
\begin{align*}
w(\mu) &= 2 \\
w(p + q) &= w(p) + w(q) \\
w(pq) &= w(p)^2 w(q) \\
w(a^*p) &= 5w(p) + 5 \\
w(a^#p) &= w(p) + 1
\end{align*}
\]

Note that terms which are equal modulo AC of the $+$ have the same weight. It is easy to see that the weight of terms strictly decreases under application of rules in $S$. Hence, $S/E$ is terminating.

2. For each rule $l \rightarrow r$ of $R$ there is a rule $l \rightarrow r'$ in $S$ such that $r \rightarrow_A^* r'$.

Only rules 8, 9 and 11 in $R$ have been adapted in $S$. For these three rules in $R$, the rule $r_0$ in $A$, $a(a^*x) \rightarrow a(a^#x)$ can be applied to obtain the corresponding right-hand sides in $S$.

The other rules in $R$ and $S$ coincide, so for those rules we can take $r$ for $r'$.

3. If $t \rightarrow_A t'$ and $t \rightarrow_R t''$, then there exists a $u$ for which $t' \rightarrow_R^* u$ and $t'' \rightarrow^*_A u$.

Note that $A$ is left-linear and non-erasing, and that $R$ is left-linear.

A straightforward analysis of overlapping redexes learns that there are three types of overlaps between a left-hand side of $A$ and a left-hand side of $R$, which involve rules 2, 5 and 9 in $R$ respectively. We treat these three cases separately.

(a) A term $(a \cdot C_i[a^*x]) \cdot y_i$ can be reduced by rule $r_i$ in $A$ and by rule 2 in $R$. This overlapping redex is convergent, owing to rule $r_{i+1}$ in $A$.

\[
\begin{align*}
(a \cdot C_i[a^*x]) \cdot y_i \quad &\rightarrow r_i \quad 2 \\
&\rightarrow (a \cdot C_i[a^#x]) \cdot y_i \\
&\quad \downarrow 2 \\
&\quad r_{i+1} \\
&\rightarrow a \cdot (C_i[a^*x] \cdot y_i) = a \cdot C_{i+1}[a^*x] \\
\end{align*}
\]
in SDL. The SDL expressions **parent**, **offspring** and **sender** are regarded as variables accesses, and variable accesses are treated as a special case of view expressions. The sixth case includes **self**, which corresponds to the SDL expressions **self**.

The remaining five cases are needed to reflect the intended meaning of various other SDL construct exactly. The expression **cnt** is used to associate a unique pid value with each created process. Expressions of the form **waiting**(s₁, ..., sₙ, u) are used to give meaning to SDL’s state definitions. They are needed to model that signal consumption is not delayed till the next time slice when there is a signal to consume. Expressions of the forms **type**(u) and **hasinst**(X) are used to give meaning to SDL’s output actions. They are needed to check (dynamically) if a receiver with a given pid value is of the appropriate type for a given signal route and to check if a receiver of the appropriate type for a given signal route exists. Expressions of the form u₁ = u₂ are, as a matter of course, used to give meaning to SDL’s decisions. Furthermore, they are used with expressions of the form **cnt** or **type**(u) as left-hand sides where the latter expressions are used.

The state transforming actions are parametrized by several domains that are built on **VExpr**:

<table>
<thead>
<tr>
<th>Domain</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>SigDₖ</td>
<td>Sₖ × <strong>VExpr</strong>ₖ *</td>
</tr>
<tr>
<td>ExtSigDₖ</td>
<td>SigDₖ × <strong>NExpr</strong>ₖ × <strong>NExpr</strong>ₖ</td>
</tr>
<tr>
<td>ExtSigPₖ</td>
<td>(Sₖ × Vₖ * × (nil) × <strong>NExpr</strong>ₖ</td>
</tr>
</tbody>
</table>

The domains **SigD** and **ExtSigD** are like **Sig** and **ExtSig**, respectively, but with **U** and **N** replaced by **VExpr** and **NExpr**, respectively. The domain **ExtSigP** differs slightly from **ExtSigD**, because it represents signal patterns, with variables used for the unknown values and **nil** for “don’t care”.

The following state transforming actions are used:

- **input** : ExtSigPₖ × P₅ₛₖ(Sₖ)
- **output** : ExtSigDₖ × (Cₖ ∪ {nil}) × **NExpr**ₖ
- **set** : **NExpr**ₖ × SigDₖ × **NExpr**ₖ
- **reset** : SigDₖ × NExprₖ
- **ass** : Vₖ × **VExpr**ₖ × **NExpr**ₖ
- **τf** : Pₖ × Vₖ * × **VExpr**ₖ * × (**NExpr**ₖ ∪ {nil})
- **stop** : **NExpr**ₖ
- **inispont** : **NExpr**ₖ

These are the ACP actions that correspond to input guards, SDL actions, **stop** and **input none**. The second parameter of an **input** action is the save set being in force. The third parameter of an **output** action denotes the delay that the signal experiences if it must pass through a channel. The last parameter of the remaining actions denotes the pid value of the process from which the action originates. Recall that the second and third parameter of an **τf** action are the formal parameters and the actual parameters, respectively, of the process to be created. The presence of **nil** needs some further explanation. The second parameter of an **output** action is a channel if the signal to be sent must pass through a channel, and **nil** otherwise. The last parameter of a **τf** action is a


A verification of the Bakery Protocol combining algebraic and model-oriented techniques

Claudia Brovedani
Via Boselli 7/9, 16146 Genova, Italy

Steven Klusener
CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
e-mail: stevenk@cwi.nl

Abstract
In this paper we give a specification of the so called Bakery protocol in an extension of the proces algebra ACP with abstract datatypes. We prove that this protocol is equal with a Queue, modulo branching bisimulation equivalence.

The verification is as follows. First we give a linear specification of the Bakery, that is a specification without parallelism. Then we introduce an invariant and encorporate this invariant into the linear specification of the Bakery and the specification of the Queue. Finally, we give a boolean function on the arguments of the resulting specification of the Bakery and the Queue, and we prove that by its equations it defines a branching bisimulation.

This paper can be considered as an alternative to the proof of Groote and Korver [GK94], that proves the correctness of the Bakery protocol modulo weak bisimulation (or observational congruence) completely within the proof system of μCRL.

1985 Mathematics Subject Classification: 68Q10, 68Q40, 68Q45, 68Q55.
Key Words & Phrases: process algebra, ACP, protocol verification, branching bisimulation.
Note: While writing this paper the first author stayed at CWI, funded by scholarships of the University of Turino and CNR.

Introduction

Process algebras, such as ACP [BK84, BW90] and CCS [Mil89], provide a simple but expressive framework for specifying and analyzing distributed systems, see for example [Bae90]. However, in their pure form these process algebras are not very well suited for dealing with systems in which the process behavior depends on the underlying data. Therefore, Groote and Ponse have developed the language μCRL[GP94], a combination of ACP with abstract datatypes [EM85]. For this language they also gave a proof system [GP91].

In order to exercise with different proof methods and techniques, that concern processes with data, various verifications of the so called Bakery protocol have been given. This protocol requires that each customer takes a number when entering the bakery shop, and that the baker serves the customers in order of their number. This protocol is correct as the customers are served in order of their entrance, in other words, the external behavior of the protocol corresponds with a queue. Although this seems
rather obvious this paper, and others, show that a detailed analysis of the correctness depends subtly on a mixture of process behavior and data properties.

In [GK94] Groote and Korver have proven the bakery protocol purely algebraically, using the axioms of $\mu$CRL, the proofrule RSP and the $r$-laws for weak bisimulation (which is a more identifying equivalence than branching bisimulation). Lately Griffioen and Korver have proven the bakery protocol in the I/O-automata model [GK95], this proof has been checked formally within the proof system LP.

When Groote and Korver started their work on the verification of the bakery protocol in $\mu$CRL, it was not completely clear how it could be done. For example, it was not yet known how one could deal with invariants, as it has been proposed later by Bezem and Groote in [BG93]. Therefore, Jan Bergstra suggested to try to verify the bakery protocol in a more model oriented way. In this paper this idea is worked out in detail.

Our verification consists of the following steps:

- First we define the bakery protocol formally; it is a parallel composition of an in-counter, a shop-floor (modelled as a bag), and the serve desk, or out-counter. Moreover, we define a queue process. The goal of the paper is to prove that these two processes are equal, modulo branching bisimulation.

- Then, we give a linear specification of the bakery protocol, i.e., a process definition without parallelism. This definition is based on the four states in which the bakery can evolve, similar definitions of these four states can be found in [GK94] as well. This definition contains a rather complex sum, based on the fact that if the baker asks for the next customer, he asks for some customer that holds a label that corresponds with his counter.

- We introduce a so called invariant that formalizes the intuition that says that there will be exactly a unique customer with that label. We encorporate this invariant into the linear specification of the bakery, by which the above mentioned sum can be simplified.

- In order to relate the resulting definition with the one of the queue, we encorporate the invariant into the definition of the queue as well.

- Finally we define a relation as a boolean funtion and we show that it can be considered as a branching bisimulation.

So, we do not use the axioms for branching bisimulation, nor the proof rule RSP.

The language we use consists of ACP constructs and a prefix summation, it is derived from real time ACP with prefix summation [FK], and it can be considered as a subcalculus of $\mu$CRL.

Acknowledgements

The first author wants to thank the CWI, and Jan Willem Klop in particular, for the offered hospitality. Furthermore, both authors want to thank Jan Willem Klop, Henri Korver, and Alban Ponse for the technical discussions on earlier versions of this paper.

Jan Bergstra is acknowledged for providing the original research proposal.
1 Some remarks on the syntax of ACP with prefix Summation

1.1 The Datatype part

The datatypes that we deal with are defined in detail in the Appendix, Section B. Their definitions are given in a μCRL style, and their standard parts are taken from [GvW94], including the modulo arithmetic of the natural numbers.

In this paper we will refer to some abstract datatype D, with typical variable d, that models the set of customers. For technical reasons we assume that D contains at least the bottom element ⊥, i.e., a customer that is not allowed in the bakery. A pair d_i models a customer d that holds a label i, and it is called a frame. The datatype of frames is denoted by Frame, with typical variable f, and the datatype of bags of frames is denoted by FBag, with typical variable b and emptybag θBag.

Nat denotes the standard datatype of natural numbers, +_n denotes addition modulo n. We have also the standard datatype of the booleans, denoted by Bool, with constants t (true) and f (false), and operators ∧ and ∨. Over this datatype we have expressions like i < n and size(b) ≤ n. An arbitrary expression of type Bool is denoted by φ. The boolean expression test(f, b) is true if the frame f is in the bag b.

All the datatypes S are provided with an equality function S × S → Bool, for which holds that eq(s, s') = t ⊸ s = s'. Hence, we may allow ourselves to write the boolean expression eq(s, s') by s = s'.

1.2 The Process part

For simplicity we restrict ourselves to prefix summation, so if x is a process term (in which the data variable v may occur), then ∑_v:φ a(v) · x is a process term as well. The set of process terms is defined by the following BNF sentence:

\[
x ::= δ \mid ∑_φ a(v) · x \mid x + x \mid φ → x \mid x || x \mid x|x \mid x|H(x) \mid H(x)
\]

were v is a data variable of some type, φ is an expression of type Bool, and H is a set of unparameterized actions.

The sum construct ∑_φ a(v) · x binds all occurrences of the data variable v in x. For some ground \(^1\) data expression e of the same type, it can execute an action a(e) and evolve into x[e/v], (e substituted for v), for every context in which φ[e/v] is true. Hence, ∑_φ a(v) · x can be considered as an abbreviation of the μCRL expression \(∑_v:S a(v) · x ∨ φ ⇒ δ\), where S is the type of v. In some cases, for example if φ is a rather large expression, we write ∑_φ:φ a(v) · x for ∑_φ a(v) · x.

The process φ → x is enabled only in contexts in which φ is true. Again, φ → x can be considered as an abbreviation of the μCRL expression x ∨ φ ⇒ δ.

This calculus is denoted by ACP_pS, the suffix pΣ stands for prefix summation, and as shown before it can be considered as a subcalculus of μCRL. The axioms for ACP_pS are given in the Appendix, Section C.

\(^1\)i.e., not containing any data variables

159
2 The Bakery Protocol

The Bakery protocol models the counter/serving process in a bakery; each customer is provided with a label, or index, \( i \) and customers are being served by the baker in order of their label. The protocol is parameterized with a natural number, \( n \), that defines its capacity; i.e., the maximal numbers of customers the shop floor can contain.

The description of the Bakery protocol, or \( B_n \), involves three processes: the "in-counter", or \( IN_n(i : \text{Nat}) \), that provides a new customer with a label \( i \), the "out-counter", or \( OUT_n(j : \text{Nat}) \), that models the baker serving the customer with label \( j \), and finally we have the clients with their label that are waiting on the shop floor, modelled by the process \( Pbagn(b : \text{FBag}) \). Initially both counters are zero and the shop floor is empty.

\[
\begin{align*}
B_n & := \tau_{(c_1,c_2)}(\mathcal{B}_{(s_1,s_2,r_1,r_2)}(IN_n(0) \parallel Pbagn(\emptyset_{\text{FBag}}) \parallel OUT_n(0))) \\
IN_n(i : \text{Nat}) & := \sum_{d : d \neq \emptyset} \text{enter}(d) \cdot \sigma_1(d') \cdot IN_n(i + n 1) \\
OUT_n(j : \text{Nat}) & := \sum_{f : \text{index}(f) = j} \tau_2(f) \cdot \text{out}(\text{data}(f)) \cdot OUT_n(j + n 1) \\
Pbagn(b : \text{FBag}) & := \begin{cases} 
\text{size}(b) \neq n : \rightarrow \sum_{f : \text{test}(f,b)} r_1(f) \cdot Pbagn(\text{add}(f,b)) \\
+ \sum_{f : \text{test}(f,b)} s_2(f) \cdot Pbagn(\text{rem}(f,b)) 
\end{cases}
\end{align*}
\]

The goal of introducing such a label/countering mechanism into a bakery is of course that customers are served properly in the order of their entrance. In fact, that is exactly what we are going to prove in this paper. To be able to state this properly, we define \( \text{DQueue} \), the datatype of queues of \( D \), see for its definition the Appendix, Section B.6. A process that behaves like a queue with capacity \( m \) is denoted by \( Q_m \) and is defined below.

\[
\begin{align*}
Q_m & := Q_m(\emptyset_{\text{dqueue}}) \\
Q_m(q : \text{DQueue}) & := \begin{cases} 
\text{size}(q) \neq m : \rightarrow \sum_{d : \neq \emptyset} \text{enter}(d) \cdot Q_m(\text{add}(d,q)) \\
+ \text{size}(q) \neq 0 : \rightarrow \text{out}(\text{top}(q)) \cdot Q_m(\text{untop}(q)) 
\end{cases}
\end{align*}
\]

So, we will prove that the Bakery equals a Queue. As the Bakery has two positions, one at the in-counter, and one at the out-counter, and a shop floor with capacity \( n \), we will prove that \( B_n \) equals \( Q_{n+2} \). The Bakery involves certain so called internal actions, both communication actions \( c_1 \) and \( c_2 \) (that model the transfer of customers from the in-counter to the shop floor, resp. the shop floor to the out-counter) are considered internal. A notion of equivalence that considers these internal steps is called (rooted) Branching Bisimulation, of van Glabbeek and Weijland for which we refer to \([GW91]\) and \([BW90]\). This equivalence abstracts from internal actions that do not enforce a choice.

Theorem 2.3 (Correctness of the Bakery Protocol)

\( B_n \) is (rooted) branching bisimilar with \( Q_{n+2} \).
3 A linear specification of the Bakery

The first step in our proof is to characterize the Bakery by four different situations. These are depicted in Figure 3. In this figure $d^i$ $IN_n(i + n 1)$ denotes the situation at the in-counter were a new customer $d$ has been provided a label $i$, but has not yet entered the shop floor. Similarly, $e$ $OUT_n(j)$ denotes the situation were customer $e$ has just been served by the baker, but he has not yet left the bakery. The starting state of the Bakery, when it is still empty, corresponds with $\alpha_n(0, \emptyset_{\text{bag}}, 0)$. Note that we have not considered yet the communication actions $c_1, c_2$ to be internal.

Definition 3.1 (Auxiliary processes $\alpha_n, \beta_n, \gamma_n$ and $\phi_n$)

$$
\begin{align*}
\alpha_n(i : \text{Nat}, b : \text{FBag}, j : \text{Nat}) & := \partial_H(IN_n(i)) || P_{\text{bag}}(b) || OUT_n(j)) \\
\beta_n(d : D, i : \text{Nat}, b : \text{FBag}, j : \text{Nat}) & := \partial_H(s_1(d^i)) \cdot IN_n(i + n 1) || P_{\text{bag}}(b) || OUT_n(j)) \\
\gamma_n(i : \text{Nat}, b : \text{FBag}, j : \text{Nat}, e : D) & := \partial_H(IN_n(i)) || P_{\text{bag}}(b) || \text{out}(e) \cdot OUT_n(j)) \\
\phi_n(d : D, i : \text{Nat}, b : \text{FBag}, j : \text{Nat}, e : D) & := \partial_H(s_1(d^i)) \cdot IN_n(i + n 1) || P_{\text{bag}}(b) || \text{out}(e) \cdot OUT_n(j))
\end{align*}
$$

In Figure 3 it is quite easy to see that the situation $\alpha$ can evolve into a situation $\beta$ (by the entrance of a new customer), or evolve into a situation $\gamma$ (when a customer leaves the shop floor and is being served by the baker). The details of these transitions are described by the next Lemma, that gives a linear specification (i.e., without parallel composition).

Lemma 3.2 (A linear specification for the Bakery)

$$
\begin{align*}
B'_n & = \alpha_n(0, \emptyset_{\text{bag}}, 0) \\
\alpha_n(i, b, j) & = \sum_{d \neq \perp} \text{enter}(d) \cdot \beta_n(d, i, b, j) \\
& + \sum_{f : \text{test}(f, b) \land \text{index}(f) = j} s_2(f) \cdot \gamma_n(i, \text{rem}(f, b), j + n 1, \text{data}(f)) \\
\beta_n(d, i, b, j) & = \text{size}(b) \neq n : \rightarrow s_1(d^i) \cdot \alpha_n(i + n 1, \text{add}(d^i, b), j) \\
& + \sum_{f : \text{test}(f, b) \land \text{index}(f) = j} s_2(f) \cdot \phi_n(d, i, \text{rem}(f, b), j + n 1, \text{data}(f)) \\
\gamma_n(i, b, j, e) & = \sum_{d \neq \perp} \text{enter}(d) \cdot \phi_n(d, i, b, j, e) \\
& + \text{out}(e) \cdot \alpha_n(i, b, j) \\
\phi_n(d, i, b, j, e) & = \text{size}(b) \neq n : \rightarrow c_1(d^i) \cdot \gamma_n(i + n 1, \text{add}(d^i, b), j, e) \\
& + \text{out}(e) \cdot \beta_n(d, i, b, j)
\end{align*}
$$

Proof. As the first case is trivial, and the other four cases are similar we only show the case for $\alpha_n(i, b, j)$ in detail.
\( \alpha_n(i, b, j) : \)

\[ \begin{array}{c}
\text{IN}_n(i) \quad \text{Pbag}_n(b) \quad \text{OUT}_n(j) \\
\end{array} \]

\( \beta_n(d, i, b, j) : \)

\[ \begin{array}{c}
d^i \\
\text{IN}_n(i + 1) \\
\end{array} \]

\[ \begin{array}{c}
\text{Pbag}_n(b) \\
\text{OUT}_n(j) \\
\end{array} \]

\( \gamma_n(i, b, j, e) : \)

\[ \begin{array}{c}
\text{IN}_n(i) \\
\text{Pbag}_n(b) \\
\text{OUT}_n(j) \\
e \\
\end{array} \]

\( \phi_n(d, i, b, j, e) : \)

\[ \begin{array}{c}
d^i \\
\text{IN}_n(i + 1) \\
\text{Pbag}_n(b) \\
e \\
\text{OUT}_n(j) \\
\end{array} \]

Figure 1: The four different situations of the Bakery
In the previous derivation we used the following identities:

\[ \sum_{f_1} x \cdot (x \mid y) = \sum_{f_1} x_1 \cdot (x \mid y) \cdot (x \mid y) \cdot (x \mid y) \]

The first identity is a direct consequence of the axiom CF1.pS. The second one can be derived from axiom CF1.pS together with COND2. Note that \( s_1(d^t) \cdot x \) abbreviates \( \sum_{f \in d^t} s_1(f) \cdot x \).

### 4 The introduction of the Invariant

#### 4.1 The definition of the Invariant

If the baker can serve the next customer, he just calls the value of the out-counter. The mechanism works because there will be exactly one customer with that index. This depends of course on the proper working of the in-counter, if this counter would provide multiple customers with the same index the whole procedure would fall down.

We introduce a so called invariant, that is a property of the system that will hold in all possible states, that says that for all indices \( j, j + n, j + 1, \ldots, i \), where \( i \) is the label of current customers (i.e., all labels of current customers) there will be exactly one customer with that index. As we do not want to deal with some quantification, we formulate this property recursively: if the Bakery is in state \((i, b, j)\) then we say that there is exactly one customer with index \( j \) (i.e., \( \text{cnt}(j, b) = 1 \)) and if that customer will leave the Bakery and the baker will call the next index then the invariant will hold again. This argument is repeated until the Bakery is empty, i.e., \( i = j \) and \( b = \emptyset_{\text{bag}} \).

**Definition 4.1.1 (The Invariant)**

\[ I_n(i, b, j) := I_n(i, b, j) \]

Intuitively we do know that the baker does not have to ask for some customer with index \( j \), but it can ask for the unique customer with index \( j \). We denote this unique customer by \( \text{frame}(j, b) \). The expression
frame(j, b) results the frame 1\(j\) if there is no frame with index \(j\) or more than one frames with index \(j\) in the bag, otherwise it results the properly defined unique frame in bag \(b\) with label \(j\).

More formally, in the context of the invariant \(I\) the condition test\((f, b)\land i\)ndex\((f) = j\) can be reduced to \(f = \text{frame}(j, b)\), whenever \(b\) is not empty. Note that if \(b\) is empty, then test\((f, b)\) reduces to \(f\).

**Lemma 4.1.2**

\[
I_n(i, b, j) \implies \sum_{f} \text{test}(f, b) \land \text{index}(f) = j \cdot a(f) \cdot x
\]

\[
= (I_n(i, b, j) \land b \neq \emptyset) \implies a(\text{frame}(j, b)) \cdot x[\text{frame}(j, b)/f]
\]

**Proof.** From Lemma B.5.2, see Appendix, we know that:

\[
\text{cnt}(j, b) = 1 \rightarrow (\text{test}(f, b) \land \text{index}(f) = j \iff \text{frame}(j, b) = f)
\]

As \(I_n(i, b, j)\) implies \(\text{cnt}(j, b) = 1\) we may replace the condition \(\text{test}(f, b) \land \text{index}(f) = j\) by \(\text{frame}(j, b)\), after which we apply axiom SUM2.

\(\square\)

### 4.2 Adding the Invariant to the linear specification of the Bakery

In order to simplify the sum expression \(\sum_{f} \text{test}(f, b) \land \text{index}(f) = j \cdot a(f) \cdot x\), in the definitions of \(\alpha_n\) and \(\beta_n\), into \(a(\text{frame}(j, b)) \cdot x[\text{frame}(j, b)/f]\) we add the invariant to the definitions of \(\alpha_n, \beta_n, \gamma_n\) and \(\phi_n\). Hence, we define the following auxiliary processes:

**Definition 4.2.1 (Auxiliary processes with the Invariant)**

\[
\begin{align*}
\alpha_n^I(i : \text{Nat}, b : \text{FBag}, j : \text{Nat}) & := I_n(i, b, j) \rightarrow \alpha_n(i, b, j) \\
\beta_n^I(d : \text{D}, i : \text{Nat}, b : \text{FBag}, j : \text{Nat}) & := I_n(i, b, j) \rightarrow \beta_n(d, i, b, j) \\
\gamma_n^I(i : \text{Nat}, b : \text{FBag}, j : \text{Nat}, c : \text{D}) & := I_n(i, b, j) \rightarrow \gamma_n(i, b, j, c) \\
\phi_n^I(d : \text{D}, i : \text{Nat}, b : \text{FBag}, j : \text{Nat}, e : \text{D}) & := I_n(i, b, j) \rightarrow \phi_n(d, i, b, j, e)
\end{align*}
\]

And we obtain:

**Lemma 4.2.2**
\[ B'_{n} = \alpha'_{n}(0, \emptyset_{\text{bag}}, 0) \]

\[ \alpha'_{n}(i, b, j) = I_n(i, b, j) \rightarrow \]
\[ \left( \sum_{d \neq \emptyset} \text{enter}(d) \cdot \beta'_{n}(d, i, b, j) \right) \]
\[ + b \neq \emptyset_{\text{bag}} \rightarrow \gamma'_{n}(i, \text{rem}(j, b), j + n, \text{frame}(j, b)) \]

\[ \beta'_{n}(d, i, b, j) = I_n(i, b, j) \rightarrow \]
\[ \left( \text{size}(b) < n \rightarrow c_1(d^i) \cdot \alpha'_{n}(i + n, \text{add}(d^i, b), j) \right) \]
\[ + b \neq \emptyset_{\text{bag}} \rightarrow \gamma'_{n}(i, \text{rem}(j, b), j + n, \text{frame}(j, b)) \]

\[ \gamma'_{n}(i, b, j, e) = I_n(i, b, j) \rightarrow \]
\[ \left( \sum_{d \neq \emptyset} \text{enter}(d) \cdot \phi'_{n}(d, i, b, j, e) \right) \]
\[ + \text{out}(e) \cdot \alpha'_{n}(i, b, j) \]

\[ \phi'_{n}(d, i, b, j, e) = I_n(i, b, j) \rightarrow \]
\[ \left( \text{size}(b) < n \rightarrow c_1(d^i) \cdot \gamma'_{n}(i + n, \text{add}(d^i, b), j, e) \right) \]
\[ + \text{out}(e) \cdot \beta'_{n}(d, i, b, j) \]

\[ \text{Proof. (Cases } \alpha'_{n} \text{ only.)} \]

\[ \alpha'_{n}(i, b, j) \]
\[ = ( \text{def } \alpha'_{n}, \text{ def } \alpha_{n}, \text{ COND1} ) \]
\[ I_n(i, b, j) \rightarrow \sum_{d \neq \emptyset} \text{enter}(d) \cdot \beta_n(d, i, b, j) \]
\[ + I_n(i, b, j) \rightarrow \sum_{f \cdot \text{test}(f, b) \land \text{index}(f) = j} c_2(f) \cdot \gamma_n(i, \text{rem}(f, b), j + n, \text{data}(f)) \]
\[ = ( \text{Lemma 4.1.2 } ) \]
\[ I_n(i, b, j) \rightarrow \sum_{d \neq \emptyset} \text{enter}(d) \cdot \beta_n(d, i, b, j) \]
\[ + \left( I_n(i, b, j) \land b \neq \emptyset_{\text{bag}} \rightarrow c_2(\text{frame}(j, b)) \cdot \gamma_n(i, \text{rem}(j, b), j + n, \text{data}(\text{frame}(j, b))) \right) \]
\[ = ( \text{SUM3, def } \text{rem}(j, b), \text{ def data}(j, b) ) \]
\[ I_n(i, b, j) \rightarrow \sum_{d \neq \emptyset} \text{enter}(d) \cdot \left( I_n(i, b, j) \rightarrow \beta_n(d, i, b, j) \right) \]
\[ + \left( I_n(i, b, j) \land b \neq \emptyset_{\text{bag}} \rightarrow c_2(\text{frame}(j, b)) \cdot \gamma_n(i, \text{rem}(j, b), j + n, \text{data}(j, b)) \right) \]
\[ = ( \text{COND1, def } \beta_n, \text{ def } \gamma_n ) \]
\[ I_n(i, b, j) \rightarrow \sum_{d \neq \emptyset} \text{enter}(d) \cdot \beta_n(d, i, b, j) \]
\[ + b \neq \emptyset_{\text{bag}} \rightarrow c_2(\text{frame}(j, b)) \cdot \gamma_n(i, \text{rem}(j, b), j + n, \text{frame}(j, b)) \]

\[ \square \]

### 4.3 Some properties of the Invariant

For sequel use we state some properties that can be derived from the Invariant, all facts are obvious facts that one indeed may expect in a Bakery.

- (a) Each customer has a unique index.
- (b) If it is not full, then the amount of customers equals the difference between the in- and out-counter (modulo \( n \)).
• (c) If it is full, then the in-counter equals the out-counter.

• (d) The amount of customers does not exceed the capacity.

• (e) If the two counters are equal, then the shop is either empty or full.

These facts are formalized as follows:

Lemma 4.3.1

\[(a) \; I_n(i, b, j) \land k < \text{size}(b) \rightarrow \text{cnt}(j + n, k, b) = 1\]
\[(b) \; I_n(i, b, j) \land \text{size}(b) < n \rightarrow \text{size}(b) = i - n, j\]
\[(c) \; I_n(i, b, j) \land \text{size}(b) = n \rightarrow i = j\]
\[(d) \; I_n(i, b, j) \rightarrow \text{size}(b) \leq n\]
\[(e) \; I_n(i, b, j) \land i = j \rightarrow \text{size}(b) = 0 \lor \text{size}(b) = n\]

Proof. See Appendix, Section A.

Furthermore we have the following facts, that consider the Invariant itself. The first one corresponds with the situation were a customer that has just been served by the baker (i.e., the customer with index \(j\)), leaves the bakery, after which the baker increases the out-counter to \(j + n + 1\). The second one corresponds with the situation in which a new customer arrives, which takes a label \(i\) and enters the bakery, after which the in-counter increases to \(i + n + 1\).

Lemma 4.3.2

\[(a) \; I_n(i, b, j) \rightarrow I_n(i, \text{rem}(j, b), j + n + 1)\]
\[(b) \; I_n(i, b, j) \land \text{size}(b) < n \rightarrow I_n(i + n + 1, \text{add}(d^i, b), j)\]

Proof. See Appendix, Section A.

4.4 Adding the invariant to the specification of the Queue

In the previous section we have shown that all customers in the Bakery have a unique label each, such that the customers can be served in order of their label. In other words, the state of the Bakery, that is the triple \((i, b, j)\), defines a queue, namely \(\text{frame}(j, b), \text{frame}(j + n + 1), \ldots, \text{frame}(i - n + 1)\). This queue is denoted by \(\text{queue}_n(i, b, j)\), its detailed definition is given in Section B.6.

In order to relate the Bakery, \(B_n\), with the Queue \(Q_{n+2}\) we define two auxiliary processes:

Definition 4.4.1 (Auxiliary Queue–processes with the Invariant)

\[\bar{Q}_m(q : \text{DQueue}, i : \text{Nat}, b : \text{FBag}, j : \text{Nat}) := I_m(i, b, j) :\rightarrow Q_m(q)\]
\[Q_m(i : \text{Nat}, b : \text{FBag}, j : \text{Nat}) := \bar{Q}_m(\text{queue}_m(i, b, j), i, b, j)\]
Before we continue we state two obvious properties, that regards the function \(\text{queue}_n(i, b, j)\) in the context of the invariant. The first one says that a new customer is placed at the back of the queue
\[
\text{addbck}(d, \text{add}(d_0, \text{add}(d_1, \ldots, \text{add}(d_n, \emptyset_{\text{queue}}) \ldots))) = \text{add}(d_0, \text{add}(d_1, \ldots, \text{add}(d_n, \text{add}(d, \emptyset_{\text{queue}})) \ldots))
\]
see for details the Appendix).

**Lemma 4.4.2**

\[
\begin{align*}
(a) \quad & (I_n(i, b, j) \land \text{size}(b) < n) \rightarrow \text{addbck}(d, \text{queue}_n(i, b, j)) = \text{queue}_n(i + n, 1, \text{add}(d^i, b), j) \\
(b) \quad & I_n(i, b, j) \rightarrow \text{size}(b) = \text{size}(\text{queue}_n(j, b)) \\
\end{align*}
\]

**Proof.** See Appendix, Section A. \(\square\)

Next, we prove that we can reformulate the definitions of \(Q_m^I\) and \(Q_m^\ell\), such that they are defined in their own terms, like the queue process itself as well.

Note, that the process \(Q_m^I(i, b, j)\) is defined completely in terms of the in- and out-counter \(i, j\) and the shop floor \(b\). The process \(Q_m^\ell(i, b, j)\) corresponds with a Bakery that does not have separate positions for the customers at the in- and out-counter, these positions are considered to be part of the shop floor as well. As a consequence there are no communications anymore between the in-counter and the shop floor, resp. the shop floor and the out-counter.

**Lemma 4.1**

\[
\begin{align*}
(a) \quad & \hat{Q}_m^I(q, i, b, j) = I_m(i, b, j) \rightarrow \\
& \quad \begin{cases} 
\text{size}(q) \neq m : \rightarrow & \sum_{d \neq \bot} \text{enter}(d) \cdot \hat{Q}_m^I(\text{addbck}(d, q), i + m, 1, \text{add}(d^i, b), j) \\
+ \text{size}(q) \neq 0 : \rightarrow & \text{out}(\text{top}(q)) \cdot \hat{Q}_m^I(\text{untop}(q), i, \text{rem}(j, b), j + m 1) 
\end{cases} \\
(b) \quad & Q_m^I(i, b, j) = I_m(i, b, j) \rightarrow \\
& \quad \begin{cases} 
\text{size}(b) \neq m : \rightarrow & \sum_{d \neq \bot} \text{enter}(d) \cdot Q_m^I(i + m, 1, \text{add}(d^i, b), j) \\
+ \text{size}(b) \neq 0 : \rightarrow & \text{out}(\text{data}(j, b) \cdot Q_m^I(i, \text{rem}(j, b), j + m 1) ) 
\end{cases}
\end{align*}
\]

**Proof.** Part \((a)\)

\[
\begin{align*}
\hat{Q}_m^I(q, i, b, j) \\
= \text{( def } \hat{Q}_m^I, \text{SUM3, Lemma 4.3.2(b), (a) )} \\
I_m(i, b, j) \rightarrow \\
(size(q) \neq m : \rightarrow \sum_{d \neq \bot} \text{enter}(d) \cdot (I_m(i + m, 1, \text{add}(d^i, b), j) : \rightarrow Q_m(\text{addbck}(d, q)) \\
+ \text{size}(q) \neq 0 : \rightarrow \text{out}(\text{top}(q)) \cdot (I_m(i, \text{rem}(j, b), j + m 1) : \rightarrow Q_m(\text{untop}(q)))) \\
= \text{( def } \hat{Q}_m^I) \\
I_m(i, b, j) \rightarrow \\
(size(q) \neq m : \rightarrow \sum_{d \neq \bot} \text{enter}(d) \cdot \hat{Q}_m^I(\text{addbck}(d, q), i + m, 1, \text{add}(d^i, b), j) \\
+ \text{size}(q) \neq 0 : \rightarrow \text{out}(\text{top}(q)) \cdot \hat{Q}_m^I(\text{untop}(q), i, \text{rem}(j, b), j + m 1) )
\end{align*}
\]
Part (b)

\[
\begin{align*}
Q_m(i, b, j) &= ( \text{def } Q_{m}, \text{ part } (a) ) \\
I_m(i, b, j) &:= \text{(Lemma 4.4.2(a), Proposition B.5.1(a), (b))}
\end{align*}
\]

Finally, we use that \( I_n(0, \emptyset_{\text{bag}}, 0) = t \), and we take the definitions of 2.2 and 4.4.1 together:

**Corollary 4.4.3** \( Q_m = Q_m(\emptyset_{\text{queue}}) = \bar{Q}_m(\emptyset_{\text{queue}}, 0, \emptyset_{\text{bag}}, 0) = Q_m(0, \emptyset_{\text{bag}}, 0) \)

5 Showing the Branching Bisimulation equivalence

From now on we consider the communication action actions \( c_1, c_2 \) that occur in the description of the Bakery, as internal actions. Hence, we take \( H = \{ c_1, c_2 \} \) and we consider \( \tau_H(\alpha_n(0, \emptyset_{\text{bag}}, 0)) \) in stead of \( \alpha_n(0, \emptyset_{\text{bag}}, 0) \).

Until now we have obtained that

\[
\begin{align*}
B_n &= \tau_H(\beta_n) = \tau_H(\alpha_n(0, \emptyset_{\text{bag}}, 0)) \\
Q_m &= Q_m(0, \emptyset_{\text{bag}}, 0)
\end{align*}
\]

So, in order to prove that \( B_n \) is (rooted) branching bisimilar with \( Q_{n+2} \) it is sufficient to prove that \( \tau_H(\alpha_n(0, \emptyset_{\text{bag}}, 0)) \) is (rooted) branching bisimilar with \( Q_{n+2}(0, \emptyset_{\text{bag}}, 0) \). That is, we have to relate the states of these processes such that the transfer property holds, i.e., each step of one state can be mimicked properly by a state to which it is related.

The state \( \tau_H(\alpha_n(i, b, j)) \) of the Bakery corresponds with the state \( Q_m(i', b', j') \) under the condition that for both states the invariant holds, \( m = n + 2 \), and the queues that are defined by the triples \( (i, b, j) \) and \( (i', b', j') \) are equal. Note that in the state \( \tau_H(\alpha_n(i, b, j)) \) the positions at the in- and out-counter are empty.

In case of \( \tau_H(\beta_n(d, i, b, j)) \) there is a "new customer" \( d \) at the in-counter. Hence, \( \tau_H(\beta_n(d, i, b, j)) \) is related with \( Q_m(i', b', j') \) under the condition that, among others, the queue of \( (i, b, j) \) with the "new customer" \( d \) added to the back corresponds with the queue of \( (i', b', j') \). In other words the condition \( R(\tau_H(\beta_n(d, i, b, j)), Q_m(i', b', j')) \) contains the condition that \( \text{addbck}(d, \text{queue}_n(i, b, j)) = \text{queue}_m(i', b', j') \).

The rooted branching bisimulation \( R \), that defines when states are related, is defined as follows:
Definition 5.1 (The bisimulation \( R \))

\[
R(\tau_H(\alpha_n^i(i, b, j)) \cdot Q_m^l(i', b', j')) = I_n(i, b, j) \land m = n + 2 \land I_m(i', b', j') \land queue_n(i, b, j) = queue_m(i', b', j')
\]

\[
R(\tau_H(\beta_n^d(d, i, b, j)) \cdot Q_m^l(i', b', j')) = I_n(i, b, j) \land m = n + 2 \land I_m(i', b', j') \land addbck(d, queue_n(i, b, j)) = queue_m(i', b', j')
\]

\[
R(\tau_H(\gamma_n^e(i, b, j, e)) \cdot Q_m^l(i', b', j')) = I_n(i, b, j) \land m = n + 2 \land I_m(i', b', j') \land add(e, queue_n(i, b, j)) = queue_m(i', b', j')
\]

\[
R(\tau_H(\phi_n^b(d, i, b, j, e)) \cdot Q_m^l(i', b', j')) = I_n(i, b, j) \land m = n + 2 \land I_m(i', b', j') \land addbck(d, add(e, queue_n(i, b, j))) = queue_m(i', b', j')
\]

The "symmetric" part, i.e., the equations for cases like \( R(Q_m^l(i', b', j'), \alpha_n^i(i, b, j)) \) have been omitted. Obviously,

\[
R(\alpha_n^i(0, 0, \text{bag}, 0) \cdot Q_{m+2}^l(0, 0, \text{bag}, 0)) = t,
\]

and \( R \) is rooted for \( \alpha_n^i(0, 0, \text{bag}, 0) \) and \( Q_{n+2}^l(0, 0, \text{bag}, 0) \), as the \( \tau \)-transition of \( \tau_H(\alpha_n^i(i, b, j)) \) is not enabled in case \( b \) is empty. Moreover, \( R \) is symmetric by definition, so, for proving that \( R \) is indeed a branching bisimulation we only have to prove that \( R \) satisfies the "transfer" property.

Lemma 5.2 \( R \) is a rooted branching bisimulation.

Proof. First we prove that the \( \text{enter}(d) \) summand of \( \tau_H(\alpha_n^i(i, b, j)) \) can be matched with the \( \text{enter}(d) \) summand of \( Q_m^l(i, b, j) \), i.e., we prove that

\[
R(\tau_H(\alpha_n^i(i, b, j)) \cdot Q_m^l(i', b', j')) \land I_n(i, b, j) \land d \neq \perp
\]

\[
\rightarrow I_m(i', b', j') \land size(b') < m \land R(\tau_H(\beta_n^d(d, i, b, j)) \cdot Q_m^l(i' + m 1, \text{add}(d', b'), j'))
\]

Note that the part \( I_n(i, b, j) \) in the premise, and the parts \( I_m(i', b, j) \) and \( size(b) < m \) in the conclusion are redundant, as the are consequences as well of \( R \).

\[
R(\tau_H(\alpha_n^i(i, b, j)) \cdot Q_m^l(i', b', j')) \land d \neq \perp
\]

\[
\leftrightarrow (\text{def } R, \text{Lemma 4.4.2(b), Lemma 4.3.1(d)})
\]

\[
I_n(i, b, j) \land m = n + 2 \land I_m(i', b', j') \land size(b') = size(\text{queue}_n(i, b, j)) = size(b) \leq n < n + 2 = m
\]

\[
\rightarrow (\text{Lemma 4.3.2(b), Lemma 4.4.2(a)})
\]

\[
I_n(i, b, j) \land m = n + 2 \land I_m(i' + m 1, \text{add}(d', b'), j') \land addbck(d, \text{queue}_n(i, b, j)) = addbck(d, \text{queue}_m(i', b', j')) = \text{queue}_m(i + m 1, \text{add}(d', b'), j')
\]

\[
\leftrightarrow (\text{def } R)
\]

\[
R(\tau_H(\beta_n^d(d, i, b, j)) \cdot Q_m^l(i' + m 1, \text{add}(d', b'), j'))
\]

\[
\square
\]
References


A more detailed treatment of the Invariant

In the sequel we will prove some lemmas that involve the invariant by induction on the size of the bag $b$. However, this does not fit directly into the induction scheme of [KvW95]. To remain within the context of [KvW95] we introduce a variant of the Invariant:

Definition A.1 (The Alternative Invariant) $I : \text{Nat} \times \text{Nat} \times \text{FBag} \times \text{Nat} \times \text{Nat} \rightarrow \text{Bool}$

$$I'_n(i, b, j, m) \ := \ i < n \wedge j < n \wedge (\text{cnt}(j, b) = 1 \wedge I'_n(i, \text{rem}(j, b), j + n, 1, m - 1)) \vee (i = j \wedge b = \emptyset_{\text{fbag}} \wedge m = 0))$$

Using the Special Bag Induction 2 from [GvW94], see Definition 7.7 of that paper, we can prove by induction on $b$ that

$$I_n(i, b, j) \iff I'_n(i, b, j, \text{size}(b))$$

When we say that we prove a property, say $\theta$, that involves $I_n(i, b, j)$, by induction on the size of $b$ we mean formally that we prove the corresponding property, say $\theta'$, that involves $I'_n(i, b, j, m)$ by induction on $m$, after which we apply the above equivalence regarding $I_n(i, b, j)$ and $I'_n(i, b, j, \text{size}(b))$ to obtain the proof of the original property $\theta$.

Lemma 4.3.1

(a) $I_n(i, b, j) \wedge k < \text{size}(b) \rightarrow \text{cnt}(j + n k, b) = 1$
(b) $I_n(i, b, j) \wedge \text{size}(b) < n \rightarrow \text{size}(b) = i - n j$
(c) $I_n(i, b, j) \wedge \text{size}(b) = n \rightarrow i = j$
(d) $I_n(i, b, j) \rightarrow \text{size}(b) \leq n$
(e) $I_n(i, b, j) \wedge i = j \rightarrow \text{size}(b) = 0 \vee \text{size}(b) = n$

Proof. Proof of (a), by induction to the size of $b$.

$$\text{size}(b) = 0 \text{ hence } b = \emptyset_{\text{fbag}} :$$

$$I_n(i, \emptyset_{\text{fbag}}, j) \wedge k < 0 \iff I_n(i, \emptyset_{\text{fbag}}, j) \wedge f \iff f \rightarrow \text{cnt}(j + n, b) = 1$$

$$\text{size}(b) > 1, \text{ by case distinction; either } k = 0 \text{ or } k > 0$$

$$I_n(i, b, j) \wedge k = 0 \rightarrow \text{cnt}(j + n k, b) = \text{cnt}(j, b) = 1$$

$$I_n(i, b, j) \wedge 0 < k < \text{size}(b)$$

$$\rightarrow (\text{Lemma } 4.3.2, \text{ Proposition } B.5.1)$$

$$0 < k < \text{size}(b) \wedge I_n(i, \text{rmv}(j, b), j + n, 1) \wedge k - 1 < \text{size}(\text{rmv}(j, b))$$

$$\rightarrow 1 \equiv \text{cnt}((j + n, 1) + (k - 1), \text{rmv}(j, b)) = \text{cnt}(j + n k, \text{rmv}(j, b)) = \text{cnt}(j + n k, b)$$

Proof of (b), by induction to the size of $b$ as well.
\( \text{size}(b) = 0 \) hence \( b = \emptyset_{\text{bag}} \):
\[
I_n(i, \emptyset_{\text{bag}}, j) \rightarrow i = j \iff i - j = 0 = \text{size}(\emptyset_{\text{bag}})
\]

\( \text{size}(b) > 0 \):
\[
\begin{align*}
I_n(i, b, j) \land \text{size}(b) < n \\
\rightarrow I_n(i, \text{rmv}(j, b), j + n) \land \text{size}(\text{rmv}(j, b)) < n \\
\rightarrow \text{size}(b) = \text{size}(\text{rmv}(j, b)) + 1 \overset{\text{ind}}{=} i - j + n = i - j 
\end{align*}
\]

For the proof of fact (c) we conclude from part (b) that
\[
\text{n} - n + 1 = \text{size}(\text{rmv}(j, b)) = i - j + n 
\]
hence \( i - j = 0 \), so \( i, j \) are equal modulo \( n \), and as both \( i \) and \( j \) are smaller than \( n \) we obtain \( i = j \).

Part (d) is a direct corollary from the previous parts.

\[\square\]

**Lemma 4.3.2**

(a) \( I_n(i, b, j) \rightarrow I_n(i, \text{rem}(j, b), j + n) \)

(b) \( (I_n(i, b, j) \land \text{size}(b) < n) \rightarrow I_n(i + n, \text{add}(d^i, b), j) \)

**Proof.** Part (a) follows directly from the definition of \( I_n \), part (b) is proven by induction on the size of \( b \).

\( \text{size}(b) = 0 \) hence \( b = \emptyset_{\text{bag}} \):
\[
\begin{align*}
I_n(i, \emptyset_{\text{bag}}, j) \land \text{size}(\emptyset_{\text{bag}}) < n \\
\rightarrow (\text{def } I_n, \text{cnt}(j, \emptyset_{\text{bag}}) \neq 1) \\
i < n \land j < n \land i = j \\
\rightarrow i < n \land j < n \land i = j
\end{align*}
\]

\( \land (i + n + 1 < n \land j < n \land i + n + 1 = j + n + 1 \\
\text{rem}(j, \text{add}(d^i, \emptyset_{\text{bag}})) = \text{rem}(i, \text{add}(d^i, \emptyset_{\text{bag}})) = \emptyset_{\text{bag}}) \\
i < n \land j < n \land i = j \land I_n(i + n + 1, \text{rem}(j, \text{add}(d^i, \emptyset_{\text{bag}})), j + n + 1) \\
i + n + 1 < n \land j < n \land \text{cnt}(j, \text{add}(d^i, \emptyset_{\text{bag}})) = \text{cnt}(i, \text{add}(d^i, \emptyset_{\text{bag}})) = 1 \\
\land I_n(i + n + 1, \text{rem}(j, \text{add}(d^i, \emptyset_{\text{bag}}), j + n + 1)
\rightarrow I_n(i + n + 1, \text{add}(d^i, \emptyset_{\text{bag}}), j)
\]

\( \text{size}(b)n + 1 > 0 \):
\[
\begin{align*}
\text{size}(b)n + 1 < \text{size}(b) < n \\
\rightarrow (\text{def } I_n, \text{Lemma 4.3.1}) \\
i < n \land j < n \land \text{cnt}(j, b) = 1 \land I_n(i, \text{rem}(j, b), j + n + 1) \\
\rightarrow (\text{Induction}) \\
i < n \land j < n \land \text{cnt}(j, b) = 1 \land I_n(i + n + 1, \text{add}(d^i, \text{rem}(j, b)), j + n + 1) \\
\rightarrow (\text{in case } i \neq j \text{ then } \text{cnt}(j, \text{add}(d^i, b)) = \text{cnt}(j, b) \text{ and } \text{add}(d^i, \text{rem}(j, b)) = \text{rem}(j, \text{add}(d^i, b)) ) \\
i + n + 1 < n \land j < n \land \text{cnt}(j, \text{add}(d^i, b)) = 1 \land I_n(i + n + 1, \text{rem}(j, \text{add}(d^i, b)), j + n + 1)
\rightarrow (\text{def } I_n) \\
I_n(i + n + 1, \text{add}(d^i, b), j + n + 1)
\]
Lemma 4.4.2

(a) \( I_n(i, b, j) \land \text{size}(b) < n \) \( \rightarrow \) \( \text{addbck}(d, \text{queue}_n(i, b, j)) = \text{queue}_n(i + n, \text{add}(d^i, b), j) \)

(b) \( I_n(i, b, j) \) \( \rightarrow \) \( \text{size}(b) = \text{size}(\text{queue}_n(j, b)) \)

Proof. The proof of (a) is similar to the proof of part (b) of Lemma 4.3.2.

\[ \text{size}(b) = 0 \text{ hence } b = \emptyset_{\text{fbag}} : \]
\[ I_n(i, \emptyset_{\text{fbag}}, j) \land \text{size}(\emptyset_{\text{fbag}}) = 0 < n \]
\[ \rightarrow ( \text{ def } I_n, \text{ cnt}(j, \emptyset_{\text{fbag}}) \neq 1 ) \]
\[ i = j \]
\[ \rightarrow i = j \land \]
\[ \text{addbck}(d, \text{queue}_n(i, \emptyset_{\text{fbag}}, j)) \]
\[ = \text{addbck}(d, \emptyset_{\text{queue}}) \]
\[ = \text{add}(d, \emptyset_{\text{queue}}) \]
\[ = \text{addbck}(\text{data}(j, \text{add}(d^i, \emptyset_{\text{fbag}})), \text{queue}_n(i + n, \emptyset_{\text{fbag}}, j + n 1)) \]
\[ = \text{queue}_n(i + n 1, \text{add}(d^i, \emptyset_{\text{fbag}}), j + n 1)) \]

\[ \text{size}(b) > 0 : \]
\[ I_n(i, b, j) \land 0 < \text{size}(b) < n \]
\[ \rightarrow i = j \land \text{size}(\emptyset_{\text{fbag}}) = 0 = \text{size}(\emptyset_{\text{queue}}) = \text{size}(\text{queue}_n(i, \emptyset_{\text{fbag}}, j)) \]

\[ \text{size}(b) = 0 \text{ hence } b = \emptyset_{\text{fbag}} : \]
\[ I_n(i, \emptyset_{\text{fbag}}, j) \]
\[ \rightarrow i = j \land \text{size}(\emptyset_{\text{fbag}}) = 0 = \text{size}(\emptyset_{\text{queue}}) = \text{size}(\text{queue}_n(i, \emptyset_{\text{fbag}}, j)) \]

\[ \text{size}(b) > 0 : \]
\[ I_n(i, b, j) \]
\[ \leftrightarrow ( \text{ in case } b \neq \emptyset_{\text{fbag}} \text{ then } \text{cnt}(j, b) = 1 \text{ and thus } \text{size}(b) = S(\text{size}(\text{rem}(j, b))) ) \]
\[ I_n(i, b, j) \land I_n(i, \text{rem}(j, b), j + n 1) \land \text{size}(\text{queue}_n(i, b, j)) \]
\[ = \text{size}(\text{add}(\text{data}(j, b), \text{queue}_n(i, \text{rem}(j, b), j + n 1))) \]
\[ = S(\text{size}(\text{queue}_n(i, \text{rem}(j, b), j + n 1))) \]
\[ = S(\text{size}(\text{rem}(j, b))) \]
\[ = \text{size}(\text{rem}(j, b)) \]
\[ = \text{size}(b) \]
B Datatypes

B.1 The datatype Bool

sort Bool
cons t, f:→ Bool
func ¬: Bool → Bool
    ∧, ∨, ⇒, eq : Bool × Bool → Bool
    if: Bool × Bool × Bool → Bool
var b, b': Bool

note Infix notation is used for ∧, ∨ and ⇒.
rew ¬t = f
    ¬f = t
    t ∧ b = b
    f ∧ b = f
    t ∨ b = t
    f ∨ b = b
    b ⇒ b' = (¬b) ∨ b'
    eq(t, t) = t
    eq(t, f) = f
    eq(f, t) = f
    eq(f, f) = t
    if(t, b, b') = b
    if(f, b, b') = b'
B.2 The datatype Nat, with modulo arithmatic

sort Nat
cons 0 :→ Nat
      S : Nat → Nat
func P : Nat → Nat
      +, − : Nat × Nat → Nat
      eq, ≤, < : Nat × Nat → Bool
      mod : Nat × Nat → Nat
      +, − : Nat × Nat × Nat → Nat
      if : Bool × Nat × Nat → Nat
var  n, m, k, i, j : Nat
note Infix notation is used for +, −, ≤, < and +n
rew P(0) = 0
     P(Sn) = n
     n + 0 = n
     n + Sn = S(n + m)
     n − 0 = n
     n − Sm = P(n − m)
     eq(0, 0) = t
     eq(0, Sn) = f
     eq(Sn, 0) = f
     eq(Sn, Sm) = eq(n, m)
     0 ≤ n = t
     Sn ≤ 0 = f
     Sn ≤ Sn = n ≤ m
     n < m = Sn ≤ m
     m mod 0 = 0
     m mod Sn = if(Sn ≤ m, (m − Sn) mod Sn, m)
     k +n m = (k + m) mod n
     k+n m = if(m mod n ≤ k mod n, k mod n + m mod n, n + (m mod n − k mod n))
     if(t, n, m) = n
     if(f, n, m) = m
B.3 The datatype $D$

sort $D$

cons $\bot: \rightarrow D$

$\bot : \text{Nat} \rightarrow D$

func $eq : D \times D \rightarrow \text{Bool}$

$if: \text{Bool} \times D \times D \rightarrow D$

var $d, d', e, e' : D$

note The injection $d : \text{Nat} \rightarrow D$ is arbitrary; it serves only to produce some elements of type $D$; a finite number of constants may serve as well

rew $eq(\bot, \bot) = t$

$eq(\bot, d(n)) = f$

$eq(d(n), \bot) = f$

$eq(d(n), d(n')) = eq(n, n')$

$if(t, d, d') = d$

$if(f, d, d') = d'$

B.4 The datatype Frame

sort Frame

cons $frame : D \times \text{Nat} \rightarrow \text{Frame}$

func $data : \text{Frame} \rightarrow D$

index : Frame $\rightarrow \text{Nat}$

if : Bool $\times \text{Frame} \times \text{Frame} \rightarrow \text{Frame}$

var $f, f' : \text{Frame}$

note $frame(d, i)$ is denoted by $d^i$

rew $data(d^i) = d$

index$(d^i) = i$

$if(t, f, f') = f$

$if(f, f, f') = f'$
B.5 The datatype FBag

sort FBag
cons \emptyset_{fbag} :\rightarrow FBag
  \text{add} : \text{Frame} \times FBag \rightarrow FBag
func \text{rmv} : \text{Frame} \times FBag \rightarrow FBag
  \text{test} : \text{Frame} \times FBag \rightarrow \text{Bool}
  \text{eq} : FBag \times FBag \rightarrow FBag
  \text{size} : FBag \rightarrow \text{Nat}
  \text{cnt} : \text{Nat} \times FBag \rightarrow \text{Nat}
  \text{frame} : \text{Nat} \times FBag \rightarrow \text{Frame}
  \text{data} : \text{Nat} \times FBag \rightarrow D
  \text{rem} : \text{Nat} \times FBag \rightarrow FBag
  \text{if} : \text{Bool} \times \times FBag \times FBag \rightarrow FBag
var b, b' : FBag

\textbf{note} The functions \text{cnt}, \text{frame}, \text{data} and \text{rem} are specific for the bakery protocol; their definitions use the fact that a frame is a pair, of which the second element is an index in Nat

\textbf{rew} \text{add}(f, \text{add}(f', b)) = \text{add}(f', \text{add}(f, b))
\text{rem}(f, \emptyset_{fbag}) = \emptyset_{fbag}
\text{rem}(f, \text{add}(f', b)) = \text{if}(f = f', b, \text{add}(f', \text{rem}(f, b)))
\text{test}(f, \emptyset_{fbag}) = f
\text{test}(f, \text{add}(f', b)) = \text{if}(f = f', t, \text{test}(f, b))
\text{eq}(\emptyset_{fbag}, \emptyset_{fbag}) = t
\text{eq}(\emptyset_{fbag}, \text{add}(f, b)) = f
\text{eq}(\text{add}(f, b), \emptyset_{fbag}) = f
\text{eq}(\text{add}(f, b), b') = \text{test}(f, b') \land \text{eq}(b, \text{rem}(f, b'))
\text{size}(\emptyset_{fbag}) = 0
\text{size}(\text{add}(f, b)) = S(\text{size}(b))
\text{cnt}(j, \emptyset_{fbag}) = 0
\text{cnt}(j, \text{add}(f, b)) = \text{if}(\text{index}(f) = j, S(\text{cnt}(j, b)), \text{cnt}(j, b))
\text{frame}(j, \emptyset_{fbag}) = \bot_2
\text{frame}(j, \text{add}(f, b)) = \text{if}(\text{index}(f) = j, \text{if}(\text{cnt}(j, b) = 0, f, \bot_2), \text{frame}(j, b))
\text{data}(j, b) = \text{data}(\text{frame}(j, b))
\text{rem}(j, b) = \text{rem}(\text{frame}(j, b), b)
\text{if}(t, b, b') = b
\text{if}(f, b, b') = b'

\textbf{Proposition B.5.1}

\text{cnt}(j, b) = 1 \rightarrow \text{size}(b) = \text{size}(\text{rem}(j, b)) + 1
\text{test}(j, b) \leftrightarrow \text{cnt}(\text{index}(f), b) > 0

\textbf{Proof.} Omitted.
Lemma B.5.2

\[ \text{cnt}(j, b) = 1 \rightarrow (\text{test}(f, b) \land \text{index}(f) = j \iff \text{frame}(j, b) = f) \]

**Proof.** By induction on \( b \). The base case is trivial, as \( \text{cnt}(j, \emptyset_{\text{bag}}) = 0 \neq 1 \).

For the inductive case, \( b = \text{add}(j', b') \), we prove the equivalent property

\[
\begin{align*}
\text{cnt}(j, \text{add}(j', b')) = 1 & \land \text{test}(f, \text{add}(j', b')) \land \text{index}(f) = j \\
\text{cnt}(j, \text{add}(f', b')) = 1 & \land
\begin{cases}
(f = f' \land \text{cnt}(j, b') = 0 \land \text{index}(f) = j) \\
\forall f \neq f' \land \text{index}(f') = j = \text{index}(f) \land \text{cnt}(j, b') = 0 = \text{test}(f, b') \\
\forall f \neq f' \land \text{index}(f') \neq j \land \text{cnt}(j, b') = 1 \land \text{test}(f, b') = f \end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cnt}(j, \text{add}(j', b')) = 1 & \land
\begin{cases}
(f = f' \land \text{frame}(j, \text{add}(f, b)) = f) \\
\forall f \neq f' \land \text{index}(f') = j = \text{index}(f) \land \text{f} \\
\forall f \neq f' \land \text{index}(f') \neq j \land \text{frame}(j, b) = f \end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{cnt}(j, \text{add}(f', b')) = 1 & \land
\begin{cases}
(f = f' \land \text{frame}(j, \text{add}(f', b)) = f) \\
\forall f \\
\forall f \neq f' \land \text{index}(f') \neq j \land \text{frame}(j, \text{add}(f', b)) = f \\
\text{cnt}(j, b) = 1 \land \text{frame}(j, b) = f
\end{cases}
\end{align*}
\]

\[\square\]
B.6 The datatype DQueue

sort DQueue
cons ⊥dqueue :→ DQueue
add : D × DQueue → DQueue
func addbck : D × DQueue → DQueue
top : DQueue → D
untop : DQueue → DQueue
queue : Nat × Nat × FBag × Nat → DQueue
if : Bool × DQueue × DQueue → DQueue
var q, q' : DQueue

note queuen(i, b, j) is specific for the bakery protocol; it takes the (unique) data for all indices in between i and j out of the bag b, and puts them in order of their indices in a queue

rew addbck(d, ⊥dqueue) = add(d, ⊥dqueue)
addbck(d', add(d, q)) = add(d, addbck(d', q))
top(⊥dqueue) =⊥
top(add(d, q)) = if(q = ⊥dqueue, d, top(q))
untop(⊥dqueue) = ⊥dqueue
untop(add(d, q)) = q
queuen(i, b, j) = if(i = j ∧ b ≠ ⊥fbag, ⊥dqueue, add(data(j, b), queuen(i, rem(j, b), j + n 1)))
if(t, q, q') = q
if(f, q, q') = q'

Lemma B.6.1

(a) b ≠ ⊥fbag → untop(queuen(i, b, j)) = queuen(i, rem(j, b), j + n 1)
(b) b ≠ ⊥fbag → top(queuen(i, b, j)) = data(j, b)

Proof.

b ≠ ⊥fbag → untop(queuen(i, b, j)) = untop(add(data(j, b), queuen(i, rem(j, b), j + n 1))
= queuen(i, rem(j, b), j + n 1)

b ≠ ⊥fbag → top(queuen(i, b, j)) = top(add(data(j, b), queuen(i, rem(j, b), j + n 1))
= data(j, b)

C The axiom system

In Table C the axioms for ACPpS are given. In that table var(φ) denotes the set of data variables that occur in the boolean expression φ, fv(x) denotes the set of free, unbound, data variables of x. Note that the prefix sum \( \sum_\phi a(v) \cdot x \) binds all occurrences of the variable v in x.
| A1        | $x + y$                           | $y + x$                          |
| A2        | $(x + y) + z$                     | $x + (y + z)$                    |
| A3<sub>pS</sub> | $\sum_\phi a(v) \cdot x + \sum_\psi a(v) \cdot x$ | $\sum_{\phi \psi} a(v) \cdot x$ |
| A6        | $x + \delta$                     | $x$                             |
| SUM1      | $\sum_\phi a(v) \cdot x$        | $\delta$                        |
| SUM2      | $\sum_{\psi \in \phi} a(v) \cdot x$ | $\sum_{\psi \in \phi} a(v) \cdot x(e/v)$ |
| SUM3      | $\sum_\phi a(v) \cdot x$        | $\sum_\phi a(v) \cdot \phi \vdash \vdash x$ |
| COND1     | $\phi :\vdash (x + y)$           | $\phi :\vdash x + \phi :\vdash y$ |
| COND2     | $\nu \notin \text{var}(\phi)$   | $\phi :\vdash \sum_\phi a(v) \cdot x$ | $\sum_\phi a(v) \cdot x$ |
| CF1<sub>pS</sub> | $\gamma(a, b)$ is defined       | $\sum_\phi a(v) \cdot x | \sum_\psi b(v) \cdot y = \sum_{\phi \psi} \gamma(a, b)(v) \cdot (x || y)$ |
| CF2<sub>pS</sub> | otherwise                         | $\sum_\phi a(v) \cdot x | \sum_\psi b(v) \cdot y = \delta$ |
| CM1       | $x || y$                         | $x || y + x || y + x || y$       |
| CM3<sub>pS</sub> | $\nu \notin f(v(x))$           | $(\sum_\phi a(v) \cdot x) || y = \sum_\phi a(v) \cdot (x || y)$ |
| CM4       | $(x_1 + x_2) || y$               | $(x_1 || y + x_2 || y$          |
| CM8       | $(x_1 + x_2) || y$               | $x_1 || y + x_2 || y$          |
| CM9       | $x || (y_1 + y_2)$               | $x || y_1 + x || y_2$          |
| D1<sub>pS</sub> | $a \notin H$                     | $\partial_H (\sum_\phi a(v) \cdot x)$ | $\sum_\phi a(v) \cdot \partial_H (x)$ |
| D2<sub>pS</sub> | $a \in H$                       | $\partial_H (\sum_\phi a(v) \cdot x)$ | $\delta$ |
| D3        | $\partial_H (x + y)$             | $\partial_H (x) + \partial_H (y)$ |
| TI1<sub>pS</sub> | $a \notin H$                     | $\tau_H (\sum_\phi a(v) \cdot x)$ | $\sum_\phi a(v) \cdot \tau_H (x)$ |
| TI2<sub>pS</sub> | $a \in H$                       | $\tau_H (\sum_\phi a(v) \cdot x)$ | $\sum_\phi \tau(v) \cdot \tau_H (x)$ |
| TI3       | $\tau_H (x + y)$                 | $\tau_H (x) + \tau_H (y)$       |

Table 1: The axiom system for ACP with prefix summation
Process Algebra with Autonomous Actions

Marc Voorhoeve and Twan Basten
Eindhoven University of Technology
email:{wspinmarc,tbasten}@win.tue.nl

Abstract
This paper introduces autonomous observable actions into process algebra. These actions can be observed but cannot be controlled by an environment. The proposed extension of ACP allows verifications without silent steps and fairness assumptions. The inclusion of inequalities makes it possible to verify that an implementation satisfies a given specification, with the specification indicating exactly where the implementation may reduce nondeterminism.

1 Introduction
Process algebra studies processes that are built from actions and operators. The occurrence of an action within a process is called an event. By executing an event, the process evolves into a new process. Two kinds of actions have been extensively studied: the standard observable ones and the unobservable (internal or silent) one, usually denoted as $\tau$. The occurrence of a standard observable action within a process is controlled both by the process itself (since it must be capable of performing the action) and its environment (who is responsible for allowing the action to occur). In contrast, the silent action $\tau$ may occur whenever a process is capable of performing it. In ACP, which is the process algebra studied in this paper, the operator by which processes can be controlled from outside is encapsulation. Ordinary observable actions can be disabled by an environment by means of encapsulation; silent actions cannot be disabled.

In this paper, we present a new kind of actions, called autonomous actions. These actions are observable, but not controllable by an environment, i.e. they share with $\tau$ the property that they can occur without any consent from the environment. Like $\tau$ in ACP, autonomous actions cannot be encapsulated.

The difference between autonomous actions and $\tau$ is that autonomous actions can be observed, whereas $\tau$ cannot. The fact that $\tau$ is unobservable means that an environment can only detect an event involving $\tau$ if the process after the event possesses less options for continuation than the process before the event. This is reflected by equations for processes, such as the branching equations $B1$ and $B2$ in ACP, that allow to eliminate silent actions. No such elimination is possible for autonomous actions, as every occurrence of such an action can be observed. The occurrence of communication errors in communication protocols form a typical example of autonomous actions, as they can be observed, but cannot be controlled.

A process that is capable of performing an autonomous action in the first step is called unstable. A process in which no autonomous action can occur is called stable, since an environment can stop its progress by encapsulating all its currently enabled actions. The discussion above shows that a choice involving at least one unstable process is nondeterministic. We call this kind of nondeterminism unstable nondeterminism. A more familiar kind of nondeterminism arises when several non-autonomous events with the same visible effect can occur, leading to different subsequent processes. Here, this is called stable nondeterminism.
Instead of only considering process equivalence, a partial order \( \leq \) on processes is introduced. If processes \( p \) and \( q \) satisfy \( p \leq q \), then either both processes are equal, or both \( p \) and \( q \) are unstable and the behaviour of \( p \) is a subset of the behaviour of \( q \). Since there must exist an autonomous action that both \( p \) and \( q \) can perform, and since this action cannot be controlled, an observer, when confronted with \( p \), cannot assure himself of the fact that he is not dealing with \( q \) instead. In a testing scenario (cf. [Abr87]), this means that global testing is disabled in, and only in, unstable processes. So \( \leq \) is based on reduction of unstable nondeterminism.

Partial order algebras that are based on reduction of nondeterminism are important for software engineering. Often, design decisions are made that reduce nondeterminism in a specification. The advantage of the approach in this paper is that in the specification one can distinguish between unstable nondeterminism that may be reduced and stable nondeterminism that may not. Technically speaking, the semantics of processes can be placed between completed simulation (making all actions autonomous) and bisimulation (considering non-autonomous actions only) (cf. [Gla90]).

The research reported here is related to Petri nets (cf. [BaVo95]), as the production of tokens has exactly the autonomous character described above. The main inspiration of the present paper is [BaBe94], where CSP-like choice operators [Hoa85] are introduced in ACP. The "partial bisimulation" preorder on processes from [BaBe94] that is also used in the present paper is related to the preorders in [Abr87] and [Wal90] that deal with the notion of divergence: a possibly infinite chain of autonomous actions (in their case silent actions \( \tau \)).

The reason for introducing CSP-like choice operators in ACP in [BaBe94] is that CSP distinguishes between deterministic and nondeterministic choice. Autonomous actions offer an alternative to nondeterministic choice operators, having the advantage of considerably simpler equations and models. Also, in this paper axioms and proof rules are given for reasoning algebraically with infinite processes, whereas the reasoning in [BaBe94] (and CSP for that matter) is model-based.

The authors wish to thank Jos Baeten, Rob van Glabbeek and Pedro D'Argenio for their support, explanations and corrections. The referees of the paper are thanked for their suggestions to improve the presentation and for pointing out references to related work.

2 The Basic Algebra

2.1 Signature and Axioms

Our basic algebra \( \text{BPA}_{\delta}^{\text{aa}} \) is an extension of \( \text{BPA}_{\delta} \): basic process algebra with inaction (cf. [BaWe90]). The signature and axioms are given in Table 1. First, in the header, the sort of atoms \( (A) \) is declared, which is a parameter of the theory. Each atom is an action. An overlining operator autonomizes an atom, thus creating an autonomous action. The sort \( F \) of autonomous actions is thus equal to \( \{\overline{a} : A \} \). The set \( A \cup F \) of all actions is denoted \( \text{AC} \). Note that \( AC \) is only an abbreviation and not a sort. The actions \( a \) and \( \overline{a} \) have the same visible effect and only differ in the way they can be controlled by an environment. The sort \( P \) of processes is a supersort of both \( A \) and \( F \). Sorts and their subsort relation are introduced in the first table entry.

The operator \( \cdot \) as well as the standard \( \text{BPA}_{\delta} \) operators \( + \), \( \cdot \), and \( \cdot \) are introduced in the second table entry, followed by the declaration of some variables and the axioms of the theory. As usual, the sequencing operator \( (\cdot) \) has priority over choice \((+)\) and may be omitted. The axiom system consists of equations and inequalities, giving rise to a partial order algebra (cf. [Hen88]), which means that in reasoning with inequalities, one is allowed to use reflexivity, transitivity, antisymmetry, closedness under contexts and

\[\text{We use the Z-style notation } \{a : A \} = \{ f(a) | a \in A \} \text{ throughout the paper.}\]
substitutivity. Note that, by antisymmetry, for two terms \( x \) and \( y \), \( x = y \) iff \( x \leq y \) and \( y \leq x \).

<table>
<thead>
<tr>
<th>[ \text{BPA}_{aa}^\leq (A) ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ F, P ; A \subseteq P ; F \subseteq P ]</td>
</tr>
<tr>
<td>[ \delta : P ; \ : A \rightarrow F \ : +, \ldots : P \times P \rightarrow P ]</td>
</tr>
<tr>
<td>[ a : A ; x, y, z : P ]</td>
</tr>
<tr>
<td>[ x + y = y + x ]</td>
</tr>
<tr>
<td>[ (x + y) + z = x + (y + z) ]</td>
</tr>
<tr>
<td>[ x + x = x ]</td>
</tr>
<tr>
<td>[ (x + y)z = x \cdot z + y \cdot z ]</td>
</tr>
<tr>
<td>[ x + \delta = x ]</td>
</tr>
<tr>
<td>[ \delta \cdot x = \delta ]</td>
</tr>
</tbody>
</table>

| \[ \bar{A} \] |
| \[ \bar{a} + a = \bar{a} \] |
| \[ \bar{a} \leq \bar{a} + x \] |
| \[ \bar{a} \cdot x \leq \bar{a} \cdot x + y \] |

Table 1: Basic process algebra with autonomous actions

Axiom \( \bar{A} \) states that a choice between an atom and its autonomization behaves like the autonomous action alone. In the presence of an encapsulation operator, the actions \( a \) and \( \bar{a} \) can be distinguished. However, \( \bar{a} + a \) and \( \bar{a} \cdot a \) cannot be distinguished: they have the same visible effect as well as the same behaviour with respect to encapsulation. The \( \bar{A} \) axioms (for "inequality") state that unstable nondeterminism may be reduced in an implementation. The algebra \( \text{BPA}_{aa}^\leq \) is obtained from \( \text{BPA}_{aa}^\prec \) by removing these inequalities from the axioms in Table 1.

The \( \text{BPA}_{aa}^\leq \) terms can be normalized. We inductively define the set \( B \) of basic terms as follows.

\[ \{ \delta \} \cup AC \subseteq B, \: \forall t : B, \: e : AC \cdot e \cdot t \in B, \: \forall s, t : B \cdot s + t \in B. \]

Property 1 For every closed term \( p : P \), there exists a \( t : B \) such that \( \text{BPA}_{aa} \vdash p = t \) (and thus \( \text{BPA}_{aa}^\leq \vdash p = t \)).

Proof

Standard term rewriting, using \( A3 \ldots A7 \) as rewrite rules from left to right (cf. [BaVe95]).

Example The theory \( \text{BPA}_{aa}^\leq \) may be used to demonstrate equivalence of the following processes that describe login procedures. The first process \( L1 \) prompts for a user identification and, having received one, prompts for a password. After having checked the combination, access is granted or refused. The user of the process perceives the prompting, granting and refusing as autonomous actions he cannot influence. The other actions are interactions between the process and the user. So \( L1 \) can be described as \( pui \cdot ui \cdot M1 \), where \( M1 = ppuw \cdot pw \cdot (ok + rej) \).

The second process \( L2 \) checks the user id before prompting for the password. If the user id is found correct, it proceeds as above; if not, it will still prompt for the password and reject regardless of the answer. The process \( L2 \) is thus described as \( pui \cdot ui (M1 + M2) \), with \( M1 \) as above and \( M2 = ppuw \cdot pw \cdot rej \).
In BPA, both login processes are equal, because we can demonstrate equality of $M_1 + M_2$ and $M_1$. Since $M_1$ is unstable, it follows from I2 that $M_1 \leq M_1 + M_2$. It follows from I1 that $\delta k + \rho e \geq \rho e$. Closedness under contexts yields $M_1 \geq M_2$, which implies by A3 that $M_1 = M_1 + M_1 \geq M_1 + M_2$. Thus $M_1 = M_1 + M_2$ by antisymmetry. As a result, $L_1 = L_2$.

The equality of $L_1$ and $L_2$ conforms to intuition and is desirable as well, since potential intruders are left uninformed whether they "guessed" a correct user id. At the end of Section 2.3, we will argue that equality of $L_1$ and $L_2$ cannot be proved without the inequalities I1 and I2.

### 2.2 Process Theory

Let $\mathcal{A}$ be a set of actions partitioned into disjoint subsets $\mathcal{A}$ (non-autonomous actions) and $\mathcal{F}$ (autonomous actions). A process space is a pair $(\mathcal{P}, \rightarrow)$, where $\mathcal{P}$ is a set of processes and $\rightarrow : \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ a ternary relation, the event relation. The event $p \xrightarrow{a} f$ represents successful termination of $p$ with action $a$. A stable process cannot perform any autonomous actions in the first step. The set $S$ of stable processes is thus defined as $S = \{ p : P \rightarrow \exists f : \mathcal{F}; p' : P \cup \{ f \} \cdot p \xrightarrow{f} p' \}$. We define a class of relations called partial bisimulations as simulations having the properties of a bisimulation for stable processes.

**Definition 1** A relation $R : \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ is called a partial bisimulation (PBS) iff it satisfies the following requirements. For $p, p', q, q' : \mathcal{P}; a : \mathcal{A}$.

1. $R$ is a simulation:
   
   $$pRq \Rightarrow (p = f \Leftrightarrow q = f) \land (pRq \land p \xrightarrow{a} p') \Rightarrow \exists q' : P \cdot (q \xrightarrow{a} q' \land p' R q'),$$

2. For stable processes, $R^{-1}$ is a local simulation:
   
   $$(pRq \land p \in S \land q \xrightarrow{a} q') \Rightarrow \exists p' : P \cdot (p \xrightarrow{a} p' \land p' R q').$$

Process $p : \mathcal{P}$ is said to implement $q : \mathcal{P}$ (notation $p \preceq q$) iff there exists a PBS $R$ such that $pRq$. The processes $p, q$ are said to be equivalent (notation $p \simeq q$) iff $p \preceq q$ and $q \preceq p$.

As usual, we can also define strong bisimulation on processes (see e.g. [BaWe00]). A strong bisimulation between processes $p$ and $q$ is a PBS relating both $p$ to $q$ and $q$ to $p$. If $\mathcal{F} = \mathcal{A}$, each PBS is a strong bisimulation.

**Property 2** The relation $\preceq$ is a preorder and, consequently, $\simeq$ is an equivalence relation.

**Proof**

We have to show that $\preceq$ is reflexive and transitive. The identity relation on processes is a PBS, which proves reflexivity. To prove transitivity, assume that $R, S$ are PBSs such that $pRq$ and $qSr$. Obviously, the relation composition $R \circ S$ is a simulation. If $p$ is stable, then $R^{-1}$ is a local simulation, so $q$ is stable and hence $S^{-1}$ is a local simulation. So $(R \circ S)^{-1}$ is a local simulation and $R \circ S$ a PBS. So $p \preceq r$.

With preorder relations like partial bisimulation, special care must be taken with respect to termination and nontermination of processes. It is undesirable that a terminating process can be implemented by a nonterminating one. Therefore, we shall define a termination property for processes. Intuitively, a process is called terminating if it has a path leading to $\checkmark$ and it can only make transitions to terminating processes. A terminating process cannot deadlock, nor can it contain a cycle from which escape is impossible.
Definition 2 The set $\mathcal{AC}^*$ of process traces consists of finite sequences $a_1 \ldots a_n$ of actions, including the empty trace $\epsilon$. The reachability relation $\Downarrow$ is defined as the smallest relation (ordered by set inclusion) such that for all $p, p', p'' : \mathcal{P} \cup \{\top\}$; $a : \mathcal{AC}$; $\alpha : \mathcal{AC}^*$,

$$p \Downarrow p', \ (p \xrightarrow{a} p' \land p' \xrightarrow{\alpha} p'') \Rightarrow p \xrightarrow{a} p'' .$$

A process $p$ is called terminating (notation $p \Downarrow$) iff it can only reach processes that can reach $\top$. Formally,

$$p \Downarrow \Leftrightarrow \forall p' : \mathcal{P}; \alpha : \mathcal{AC}^* \bullet (p \xrightarrow{a} p' \Rightarrow \exists \beta : \mathcal{AC}^* \bullet p' \xrightarrow{\beta} \top).$$

2.3 A Process Model for BPA_{Aaa}$^\leq$

In order to construct a process model for BPA_{Aaa}$^\leq$, we have to give interpretations $A, F$, and $P$ respectively for the sorts $A, F$ and $P$. Let the process space $\mathcal{P}$ be the set of closed BPA_{Aaa}$^\leq(A)$ terms. Let $A$ and $F$ simply be equal to $A$ and $F$ respectively. Obviously, these interpretations do satisfy the requirements $A \subseteq \mathcal{P}$ and $F \subseteq \mathcal{P}$. Let $\mathcal{AC} = A \cup F$. The relation $\Downarrow$ is given as the smallest relation satisfying the requirements in Table 2. In this table, the variables $a : A$; $p, p', q : \mathcal{P}$; $e : \mathcal{AC}$ are used.

<table>
<thead>
<tr>
<th>$e$</th>
<th>$\Downarrow$</th>
<th>$\Downarrow$</th>
<th>$\Downarrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>$\Downarrow$</td>
<td>$\Downarrow$</td>
<td>$\Downarrow$</td>
</tr>
<tr>
<td>$p \xrightarrow{a} p'$</td>
<td>$p \xrightarrow{a} p'$</td>
<td>$p \xrightarrow{a} p'$</td>
<td></td>
</tr>
<tr>
<td>$p + q \xrightarrow{a} p'$</td>
<td>$q + p \xrightarrow{a} p'$</td>
<td>$p \cdot q \xrightarrow{a} p' \cdot q$</td>
<td></td>
</tr>
<tr>
<td>$p \xrightarrow{a} q$</td>
<td>$p \xrightarrow{a} q$</td>
<td>$p \xrightarrow{a} q$</td>
<td></td>
</tr>
<tr>
<td>$q \xrightarrow{a} p'$</td>
<td>$q \xrightarrow{a} p'$</td>
<td>$p \cdot q \xrightarrow{a} q \cdot p$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Plotkin-style SOS rules for BPA_{Aaa}$^\leq$

Our model will be the set of equivalence classes of $\mathcal{P}$ modulo PBS equivalence, denoted $\mathcal{P} / \simeq$. The following property has as a consequence that $\mathcal{P} / \simeq$ with $\preceq$ is a partial order algebra.

Property 3 The relation $\preceq$ is a precongruence w.r.t. the BPA_{Aaa}$^\leq$ operators, i.e., for all $a, b : A$ with $a \preceq b$, $\overline{a} \preceq \overline{b}$ and for all $p_1, p_2, q_1, q_2 : \mathcal{P}$ with $p_1 \preceq p_2$ and $q_1 \preceq q_2$, $p_1 \cdot q_1 \preceq p_2 \cdot q_2$ and $p_1 + q_1 \preceq p_2 + q_2$.

Proof

If $a \preceq b$ then $a = b$ and thus $\overline{a} = \overline{b}$. The other two cases are proved by constructing a PBS.

For closed terms $p$ and $q$, BPA_{Aaa}$^\leq$ $\vdash p \preceq q$ signifies that $p \preceq q$ can be derived from the axioms, whereas $\mathcal{P} / \simeq \vdash p \preceq q$ signifies that $p \preceq q$ is valid in the model $\mathcal{P} / \simeq$. That is, $\mathcal{P} / \simeq \vdash p \preceq q$ iff $p \preceq q$. The following theorem states soundness and completeness of BPA_{Aaa}$^\leq$ with respect to $\mathcal{P} / \simeq$.

Theorem 1 Let $p, q$ be closed BPA_{Aaa}$^\leq$ terms. Then BPA_{Aaa}$^\leq$ $\vdash p \preceq q$ iff $\mathcal{P} / \simeq \vdash p \preceq q$.

Proof

It follows from Property 3 that it suffices to check the validity of the axioms. In each case, PBSs can be constructed easily. For the axioms $A1 \ldots A7$ and $\overline{A}$, even strong bisimulations can be constructed.
Completeness is more involved. Suppose $p \preceq q$. Then there exists a PBS $R$ such that $pRq$. We use induction on the number of symbols in $p$ plus the number of symbols in $q$. The basic case, where $p$ and $q$ each consist of a single symbol follows from the fact that $R$ is a PBS such that $pRq$. Processes $p$ and $q$ are either both equal to $\delta$ or both consist of the same action. Hence, in the basic case, $\text{BPA}_{\text{aa}} \vdash p \preceq q$.

In the induction step, from Property 1, we may assume that $p$ and $q$ are basic terms; so

$$p = \sum_{i,j} a_i + \sum_{j,k} b_j \cdot s_j, \quad q = \sum_{k,L} c_k + \sum_{k,L} d_k \cdot t_k,$$

where $I, J, K$ and $L$ are finite index sets, the $a_i, b_j, c_k$ and $d_k$ are actions from $\text{AC}$ and the $s_j$ and $t_k$ are basic terms. We distinguish two cases.

First, assume that $p$ is stable. Then $R^{-1}$ is a local simulation. We may conclude from the derivation rules in Table 2 that

$$\forall i : I \bullet \exists k : K \bullet a_i = c_k, \quad \forall j : J \bullet \exists l : L \bullet b_j = d_l \land s_j R t_l, \quad \forall k : K \bullet \exists i : I \bullet c_k = a_i, \quad \forall l : L \bullet \exists j : J \bullet d_l = b_j \land s_j R t_l.$$

The induction hypothesis yields

$$\forall j : J \bullet \exists l : L \bullet b_j = d_l \land \text{BPA}_{\text{aa}} \vdash s_j \leq t_l.$$

It follows immediately that $\text{BPA}_{\text{aa}} \vdash p \preceq q$.

Second, assume that $p$ is unstable. This implies that one of its initial actions is autonomous. Since $R$ is a simulation, we may conclude from the derivation rules in Table 2 that

$$\forall i : I \bullet \exists k : K \bullet a_i = c_k \lor \exists_i = c_k, \quad \forall j : J \bullet \exists l : L \bullet (b_j = d_l \lor \exists_j = d_l) \land s_j R t_l.$$

The induction hypothesis again yields

$$\forall j : J \bullet \exists l : L \bullet (b_j = d_l \lor \exists_j = d_l) \land \text{BPA}_{\text{aa}} \vdash s_j \leq t_l.$$

Using axioms $\overline{1}, \overline{11}$ and $\overline{2}$, $\text{BPA}_{\text{aa}} \vdash p \preceq q$ follows easily by observing that, for any $a, b : A; x, y, z : P$ and $\xi$ of the form $\overline{a} \lor \overline{a} \cdot x$,

$$\xi + b \leq \xi + \overline{b} + b = \xi + \overline{b} \leq \xi + \overline{b} + y, \quad \xi + b \cdot y \leq \xi + \overline{b} \cdot y + b \cdot y = \xi + \overline{b} \cdot y \leq \xi + \overline{b} \cdot y + z.$$

The set of equivalence classes of $\mathcal{P}$ modulo strong bisimulation equivalence is a sound and complete model for $\text{BPA}_{\text{aa}}$. This can be shown using a different interpretation of atoms and autonomous actions. Let $F$ be empty and let $A$ be equal to $\text{AC}$. This means that all actions in $\text{AC}$ are considered to be non-autonomous and thus all processes are considered to be stable. Using this different interpretation, all proofs above can be easily adapted. This result has the consequence that, as long as no inequality is used, two closed $\text{BPA}_{\text{aa}}$ terms that are derivably equal are strongly bisimilar. The use of an inequality in a verification represents the reduction of unstable nondeterminism. It is important to keep track at which points this reduction occurs. This can be achieved by delaying the use of $\overline{11}$ and $\overline{12}$ until they are really needed, i.e., proceeding within $\text{BPA}_{\text{aa}}$ as long as possible.

At this point, we can also justify our claim that the equivalence of the login procedures $L_1$ and $L_2$ in the example of Section 2.1 cannot be proved without the inequalities $\overline{11}$ and $\overline{12}$. This follows immediately from the observations above and the fact that $L_1$ and $L_2$ are not bisimilar.

### 3 Iteration

In this section, an iteration operator, c.q. the binary Kleene star (BKS) is added to the basic theory. The additional signature and axioms, which have been taken from [BeBP94], are stated in Table 3.
Table 3: Axioms for the binary Kleene star

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z(x^*y) + y = x^*y$</td>
<td>BKS1</td>
</tr>
<tr>
<td>$x^<em>(y</em>z) = (x^*y)z$</td>
<td>BKS2</td>
</tr>
<tr>
<td>$x^*(y((x + y)^*z) + z) = (x + y)^*z$</td>
<td>BKS3</td>
</tr>
</tbody>
</table>

Table 4: SOS rules for the binary Kleene star

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p \xrightarrow{e} p'$</td>
<td>$p^<em>q \xrightarrow{e} p'^</em>(p^*q)$, $q^*p \xrightarrow{e} p'$</td>
</tr>
<tr>
<td>$p \xrightarrow{\varnothing}$</td>
<td>$p^*q \xrightarrow{e} p^*q$, $q^*p \xrightarrow{e} \varnothing$</td>
</tr>
</tbody>
</table>

Theorem 2 Let $p, q$ be closed $\text{BPA}_a^\leq$ terms. Then $\text{BPA}_a^\leq \vdash p \leq q$ implies $\mathcal{P}/\sim \models p \leq q$.

Proof
Standard. First, show that $\leq$ is a precongruence w.r.t. the binary Kleene star. Second, check the validity of $BKSI \ldots BKSI3$ by constructing PBSs. Note that even strong bisimulations can be constructed for these axioms.

The following theorem states that the theory $\text{BPA}_a^\leq$ does not introduce any inequalities (and therefore equations) between closed $\text{BPA}_a^\leq$ terms which were not already derivable from $\text{BPA}_a^\leq$. That is, the theory $\text{BPA}_a^\leq$ is a conservative extension of $\text{BPA}_a^\leq$.

Theorem 3 Let $p$ and $q$ be closed $\text{BPA}_a^\leq$ terms. Then $\text{BPA}_a^\leq \vdash p \leq q$ iff $\text{BPA}_a^\leq \vdash p \leq q$.

Proof (sketch)
We use the theory and terminology developed in [D'Ar95]. It follows from the format of the derivation rules that the term deduction system defined by Table 2 is source dependent. Furthermore, it has no rules with negative premises, which implies that it has a unique well supported model. For the same reason, the term deduction system defined by Tables 2 and 4 has a unique well supported model. As a result, it is an operational conservative extension (up to $\leq$) of the term deduction system of Table 2. It then follows from Theorems 1 and 2 that $\text{BPA}_a^\leq$ is a conservative extension of $\text{BPA}_a^\leq$.

A complete finite axiomatization of $\text{BPA}_a^\leq$ and thus $\text{BPA}_a^\leq$ is impossible ([Sew94]; cf. [FoZa94] for related results).

For reasoning with iterative processes, we formulate some conditional equations and inequalities. $\text{RSP}^+$ is the iteration variant of the "Recursive Specification Principle" (cf. e.g. [BaWe90]); the other names are derived from "Iteration Inequality." For any $v, w, x, y, z : \mathcal{P}$,
Clearly, \((IL \land IR) \Rightarrow II \Rightarrow RSP^*\). It is not hard to show that \(RSP^*\) and \(BKS1\) imply \(BKS2\) and \(BKS3\).

**Theorem 4** The above derivation rules are valid in the model \(P/\sim\).

**Proof (sketch)**

Inequality \(IR\) is proved by constructing a PBS as follows. Let \(p\), \(q\) and \(r\) be closed \(BPA^a\) terms and let \(R\) be a PBS such that \(pR(q\ p\ r)\). \(R^k\) is the \(k\)-fold composition of \(R\), where \(R^0\) equals the identity relation and \(R^1\) equals \(R\). The relation \(Q\) is defined as the smallest relation such that \(\checkmark Q\checkmark\) and for any \(k \geq 0\), closed terms \(\xi\) and open term \(E(X)\) with \(X\) as its only variable, \(\xi Q\xi\) and \(\xi R^k E(p) \Rightarrow \xi Q E(q^*r)\). Using the operational semantics, it can be shown that this relation is a PBS relating \(p\) and \(q^*r\). So \(IR\) is valid. The validity of \(IL\) is shown using a symmetrical argument.

We have thus shown the validity of \(IL, IR, II\) and \(RSP^*\). However, the inequalities \(IL\) and \(IR\) permit the derivation of, e.g., \(\overline{a} \overline{b} \leq \overline{a} \overline{b}\). This is undesirable, since the second process is terminating, whereas the first process is not. For that reason, in the remainder, we only use the weaker inequality \(II\). In \(BPA^a\), \(II\), a terminating process cannot be implemented by a nonterminating process.

**Property 4** Let \(p, q : P\). If \(BPA^a\) \(/\mathcal{I} \vdash p \leq q\) and \(q \notin\), then \(p \notin\).

**Proof (sketch)**

Structural induction on \(p\) is used. The basic case is trivial. In the induction step, we consider three cases. First, if \(p = p_1 + p_2 \leq q\), we may assume that \(\delta \neq p_1 \neq p_2 \neq \delta\). The only way that \(p \leq q\) can be deduced from \(BPA^a\) \(/\mathcal{I} \vdash p_1 \leq q_1 \land p_2 \leq q_2\) is when \(q\) has the form \(q_1 + q_2\) and \(BPA^a\) \(/\mathcal{I} \vdash q_1 \leq q_1 \land p_2 \leq q_2\). If \(q \notin\), then \(q_1 \notin\) and \(q_2 \notin\), allowing completion of the induction step in this case. Second, if \(p = p_1 \cdot p_2\), the argument is similar. Third, if \(p = p_1 \cdot p_2\), \(p \leq q\) can be derived in two ways.

The first possibility is that \(q\) has the form \(q_1 \cdot q_2\) and \(BPA^a\) \(/\mathcal{I} \vdash p_1 \leq q_1 \land p_2 \leq q_2\), which is dealt with as above. The second possibility follows from \(II\) if \(p_1 \cdot q + p_2 \leq q \leq r \cdot q + s\) and \(p_2 \leq s\), for some processes \(r\) and \(s\). Now suppose \(q \notin\). We have to show that \(p \notin\), i.e. that \(p_1 \notin\) and \(p_2 \notin\). Assume \(p_2 = \delta\). Then \(s = \delta\) and \(q \leq r \cdot q\). Soundness gives \(q \leq r \cdot q\). Since \(q \notin\), it follows that \(q \xrightarrow{\alpha} \checkmark\) for some trace \(\alpha\). So there exist nonempty traces \(\beta, \gamma\) such that \(\alpha = \beta \gamma\), \(r \xrightarrow{\beta} \checkmark\) and \(q \xrightarrow{\gamma} \checkmark\). So for any path of \(q\) leading to \(\checkmark\), a shorter path can be found, which is a contradiction. So \(p_2 \neq \delta\). Therefore, we may again deduce that \(q\) has the form \(q_1 + q_2\) and \(BPA^a\) \(/\mathcal{I} \vdash p_1 \leq q_1 \land p_2 \leq q_2\), which can again be treated as above.

\[\square\]

### 4 Concurrency and Communication

Table 5 gives the additional operators and axioms for the extension of ACP with autonomous actions. An extra parameter, the communication function \(\gamma : A \times A \rightarrow A \cup \{\delta\}\), is added, that must satisfy the rules of commutativity and associativity whenever applicable. The parameter function \(\gamma\), is lifted to a full communication function \(\bar{\gamma} : (ACU\{\delta\}) \times (ACU\{\delta\}) \rightarrow\)
ACU, by specifying for any \( a, b : A \) that 
\[
\gamma(a, b) = \gamma(a, b) = \delta 
\]
if \( \gamma(a, b) = \delta \),
\[
\gamma(a, b) = \gamma(a, b) = \gamma(a, b) = \delta 
\]
if \( \gamma(a, b) \neq \delta \)
\[
\gamma(a, b) = \gamma(a, b) = \gamma(\delta, a) = \gamma(\delta, a) = \delta 
\]
As the above equations show, the visible effect of a communication depends on the visible effect of the participating actions. If any participant in a communication is autonomous, the resulting action (if defined) is autonomous.

Only a few of the additional operators need explanation. Encapsulation of autonomous actions is impossible \((D)\), which is the essence of being autonomous, as explained in the introduction. The renaming operator \( t_I \) renames the atoms from a given set \( I \) into a special atom \( t \); autonomized atoms from \( I \) are renamed into \( t \). An operator \( \alpha_I \) is added, that autonomizes the atoms from \( I \). The composed operator \( t_I \circ \alpha_I \) can be situated between pre-abstraction [BaBe88] and full abstraction (cf. e.g. [BaWe90]).

<table>
<thead>
<tr>
<th>ACPaa~(A, ( \gamma ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPAaa~(A)</td>
</tr>
<tr>
<td>( t : A ) ( \partial(_), \alpha(_), t_1(_): IP(A) \times P \rightarrow P ) ( _ ), ( _ ), ( _ ) : ( P \times P \rightarrow P )</td>
</tr>
<tr>
<td>( a : A ; e, f, g : ACU{b }; x, y, z : P ; H, I : IP(A) ; L : { \partial_H, \alpha_I, t_I } )</td>
</tr>
</tbody>
</table>
| \[
\begin{array}{c|c}
\hline
a \notin H \Rightarrow \partial_H(a) = a & D1 \quad e \mid f = \gamma(e, f) & CF \\
\hline
a \in H \Rightarrow \partial_H(a) = \delta & D2 \\
\hline
\delta_H(\bar{a}) = \bar{a} & D \\
\hline
a \notin I \Rightarrow \alpha_I(a) = a & A1 \quad x \parallel y = x \parallel y + y \parallel x + y \parallel x + y \parallel y & CM1 \\
a \in I \Rightarrow \alpha_I(a) = \bar{a} & A2 \quad e \parallel x = e \cdot x & CM2 \\
\alpha_I(\bar{a}) = \bar{a} & A3 \quad (x + y) \parallel z = z \parallel x \parallel z + y \parallel z & CM3 \\
\hline
a \notin I \Rightarrow t_I(a) = a & A4 \quad e \parallel f \cdot x = (e \parallel f) x & CM4 \\
a \in I \Rightarrow t_I(a) = t & A5 \quad e \parallel f \cdot y = (e \parallel f)(a) \parallel y & CM5 \\
t_I(\bar{a}) = t_I(a) & A6 \quad (x + y) \parallel z = x \parallel z + y \parallel z & CM6 \\
\hline
L(\delta) = \delta & L1 \\
L(x + y) = L(x) + L(y) & L2 \\
L(x \cdot y) = L(x) \cdot L(y) & L3 \\
\hline
\end{array}
\] |

Table 5: ACP with autonomous actions

The following theorem shows that the non-BPAaa~ operators can be eliminated.

**Theorem 5** For every closed ACPaa~ term \( p \), there is a basic BPAaa~ \( s \) such that ACPaa~ \( p \rightarrow s \).

**Proof (sketch)**
Consider ACPaa~ as a term rewriting system. That is, consider the axioms \( A3 \ldots A7 \) plus all the axioms in Table 5 as rewrite rules from left to right. We can use the method of recursive path ordering to show that this term rewriting system is strongly normalizing (see [BaVe95] for a detailed description). Here, we only give the partial ordering on the ranked operators of ACPaa~. For \( k > 1 \) and \( a : A \),

\[
\delta < a \quad \{ a, +, \} < \cdot < \{ \parallel, z, \partial_H, \alpha_I, t_I \} \quad \{ \parallel, x \parallel i, k \} \quad \{ \parallel, x \parallel i, k + 1 \}
\]
Now, it can be verified by structural induction that the normal forms are basic BPA\(\leq\) terms. The basic step is simple. For the induction step, note that for any normal form containing a non-BPA\(\leq\) operator applied to normal forms, which by the induction hypothesis are basic BPA\(\leq\) terms, a rewrite rule can be found in Table 5, which is a contradiction.

A process model is defined as before. The process space \(\mathcal{P}\) is defined as the set of closed ACP\(\leq\) terms. The relation \(\Rightarrow\) is the smallest relation satisfying the rules in Tables 2 and 6. Let \(a : \mathcal{A}; e, f, g : \mathcal{AC}; H, I : \mathbb{P}(\mathcal{A})\) and \(p, p', q, q' : \mathcal{P}\).

\[
\begin{array}{l}
p \xrightarrow{e} p' \\
p \xrightarrow{f}, p' = \top_{\mathcal{A}}(e, f) \\
p \xrightarrow{f}, q \xrightarrow{e} q' \\
p \xrightarrow{f}, q \xrightarrow{f}, q' = \top_{\mathcal{A}}(e, f) \\
p \xrightarrow{f}, q \xrightarrow{f}, q' = \top_{\mathcal{A}}(e, f) \\
\end{array}
\]

Table 6: SOS rules for non-BPA\(\leq\) operators

The following property and theorem deal with soundness of ACP\(\leq\).

Property 5 The \(\leq\) preorder is a precongruence w.r.t. the ACP\(\leq\) operators.

Proof Let \(R_1\) and \(R_2\) be PBSs. Let \(Q = \{p_1, p_2, q_1, q_2 : \mathcal{P} \mid p_1 \leq p_2 \land q_1 \leq q_2 \Rightarrow p_1 \parallel q_2 \leq q_1 \parallel q_2\}\). It is easy to check that \(Q\) is a PBS. So for any \(p_1, p_2, q_1, q_2 : \mathcal{P}, (p_1 \leq q_1 \land p_2 \leq q_2) \Rightarrow p_1 \parallel q_2 \leq q_1 \parallel q_2\). So \(\leq\) is a precongruence for the merge operator. Let \(S = \{p_1, p_2, q_1, q_2 : \mathcal{P} \mid p_1 \leq p_2 \land q_1 \leq q_2 \Rightarrow p_1 \parallel q_2 \leq q_1 \parallel q_2\}\). Then \(Q \cup S\) is a PBS, showing that \(\leq\) is a precongruence for the left merge operator. The communication merge operator is treated likewise.

Let \(R\) be a PBS. Define \(Q = \{p, q : \mathcal{P} \mid p \leq q \Rightarrow \partial_H(p) \leq \partial_H(q)\}\). It is easy to show that \(Q\) is a PBS, whence \(\leq\) is a precongruence for encapsulation. It is essential that the encapsulation of an autonomous action is not \(\delta\), so that encapsulating an unstable process does not yield a stable process. The \(t\) and \(\alpha\) operators are treated likewise.
ACP^*,Sa\leq(A,\gamma)
\begin{tabular}{l}
BPA^*,Sa\leq(A); ACP^*,Sa\leq(A,\gamma) \\
L(x*y) = L(x)*L(y) & L4
\end{tabular}

Table 7: ACP with autonomous actions and iteration

**Theorem 6** Let p and q be closed ACP^*,Sa\leq terms. Then ACP^*,Sa\leq \vdash p \leq q implies \mathcal{P}/\simeq \vdash p \leq q.

**Proof** It follows from Property 5 that it suffices to check the validity of all the axioms. In each case it is straightforward to give a strong bisimulation for the added axioms.

A completeness result can be given too. The proof uses the following conservativity result.

**Theorem 7** Let p and q be closed BPA^*,Sa\leq terms. Then ACP^*,Sa\leq \vdash p \leq q iff BPA^*,Sa\leq \vdash p \leq q.

**Proof** Similar to the proof of Theorem 3.

**Theorem 8** Let p and q be closed ACP^*,Sa\leq terms. Then ACP^*,Sa\leq \vdash p \leq q iff \mathcal{P}/\simeq \vdash p \leq q.

**Proof** Completeness follows immediately from Theorems 1, 5, 6 and 7. See [D'Ar95] for more details.

5 Example: ABP

The benchmark example for process algebra is the Alternating Bit Protocol (ABP). As the ABP has recursion as well as communication, we define the theory ACP^*,Sa\leq by combining ACP^*,Sa\leq and BPA^*,Sa\leq; see Table 7. A process model can be defined as before. Soundness of ACP^*,Sa\leq follows from Theorems 2 and 6 and the validity of axiom L4. ACP^*,Sa\leq is a conservative extension of ACP^*,Sa\leq (and thus of BPA^*,Sa\leq). A finite complete axiomatization does not exist. Inequality II and hence equation RSP^* are still valid. Furthermore, Property 4 can be generalized to closed ACP^*,Sa\leq terms.

The specification of the ABP usually is of the form (\sum_{d:D} in_d \cdot out_d)^*\delta, a never-ending loop of reading a message at one end and producing the same message at the other end. Verifications based on abstraction give an implementation with internal actions renamed to \tau and use the special properties of the silent action \tau to prove the implementation equal to the specification. Use is made of a “fairness” axiom, stating something like \epsilon(\tau*x* y) = \epsilon*y. Although such a fairness axiom is sound in many bisimulation-oriented models, one may disagree with it and use more discriminating divergence-sensitive models instead, in which case the specification is modified to something like (\sum_{d:D} in_d(\tau*\tau)out_d(\tau*\tau))^*\delta.

In this paper the latter approach is followed, but without silent actions and using pre-abstraction instead. Internal actions are renamed into an autonomous, but visible internal action \tilde{\tau}. Our specification ABP_{spec} now reads (\sum_{d:D} in_d(\tilde{\tau}^*\tau^*)out_d(\tilde{\tau}^*\tau^*))^*\delta, i.e., between any two external actions, a terminating \tilde{\tau}-loop may take place. An implementation
ABP\textsubscript{imp} must satisfy $\text{ABP}_{\text{imp}} \leq \text{ABP}_{\text{spec}}$. A proper implementation never loses the option of proceeding to a next external action, so it should be of the form \text{CYC}^*\delta with \text{CYC} \leq \sum_{d,D} \text{in}_d((\text{I}^*\text{T})\text{out}_d((\text{I}^*\text{T}) and \text{CYC} \subseteq \text{I}^*\text{T}. It follows from the (generalized) Property 4 that this can be established by only using the axioms from $\text{ACP}^\text{aa}S + \text{II}$ in the verification and avoiding the use of $\text{III}$ and $\text{IIR}$.

![Figure 1: ABP implementation.](image)

We adapt the implementation of the ABP given in [BaWe90]. It consists of a sender $S$, a receiver $R$, a message channel $K$ and an acknowledgement channel $L$. See Figure 1. Let $D$ be the finite set of data to be transmitted. For acknowledgements, the element $\text{ack} \notin D$ is added and for perturbed communications the element $\text{junk} \notin D \cup \{\text{ack}\}$. The set $B == \{0, 1\}$ contains the alternating bit. For $b \in B$, its negation $1 - b$ is written $\overline{b}$. We define the set $M$ of messages as $(D \cup \{\text{ack}\}) \times B \cup \{\text{junk}\}$. For $d : D \cup \{\text{ack}\}; b : B$, we denote the message $(d, b)$ by $d_b$. The set $K == \{1, 2, 3, 4\}$ defines the communication ports. We add the rules for standard read/send communication. For $k : K; m : M$, $(rk(m), sk(m)) == (m, m)$. All other communications are set to $D$. Let $H == \bigcup_{k,K; m,M}(d_b, m)$ be the set of communication actions and $I == \{k : K; m : M \cdot c_k(m)\}$ the set of internal actions. Now we can define $\text{ABP}_{\text{imp}}$ as follows. For $b \in B$,

$\text{ABP}_{\text{imp}} = (S \alpha I \alpha \delta_H(S\text{|}\text{|I}\text{|R}))$

$S = (S_0 \cdot S_1)^*\delta$

$S_0 = \sum_{d,D} \text{in}_d \cdot s_1(d_b) \cdot ((r_4(\text{ack}_b) + r_4(\text{junk}))s_1(d_b))r_4(\text{ack}_b))$

$R = (R_0 \cdot R_1)^*\delta$

$R_0 = ((\sum_{d,D} r_2(d_b) + r_2(\text{junk}))s_3(\text{ack}_b))\cdot ((\sum_{d,D} r_2(d_b) \text{out}_d)s_3(\text{ack}_b))$

$K = (\sum_{m,M} r_3(m)(\overline{I} \cdot s_2(m) + \overline{I} \cdot s_2(\text{junk})))^\text{I} \cdot \text{I}$

$L = (\sum_{m,M} r_3(m)(\overline{I} \cdot s_2(m) + \overline{I} \cdot s_2(\text{junk})))^\text{I} \cdot \text{I}$

By standard techniques and $\text{RSP}^\ast$, one derives

$\text{ABP}_{\text{imp}} = (\sum_{d,D} \text{in}_d \cdot (I^* \text{T}') \text{out}_d \cdot (I^* \text{T}'))^\ast \delta$

Here, $\bar{t}^k$ is the abbreviation of $\bar{t} \cdot \bar{t} \ldots \bar{t}$ iterated $k$ times. The fact that $\text{ABP}_{\text{imp}} \leq \text{ABP}_{\text{spec}}$ follows from the following derivation, showing that for any $k, l, m > 0$, $\bar{t}^k(\bar{t}^l \text{T}^m) \leq \bar{t}^k \bar{t}^l \text{T}^m$. By $\text{BKS}^1, \bar{t}^k \bar{t}^l = \bar{t}^{k+l}$ and $\bar{t}^l(\bar{t}^k \bar{t}^l) = \bar{t}^{k+l} \bar{t}^{l-k}$. By induction, and the transitivity of $\leq$, we conclude that for all $k > 0$,

$\bar{t}^k(\bar{t}^l \bar{t}^m) \leq \bar{t}^{k+l} \bar{t}^m$.

By $\text{BKS}^1$, $\bar{t}^k \bar{t}^l = \bar{t}^{k+l}$. So from $\text{I1}$ and $\text{I2}$, $\bar{t} \leq \bar{t}^k \bar{t}^l$ and $\bar{t}^{k+l} \bar{t}^m$. By induction, and the transitivity of $\leq$, we conclude that for all $k > 0$,

$\bar{t}^k \bar{t}^l \leq \bar{t}^{k+l} \bar{t}^m$.

Using these results, $\text{A3}, \text{BKS}^1$ and $\text{I1}$, we deduce that for any $l, m > 0$,

$\bar{t}^l(\bar{t}^m) \leq \bar{t}^l \bar{t}^m.$

Applying $\text{II}$, we obtain

$\bar{t}^{l+m} \leq \bar{t}^{l+m}.$

This result, the fact that for all $k > 0$, $\bar{t}^k(\bar{t}^l \bar{t}^m) \leq \bar{t}^{k+l} \bar{t}^{l-m}$ and the transitivity of $\leq$ yield

$\bar{t}^k(\bar{t}^{l+m}) \leq \bar{t}^{k+l} \bar{t}^{l+m}$.

This concludes our verification. Since the verification took place within $\text{ACP}^\text{aa}S + \text{II}$, the implementation $\text{ABP}_{\text{imp}}$ is a proper one.
6 Conclusions and Further Work

Actions with special properties do form an enrichment of process algebra. The present paper discusses autonomous actions, which form an alternative to the "partial choice" operator of [BaBe94]. This leads to an algebra with similar expressive power but with simpler axioms and models. Terms in the theory BPA\_p, Basic Process Algebra with partial choice, can be expressed in BPA\_aa by means of a single autonomous action \( \mathit{\hat{a}} \) and following the p-graph construction of [BaBe94]. For example, the expression \((a \oplus b) + c\) is mapped to \( \mathit{I}(a + c) + \mathit{I}(b + c) \). It is not clear whether a converse mapping can be constructed.

For reasoning with infinite processes, the II, IIL and IIR inequalities have been introduced. These inequalities can be carried over to other simulation-oriented preorders and can probably be widened in scope to (guarded) recursion. Models that do satisfy II but not IIL and IIR are worth investigating.

A possible extension of autonomous actions is an autonomous "true deadlock" \( \mathit{\hat{\delta}} \). A process containing \( \mathit{\hat{\delta}} \) as summand may autonomously refuse to perform any further actions. So \( \mathit{\hat{\delta}} \) does not vanish in a sum context like \( \delta \). One can also imagine other kinds of special actions, like delayable actions of the form \( \mathit{\hat{a}} \) or irrevocable actions of the form \( \mathit{\hat{a}} \) that e.g. satisfy the equations \( \mathit{\hat{a}} \cdot x + x \cdot \mathit{\hat{a}} = \mathit{\hat{a}} \cdot x \) (which is valid for the silent step in weak bisimulation) and \( x \cdot \mathit{\hat{a}} \cdot y + x \cdot \mathit{\hat{a}} \cdot z = x \cdot (\mathit{\hat{a}} \cdot y + \mathit{\hat{a}} \cdot z) \) (which is valid in ready trace semantics, cf. [Gla90]). The details of such extensions have to be investigated yet.

Another future extension is the addition of the silent action \( \mathit{\tau} \). The notion of branching bisimulation [GiWe99] gives an interesting algebra with two different silent actions: the nearly autonomous \( \mathit{\tau} \) and the fully autonomous \( \mathit{\tau} \). Although the ABP example shows that silent actions and full abstraction have ceased to be necessary for some verifications, they can nevertheless simplify matters considerably.

The theory in this paper can be applied to Petri Nets like in [BaVo95] and used to discriminate between consumptions and productions. Unlike consumption, production of tokens cannot be blocked and can therefore be considered as autonomous in the sense of the present paper. The approach in this paper leads to an (in)equational theory for nets that identifies more processes than in [BaVo95]. The advantages with respect to software engineering as stated in the introduction also apply here.

References


BaWe90 J.C.M. Baeten, W.P. Weijland: Process Algebra, Cambridge University Press 1990

D'Ar95 P.R. D'Argenio: *A General Conservative Extension Theorem in Process Algebras with Inequalities*, these proceedings


Sew94 P. Sewell: *Bisimulation is not (First Order) Equationally Axiomatizable*, in: LICS 1994

Abstract

Variable binding input actions in process algebra expressions can be characterized by early as well as by late bisimulation, where the distinction is concerned with whether or not variables are instantiated when considering process equivalence. Baeten and Bergstra have given an axiomatization of the concepts of late and early bisimulation. We illustrate their method by an example, provide the necessary intuition, formulate correctness properties, list errata, and discuss possibilities for future research.

Note: Supported by NWO, the Netherlands Organization for Scientific Research, project HOOP – Higher-Order and Object-Oriented Processes.

1 Introduction

In process algebra, input actions can have a variable-binding effect, i.e., some value is read and “assigned” to a variable, which later on in the process-expression can be used. A distinction can then be made between “late” and “early” instantiation of variables, a distinction which has consequences for the notion of equality and bisimulation of processes. Baeten and Bergstra have proposed [BB94] a general framework, Functional Prefix Algebra, which they use to come up with various algebraic characterizations of late and early input actions, thereby cleverly circumventing problems concerned with free and bound variables. Their approach is simple and elegant, but not so easy to grasp when you only see the axioms. We discuss one running example for all specifications given, show how the algebras work by executing the specifications using term rewriting, formulate the claims that should be proven in order to assess the correctness of all specifications, and finally propose directions for further research.

This paper is to be read in combination with [BB94]. Before studying that paper, you might find it helpful to make the corrections suggested in Appendix B.
1.1 Motivating Example

The following example, taken from [MPW91, p.46], illustrates the differences between early and late input (here we are using CCS notation):

\[
R = r(x).\left(\text{if } x = 2 \text{ then } P \text{ else } Q\right) + r(x).0
\]

\[
S = r(x).\left(\text{if } x = 2 \text{ then } P\right) + r(x).\left(\text{if } x \neq 2 \text{ then } Q\right)
\]

In [Mil89] the early approach is taken, and \(R\) and \(S\) are equal (i.e., they are strongly bisimilar). An input action \(r(x)\) is interpreted as an abbreviation for a sum of individual read actions for every possible data value for \(x\), say \(\{1, \ldots, N\}\), thus:

\[
r(x).Y = \sum_{n=1}^{N} r_n.Y[n/x]
\]  

Hence, \(2 \times N\) steps are possible from both \(R\) and \(S\), as shown in Figure 1. The steps lead to \(P, Q\) or \(0\) nodes, and for both \(R\) and \(S\) the same transitions lead to the same nodes (although they are given in different order). In other words, they can mimic each other, and hence are bisimilar. This interpretation is referred to as early, because the variable \(x\) is instantiated as early as possible.

Figure 1: Early Input Transitions
1.1 Motivating Example

The following example, taken from [MPW91, p.46], illustrates the differences between early and late input (here we are using CCS notation):

\[
R = r(x).(\text{if } x = 2 \text{ then } P \text{ else } Q) + r(x).0 \\
S = r(x).(\text{if } x = 2 \text{ then } P) + r(x).(\text{if } x \neq 2 \text{ then } Q)
\]

In [Mil89] the early approach is taken, and \( R \) and \( S \) are equal (i.e., they are strongly bisimilar). An input action \( r(x) \) is interpreted as an abbreviation for a sum of individual read actions for every possible data value for \( x \), say \( \{1, \ldots, N\} \), thus:

\[
r(x).Y = \sum_{n=1}^{N} r_n.Y[n/x] \tag{1}
\]

Hence, \( 2 \times N \) steps are possible from both \( R \) and \( S \), as shown in Figure 1. The steps lead to \( P \), \( Q \) or \( 0 \) nodes, and for both \( R \) and \( S \) the same transitions lead to the same nodes (although they are given in different order). In other words, they can mimic each other, and hence are bisimilar. This interpretation is referred to as early, because the variable \( x \) is instantiated as early as possible.
equivalent under late bisimulation. Late bisimulation is particularly relevant to mobile
to mobile process calculi, such as the π-calculus [MPW92] (we will not yet consider mobility here —
this is deferred to a forthcoming paper).

1.2 \textit{FPA_{ECA}} and its Subalgebras

Figure 3 gives an overview of the various algebraic specifications we consider, illustrating
which symbols occur in each of them. The first few signatures are more or less standard
\textit{ACP}: the special operations are in \textit{FPA} ("Functional Prefix Algebra") and Prefixing. All
these together constitute a large specification called \textit{FPA_{ECA}}, and we will first try to
understand its initial algebra. Next, we will study four subalgebras of \textit{FPA_{ECA}}, referred to
as \textit{BVMA}, \textit{VMC}, \textit{VPC}, and \textit{VPA} (the names will be explained in Section 3). \textit{BVMA} amounts
to standard \textit{ACP} with Read/Send communication (as in [BK86]); \textit{VMC} is early read with
free and bound variables, while \textit{VPC} is its late counterpart; \textit{VPA}, finally, is an intriguing
setting where we have (late) input actions but no variables.

1.3 Reduced Model Specifications

When considering a sub-signature \( M \in \{ \textit{BVMA, VMC, VPC, VPA} \} \) of \textit{FPA_{ECA}}, we change the
operations not in \( M \) into hidden functions in \textit{FPA_{ECA}}. This amounts to taking the initial

\begin{align*}
\text{ACP} &= \{ \delta, +, \cdot, \parallel, \parallel, \partial_H \} \\
\text{Val} &= \{ 1, \ldots, N \} \\
\text{Var} &= \{ v_1, \ldots, v_m \} \\
D &= \text{Val} \cup \text{Var} \\
\text{Booleans} &= \{ \neg, \wedge, \vee, eq_D(p,q), :\rightarrow, \odot \} \\
\text{Substitution} &= \{ P[p/v], \delta[p/v], q[p/v] \} \\
\text{RSC} &= \{ r_m(i), s_m(i), c_m(i) \} \\
\text{FPA} &= \{ \langle P_1, \ldots, P_N \rangle, \bullet_N, \circ_N, \wedge \} \\
\text{Prefixing} &= \{ er_m(v);_\leftarrow, lr_m(v);_\leftarrow, s_m(i);_\leftarrow, c_m(i);_\leftarrow, r_m, \lambda v \} \\
\text{Restriction} &= \{ \delta \} \\
\text{FPA_{ECA}} &= \text{ACP} \cup D \cup \text{Substitution} \cup \text{Booleans} \\
& \quad \cup \text{RSC} \cup \text{FPA} \cup \text{Prefixing} \cup \text{Restriction} \\
\text{BVMA} &= \text{ACP} \cup \text{Val} \cup \text{Booleans} \cup \text{RSC} \\
\text{VMC} &= (\text{FPA_{ECA}} - \text{FPA}) - \{ \cdot, \partial_H, lr_m(v);_\leftarrow, r_m(i), \lambda v \} \\
\text{VPC} &= (\text{VMC} - \{ er_m(v);_\leftarrow \}) \cup \{ lr_m(v);_\leftarrow \} \\
\text{VPA} &= \text{ACP} \cup \text{Val} \cup (\text{RSC} - \{ r_m(i) \}) \cup \{ r_m \} \cup (\text{FPA} - \{ \circ_N, \wedge \})
\end{align*}
model of $FPA_{ECA}$, written $I(FPA_{ECA})$, and eliminating those operations corresponding to functions to be hidden, a construction called taking the $M$-reduct of $FPA_{ECA}$, written $I(FPA_{ECA})_M$. This does not change the equalities that are valid between the remaining operations; the hidden functions are just invisible, not deleted.

After eliminating operations from $I(FPA_{ECA})$, it is necessary to throw out carrier elements, as for some values (those corresponding to hidden constructor terms) there is no notation anymore. We can make our algebra smaller by reducing the carrier sets, until the minimal subalgebra remains, which is written $(I(FPA_{ECA}))_M$.

In this way, one can characterize the intended algebras for the signatures of $BYMA$, $YMC$, $YPC$, and $YPA$. In a sense, however, this is unsatisfactory, as the specifications contain too many hidden functions and equations. Therefore, as a last step, we will look for shorter, direct, axiomatizations for the reduced models of $BYMA$, $YMC$, $YPC$, and $YPA$. Naturally, the initial models of these direct axiomatizations should be equivalent to the minimal sub-algebras of the reduced models.

Reduced models are well-known in algebraic specification, and discussed, e.g., in [EM85, Section 6.8] or [Wir90, Section 2.2]. The difference between hiding and deleting is elaborated on in [BH93].

1.4 Understanding by Experiment

In order to gain some intuition concerning the specifications of $FPA_{ECA}$, $BYMA$, $YMC$, $YPC$, and $VPA$ as presented in [BB94], we will sometimes explain them by showing what the equations do when they are interpreted as rewrite rules. For instance, when studying $FPA_{ECA}$, we reduce terms $R_E$ and $S_E$ (see Section 1.1) to their normal forms, and see that they are the same (as they should with early input).

We have used the ASF+SDF Meta-environment [BHK89, Kli93] to perform such experiments. ASF+SDF supports execution of specifications based on term rewriting. Moreover, its literate specification facilities (in the sense of [Knu92]) translate ASCII to $\LaTeX$, allowing one to incorporate machine-checked specifications directly as texts in documentation or technical reports.

Whenever we mention "normal forms" in this document, we refer to normal forms obtained by orienting the equations from left to right, and executing them as rewrite rules. To avoid non-termination, we assume rewriting to take place modulo commutativity and associativity of certain operators. To emphasize that we obtained a certain result using rewriting, we will sometimes use an arrow instead of an equality symbol.

2 Functional Prefix Algebra

We first discuss $FPA_{ECA}$ step by step, following the signatures of Figure 3. The definitions of $ACP$ are very similar to the standard ones and not discussed any further (see, e.g.,

---

The full results of these experiments can be inspected at URL http://www.win.tue.nl/win/cs/fm/arie/elate.html.
We start with a setting without atoms, which is gradually extended with atoms (e.g., in the signature of RSC) which all deal with communication. Missing is the communication function $\gamma$: communication between atoms is axiomatized directly. A noteworthy point, finally, is that we restrict ourselves to finite processes, and finite data.

2.1 Booleans, Data Values and Data Variables

For input actions, we need a notion of data values. We restrict ourselves to a finite set $\text{Val}$ of data values, which are assumed to be the numbers $\{1, \ldots, N\}$. We introduce a sort $\text{Bool}$ with constants $T$ and $F$, as well as operators like $\wedge, \vee, \neg$. Equality over data values is a function $\text{eq}_{\text{Val}} : \text{Val} \times \text{Val} \rightarrow \text{Bool}$ which is either $T$ or $F$ for any value $i, j \in \text{Val}$.

We moreover assume a (countably) infinite collection $\text{Var} = \{v, v_1, v_2, \ldots\}$. The union of $\text{Val}$ and $\text{Var}$ is the full data sort $D$. When writing equations involving data, $i, j$ denote elements from $\text{Val}$, $v, w$ from $\text{Var}$, and $p, q$ from $D$.

Over this full sort $D$, we need an equality function. As we do not know whether two variables $v$ and $w$ are equal (we are interested in equality over their instantiated values), we can only give the following equations:

$$
\text{eq}_D(i, j) = \text{eq}_{\text{Val}}(i, j)
$$

$$
\text{eq}_D(p, p) = T
$$

$$
\text{eq}_D(p, q) = \text{eq}_D(q, p)
$$

Thus, a term like $\text{eq}_D(v_1, v_2)$ is neither equal to $T$, nor to $F$: The definition of $\text{eq}_D$ is not sufficiently complete.

This has consequences for the axiomatization of other operators, such as $\vee, \wedge, \neg$, etc. Just specifying these operators for the $T$ and $F$ cases is not sufficient. For example, when we will try to prove the equality between $R_E$ and $S_E$ (our example from Section 1.1), we will also need ($\beta$ a variable over $\text{Bool}$):

$$
\beta \wedge \beta = \beta
$$

(6)

$$
\beta \wedge \neg \beta = F
$$

(7)

In [Si064] some equational bases for Boolean algebras are discussed in more detail.

In [BB94] the if-then and if-then-else operators (written $\rightarrow$ and $\leftarrow$) over processes are introduced as well. The if-then operator is equal to $\delta$ if the condition is false. Again, the non-standard elements force us to give several extra equations indicating how certain combinations of operators can be eliminated. Two equations not mentioned in [BB94, Section 3.2] (but again needed to show $R_E = S_E$) are:

$$
\beta_1 : \rightarrow (\beta_2 \rightarrow X) = (\beta_1 \wedge \beta_2) : \rightarrow X
$$

(8)

$$
\beta_1 : \rightarrow X + \beta_2 : \rightarrow X = (\beta_1 \vee \beta_2) : \rightarrow X
$$

(9)

These equations are provided in [BB92].

In addition to Booleans and Data, we need a substitution operator, which changes a variable occurring in a process, Boolean expression, or data expression into a new data element (either a value or a variable).
sorts : $\text{Port}$
functions : $\text{r}_*(-), \text{s}_*(-), \text{c}_*(-) : \text{Port} \times \text{Val} \rightarrow \mathbb{A}$
equations : $\text{r}_m(i) \mid \text{s}_m(i) = \text{c}_m(i)$

$\text{r}_m(i) \mid \text{s}_k(j) = \delta$

$\text{r}_m(i) \mid \text{r}_k(j) = \delta$

$\text{s}_m(i) \mid \text{s}_k(j) = \delta$

$\text{c}_m(i) \mid a = \delta$

Figure 4: RSC Communication Primitives

2.2 Read/Send Communication

In ACP, communication involving data typically comes with the Read/Send communication primitives shown in Figure 4, introduced in [BK86]. The axioms are standard (although here not formulated using the $\gamma$ notation). We explicitly mention port names as a sort, where $k, m$ are variables ranging over this sort. It is important to realize that these primitives only deal with values, not with variables! In the next sections, we will see how input actions (or at least the early ones) involving variables can be translated into the Read/Send primitives.

The standard CCS restriction operation [Mil89] can be defined on top of these Read/Send primitives (translating to $\delta$):

$$P \setminus \delta = P \setminus \text{Port} \rightarrow P$$

$$X \setminus \delta m = \partial_\{\text{r}_m(i), \text{s}_m(i)|i \in \text{Val}\}(X)$$

2.3 Sequences and Prefixing

The functions introduced so far were probably more or less familiar: from now on we deal with operations especially invented for functional prefix algebra. First, we need a new sort $P^N$, with the following constructor:

$$\langle \_\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \rangle : P \times \cdots \times P \rightarrow P^N$$

which builds a sequence of $N$ processes (recall that $N$ is the number of data values). It will be used to represent a family of processes indexed by the value of some variable. It is best understood in combination with an expansion operator written $\lambda v.X$, which takes a variable $v$ and a process $X$, and produces a sequence of $N$ variants of $X$, one for each instantiation of $X$:

$$\lambda v.X = \langle X[1/v], \ldots, X[N/v] \rangle$$

$^2$In such a setting there is no need to regard equations as "axiom schemas".
When modelling input operations, we will prefix such sequences with atomic read actions. To that end, we introduce

\[ r_\cdot: \text{Port} \rightarrow A^e \]

an atomic action which just models "I am going to do some kind of read". It can be put in front of process sequences by means of the following two prefixing operators:

\[ o_N \cdot, \bullet_N \cdot : A \times P^N \rightarrow P \]

The \( o_N \) operator is called early functional prefix, the \( \bullet_N \) late functional prefix. The following axiom takes care that an early prefix can always be eliminated:

\[ a \circ_N (X_1, \ldots, X_N) = (a^1) \cdot X_1 + \cdots + (a^N \cdot X_N) \quad (11) \]

where an extra operator

\[ \land: A \times \text{Val} \rightarrow A \]

is used. This operator can be paraphrased as "confront an action with its ith data value". It can be defined for various atomic actions. The crucial equation is for the \( r_m \) action from above:

\[ r_m \land i = r_m(i) \quad (12) \]

The remaining equations defining \( \land \) take care that it is equal to \( \delta \) for other atoms. With the \( r_m(i) \) occurring in the right-hand side, we return to our simple Read/Send communication primitives, and combining (11) and (12) we have, e.g.:

\[ r_m \circ_N (X_1, \ldots, X_N) \rightarrow r_m(1) \cdot X_1 + \cdots + r_m(N) \cdot X_N \]

Such an elimination is in general not possible for the late prefix.\(^3\) Therefore, the \( \bullet_N \) operator acts like a "normal form" or equivalence class representative, which is not equal to a simpler form. Only under suitable circumstances, i.e., when we know that we are indeed communicating, the \( \bullet_N \) can be translated to an early prefix. Axioms expressing this are, e.g.:

\[ (a \bullet_N F) \mid b = (a \circ_N F) \mid b \]

\[ (a \bullet_N F) \mid b \cdot X = (a \circ_N F) \mid b \cdot X \]

Moreover, certain combinations can be transformed, axiomatized, e.g., by:

\[ (a \bullet_N (X_1, \ldots, X_N)) \parallel Y = a \bullet_N (X_1 \parallel Y, \ldots, X_N \parallel Y) \]

Finally, one type of communication is guaranteed to be non-effective:

\[ (a_1 \bullet_N F_1) \mid (a_2 \bullet_N F_2) = \delta \]

This axiom is baptized the Early Communication Axiom (ECA), which states that two late read actions have no communication possibility.

The above should explain the intuition of the various prefixing and sequencing operators. The complete set of axioms is given in [BB94, Table 4], except for \( r_m \) and \( \lambda \) which are specified in [BB94, Table 8].

\(^3\)Observe that [BB94, Section 3.8] indeed does not list \( \bullet_N \) as an operator that can be eliminated (it does not list \( \lambda \) either, but that one can be eliminated).
2.4 Early and Late Read

2.4.1 Axiomatization

With the above preliminaries, defining early and late input actions is straightforward. We introduce them as variable binding prefix operators, with the following signature:

\[ \text{er}_\text{m}(\text{v}); X = r_m \circ_N \lambda v. X \]  
\[ \text{lr}_\text{m}(\text{v}); X = r_m \bullet_N \lambda v. X \]  

The axioms now simply are:

\[ \text{erm}(\text{v}); X = r_m \circ_N \lambda v. X \]  
\[ \text{lr}_\text{m}(\text{v}); X = r_m \bullet_N \lambda v. X \]

For the early read, we can prove from (13), (10), (11), and (12) the following identity:

\[ \text{erm}(\text{v}); Y = \sum_{n=1}^{N} r_m(n) \cdot Y[n/v] \]

which we encountered before as an informal characterization of the early read.

2.4.2 The if-then-else Example Revisited

Let us keep our promises, and see how the early and late versions of \( R \) and \( S \) behave:

\[ R_E = \text{erm}(\text{v}); \{P \circ_D (\text{v},2); Q\} + \text{erm}(\text{v}); \delta \]
\[ S_E = \text{erm}(\text{v}); (\text{eq}_D (\text{v},2); \rightarrow P) + \text{erm}(\text{v}); (\neg \text{eq}_D (\text{v},2); \rightarrow Q) \]

With the intuition gained from the previous section, it should be easy to see that:

\[ R_E \rightarrow r_m(1) \cdot Q + r_m(2) \cdot P + r_m(3) \cdot Q + \ldots + r_m(N) \cdot Q + r_m(1) \cdot \delta + r_m(1) \cdot \delta + \ldots + r_m(N) \cdot \delta \]

Likewise, we have:

\[ S_E \rightarrow r_m(1) \cdot \delta + r_m(2) \cdot P + r_m(3) \cdot \delta + \ldots + r_m(N) \cdot \delta + r_m(1) \cdot Q + r_m(2) \cdot \delta + r_m(3) \cdot Q + \ldots + r_m(N) \cdot Q \]

and by commutativity and associativity of the + we have equality of \( R_E \) and \( S_E \).

For the late version, we have:

\[ R_L = \text{lr}_m(\text{v}); \{P \circ_D (\text{v},2); Q\} + \text{lr}_m(\text{v}); \delta \]
\[ S_L = \text{lr}_m(\text{v}); (\text{eq}_D (\text{v},2); \rightarrow P) + \text{lr}_m(\text{v}); (\neg \text{eq}_D (\text{v},2); \rightarrow Q) \]
\[ R_L \rightarrow r_m \bullet_N \langle Q, P, Q, \ldots, Q \rangle + r_m \bullet_N \langle \delta, \ldots, \delta \rangle \]
\[ S_L \rightarrow r_m \circ_N \langle \delta, P, \delta, \ldots, \delta \rangle + r_m \circ_N \langle Q, \delta, Q, \ldots, Q \rangle \]

Here the "normal forms" are different. Can we conclude that the \( R_L \) and \( S_L \) must be different as well? A demonstration of that requires a careful case distinction of all equations given in [BB94]. The intuition given by the above normalization should convince the reader that it is safe to conclude that \( R_L \neq S_L \).

This example illustrates the intended behavior of the equations: the original terms with input actions are reduced to terms without the variable \( v \) occurring in it.
2.4.3 Bound Variables and $\alpha$-Conversion?

This axiomatization of early and late read does not explicitly mention free or bound variable occurrences, nor does it include a rule for $\alpha$-conversion. Nevertheless, the $er_m(v); X$ and $lr_m(v); X$ expressions do have a binding effect in the process $X$. Can this be correct?

Observe that [BB94] does not give equations specifying the effect of substituting over $er_m(v); X, lr_m(v); X$, and $\lambda v.X$. In spite of this, the substitution operator can always be eliminated [BB94, Section 3.8]. This is because the operators for which no substitution equations are given can themselves be eliminated (e.g., the $er_m(v); X$ is equal to a summation for every data element, and substitution is defined for the choice operator). In other words: "normalization" of terms involving bound variables can only be achieved by first eliminating (i.e., expanding) all operations involving bindings. As a result, substitutions cannot be used to change bound variables.

Secondly, let us study the counterpart of the $\alpha$-conversion rule in this algebra. If we define free variables in the usual way, we can use structural induction to prove:

$$w \notin FV(X) \Rightarrow er_m(v); X = er_m(w); (X[w/v]) \quad (15)$$

This is fairly obvious, as (provided $w \notin FV(X)$)

$$\lambda w.X = (X[1/v], \ldots, X[N/v])$$

$$= (X[w/v][1/w], \ldots, X[w/v][N/w])$$

$$= \lambda w.(X[w/v])$$

Naturally the same holds for the late read operation.

2.5 Process Prefix and Further Extensions

The $\bullet_N$ and $\circ_N$ operators are "action prefixes" in the sense that their first argument is just a single atomic action. To illustrate the generality of the FPAECA setting, we show how it can be used to arrive at a "process prefix" situation, where an arbitrary process can be used to read variables which are used in a subsequent process. The signature we need includes two new core atoms:

$$er_\cdot(x), lr_\cdot(x) : Port \times Var \rightarrow A^c$$

and a process combinator which should have the binding effect:

$$\_\cdot : P \times P \rightarrow P$$

With this signature we can write all terms we could express before, but some extra terms as well. We therefore need a few extra equations:

$$er_m(v) | a = \delta \quad (16)$$

$$lr_m(v) | a = \delta \quad (17)$$
which state that read actions in isolation cannot communicate. For \( \text{er}_m(v); X \) we have the same equations (13) and (14), but these are constructed from the new operators now (they are parsed differently). For the process prefix combinator we further have the following equations:

\[
\begin{align*}
(X + Y); Z &= X; Z + Y; Z \\
(X \cdot Y); Z &= X; (Y; Z) \\
(a \cdot_N (X_1, \ldots, X_N)); Y &= a \cdot_N (X_1; Y, \ldots, X_N; Y)
\end{align*}
\]

For other cases, the ";" operator can be translated directly to the ",", the normal ACP sequential composition.

An example term, which could not be expressed with only action prefix, is the following, where \( P(v, w) \) is some term containing variables \( v, w \).

\[
(er_1(v) \parallel er_2(w)); P(v, w)
\]

With the equations from above, we can translate this, using [CM1], (16), (18), (19) to:

\[
= (er_1(v) \cdot er_2(w) + er_2(w) \cdot er_1(v) + er_1(v) \mid er_2(w)); P(v, w)
= (er_1(v) \cdot er_2(w)); P(v, w) + (er_2(w) \cdot er_1(v)); P(v, w)
= er_1(v); (er_2(w); P(v, w)) + er_2(w); (er_1(v); P(v, w))
\]

From here on, we are back at notation from Section 2.4, and the terms behave as before. The new notation allows one to mix early and late inputs and to merge them in parallel.

In [BB94], the action prefix setting is also being used to axiomatize restricted input, Hoare’s input action, prefix iteration, exits, and CSP synchronization merge. Once the reader grasps the early and late input actions these extensions are straightforward, so there is no need for us to dwell on these issues in this document.

## 3 Reduced Models of FPA

Now that we have seen the full FPAECA (in Sections 2.1 to 2.4), we can study four interesting subalgebras, as indicated by the signature overview of Figure 3.

### 3.1 BVMA

The signature of Basic Value Matching Algebra (BVMA) gives us ACP with existing Read/Send communication, as discussed, e.g., by [BK86]. The direct axiomatization is simply obtained by taking the axioms from ACP, Val, Boolean, and RSC.

BVMA should not include variables nor substitutions (as suggested in [BB94]) if it is to be the algebra for Read/Send Communication from [BK86], as these will introduce non-standard elements in the Boolean.
3.2 VMC

The signature of Value Matching Calculus gives us an algebra which is very close to CCS under early bisimulation (i.e., CCS as discussed in [Mil89]). ACP's sequential composition is dropped, and replaced by an early read prefix operator.

We obtain a direct axiomatization by taking some of the theorems we could prove (in Section 2.4) as our new axioms. For example, we now adopt

\[ w \not\in FV(X) \Rightarrow \text{erm}(v); X = \text{erm}(w); (X[w/v]) \]

which is exactly equation (15) from Section 2.4. To make this possible, we have to distinguish between bound and free variable occurrences, so we introduce a function

\[ FV(\_): P \rightarrow P(\text{Var}) \]

with straightforward axiomatization. Conditional equations then are used to express equalities like the \( \alpha \)-conversion above.

In the full \( \text{FPAECA} \) specification, early read was axiomatized by "exploding" it to a summation for all possible inputs. In the setting of VMC, the same effect is achieved in smaller steps, by means of the early input axiom \( \text{EIA} \):

\[ \text{erm}(v); X + \text{erm}(v); Y = \text{erm}(v); X + \text{erm}(v); Y + \text{erm}(v); \{ X \triangleleft \text{eqD}(v, i) \triangleright Y \} \]

This axiom states that if we have a choice between two read actions on \( v \), we can add a third summand, in which we test for equality with a particular value \( i \). Intuitively, it can be used to add a summand for any data value \( i \), until we have the full summation again.

Let us see how this works for our \( R, S \) example. The point is that we can add \( R_E \) to \( S_E \) and vice versa, and hence they are equal. For example, we can add \( (\text{eqD}(v, 2); \rightarrow Q) \) to \( R_E \) as follows:

\[ R_E = \text{erm}(v); \{ P \triangleleft \text{eqD}(v, 2); \rightarrow Q \} + \text{erm}(v); \delta \]
\[ = R_E + \text{erm}(v); \{ \delta \triangleleft \text{eqD}(v, 2); \rightarrow \{ P \triangleleft \text{eqD}(v, 2); \rightarrow Q \} \} \]
\[ = R_E + \text{erm}(v); \{ (\text{eqD}(v, 2); \rightarrow \delta) + (\neg\text{eqD}(v, 2); \rightarrow \{ P \triangleleft \text{eqD}(v, 2); \rightarrow Q \}) \} \]
\[ = R_E + (\neg\text{eqD}(v, 2); \rightarrow Q) \]

were we have been using \( \text{EIA} \) (filling in 2 for \( i \)) and several equations over the Booleans such as (6) and (8).

3.3 VPC

The Value Passing Calculus (VPC) models CCS under late bisimulation. Its signature is that of VMC, but with the early read operation replaced by the late input. Its direct axiomatization is exactly the same as the one for VMC, with early reads replaced by late ones, and, most importantly, without the Early Input Axiom.

\footnote{The \( \text{EIA} \) as formulated in [BB94] uses a \( p \) instead of an \( i \) in the third summand, thus allowing tests involving arbitrary free variables. This is incorrect as (i) it does not follow from the equations over \( \text{FPAECA} \), and (ii) it conflicts with the aim of equating a term with only bound variables to one without any variable. Also observe that the \( \text{EIA} \) characterizations of [MPW91], [PS93, Law SP] are based on the \( \pi \)-calculus, and therefore do not distinguish variables and values.}
3.4 VPA

Recall from Section 2.4.2 that the normal forms of $R_L$ and $S_L$ did not contain variable occurrences of $v$. This observation is used in the last algebra we consider, *Value Passing Algebra*. Its signature only contains the functions needed to write the normal forms of $R_L$ and $S_L$, i.e., we do not have variables, conditionals, nor the $lr_m;(X)$ operation: instead we only have a single read atom $r_m$ which can be put as prefix before a sequence. The direct axiomatization merely indicates which communications are known to be $\delta$, and which ones can be successful.

4 Assessment

The following propositions, which have not (yet) been formally proved, formulate four properties needed to assess the correctness of the various specifications discussed.

**Definition 4.1** Let $M$ be any of \{FPAECA, BVMA, VPC, VMC, VPA\}. Define $R_M$ as the term rewriting system (TRS) obtained by orienting all equations of the axiomatization of $M$ from left to right. Rewriting of the $+$ takes place modulo ACI (Associativity, Commutativity, and Idempotency), $|$ modulo AC, and $\cdot$ modulo associativity.

**Proposition 4.2** The following TRSs are strongly normalizing and confluent:

1. $R_{FPAECA}$, $R_{BVMA}$, and $R_{VPA}$.
2. $R_{VPC}$ and $R_{VMC}$ modulo $\alpha$-conversion of bound variables.

**Proposition 4.3** Rewriting a term with only bound variables over $R_{FPAECA}$, $R_{BVMA}$ or $R_{VPA}$ yields a normal form without variables occurring in it.

**Proposition 4.4** Let $M$ be one of the signatures in \{FPAECA, BVMA, VMC, VPC, VPA\}, let $I(M)$ be the initial model of the direct axiomatization of $M$. and let $t_1, t_2$ be terms over $M$. Then

$$\langle I(FPAECA) \rangle_M \models t_1 = t_2 \Leftrightarrow I(M) \models t_1 = t_2$$

**Proposition 4.5** Define labeled transition systems over VPC and VMC following the approach of Appendix A. Then:

1. Equality over terms from VMC coincides with the early bisimulation relation $\sim_E$.
2. Equality over terms from VPC coincides with the late bisimulation relation $\sim_L$. 
5 Concluding Remarks

In this paper, we studied *Functional Prefix Algebra*, as proposed by Baeten and Bergstra [BB94]. Their method is simple and elegant, but their article pays more attention to listing the signatures and axioms than to explaining the underlying intuitions. We remedied this deficiency by presenting an appropriate example, by illustrating the operational behavior of several operators, by emphasizing that variable elimination is the aim of these operators, and by listing the properties that can be used to judge the axiomatizations chosen.

An area for future work might include a generalization to recursively defined (infinite) processes, and to infinite data sorts.

An other challenging extension of the work presented here is in the area of *mobility*. The notions of early and late bisimilarity naturally occur in the π-calculus, and it would be intriguing to study whether a similar approach can be used to arrive at a relatively simple axiomatization of the π-calculus. A first step could be to allow for internal mobility only, as in the πI-calculus of [San95]. It might be wise to start with finite processes only, which in the π-calculus amounts to omitting the replication operator.

Acknowledgments

I would like to thank Jos Baeten, Jan Bergstra, Gerard Kok, and an anonymous ACP’95 referee for numerous helpful comments.

References


A Bisimulations

We briefly sketch how to build an operational semantics for the terms built using \textsc{FPAECA}. We assume the following labels: \{\(r_m(i), s_m(i), c_m(i), r_m(v)\}\}. The SOS rules are mostly straightforward, following the lines of [BV95, Table 5, Table 44]. The more interesting rules are the following four, which are similar to rules with the same names occurring, e.g., in [MPW91, PS93].
Next, we define early and late bisimulations, closely following [PS93]. The definition of early simulation is obtained from the late one by commuting the quantifiers in the second clause.

**Definition A.1** A binary relation $S$ on processes is a late simulation if $PSQ$ implies that

1. If $P \xrightarrow{a} Q$, with $a \in \{r_m(i), s_m(i), c_m(i)\}$ then there exists a $Q', Q \xrightarrow{a} Q'$ and $P'^S(Q')$.

2. If $P \xrightarrow{r_m(v)} P'$ and $v \notin FV(P) \cup FV(Q)$, then there exists a $Q'$ such that $Q \xrightarrow{r_m(v)} Q'$ and for all $i \in Val$, $P'[i/v]S'Q'[i/v]$.

The relation $S$ is a late bisimulation if both $S$ and $S^{-1}$ are late simulations. Two processes $P$ and $Q$ are late bisimilar, written $P \sim_L Q$ if $PSQ$ for some late bisimulation $S$.

**Definition A.2** The definition of early simulation is the same as for late simulation, where clause 2 should be replaced by

2': If $P \xrightarrow{r_m(v)} P'$ and $v \notin FV(P) \cup FV(Q)$, then for all $i$ there exists $Q'$ such that $Q \xrightarrow{r_m(v)} Q'$ and $P'[i/v]S'Q'[i/v]$

Early bisimulation is written $\sim_E$.

**B  Errata**

Before studying [BB94], you should make the following small corrections to it:

- p. 255, Table 4, 6th equation, and p. 266, Table 18, one but last equation: variable $a$ should be a $d$ (encapsulation works on core atoms).
- p. 256, Section 3.1, line above Table 5: Delete “C0,” (there is no such axiom).
- p. 257, Table 7, Booleans. Add the following equations (see [BB92] and our Section 2.1):

\[
\begin{align*}
\beta : \rightarrow \gamma : \rightarrow X &= (\beta \land \gamma) : \rightarrow X \\
\beta : \rightarrow X + \gamma : \rightarrow X &= (\beta \lor \gamma) : \rightarrow X
\end{align*}
\]
• p. 259, Section 3.8, line 5: replace first | by || in list of operators that can be eliminated. Add λ to it, and replace the “;” by er_m(v); tr_m(v); s_m(i); c_m(i); ...
• p. 260, Section 4, line 4: Table 7 should be Table 8.
• p. 261, line 3: Table 9 should be Table 11.
• p. 261, Section 4.3, last + in definition of s_m(v) should be a “=”.
• p. 263, Section 5, line 3: through 3.5 should be “through 3.4” (section 3.5 only contains an example).
• p. 263, Section 5.1. Substitution and Variables should not be included. See Section 3.1 of this document.
• p. 263, Section 5.2, last line. Axiom A7 should not be included.
• p. 264, Table 15 (VMC). p in right-hand side of Early Input Axiom should be an i (see our Section 3.2).
• p. 264, line 1: Table 5 (Booleans) should be Table 6 (Booleans) except for equations
  \[(\beta : \rightarrow X) \cdot Y = \beta \rightarrow X \cdot Y\]
  \[\partial_{[\delta]}(\beta : \rightarrow X) = \beta \rightarrow \partial_{[\delta]}(X)\]
  \[\rho_f(\beta : \rightarrow X) = \beta \rightarrow \rho_f(X)\]
• p. 264 (VMC), line 2: “except for the sixth and the seventh” should be “except for the fourth and fifth” (the equations for \cdot and \bullet_N).
• p. 264 (VMC), Sorts part of signature: include A, N. Constants part of signature: \delta \in P should be “\delta \in A” (for consistency with p.252, Section 2.1).
  An alternative interpretation could be that VMC and VPC do not have atomic actions (as \delta would be the only one). The sort N should be included in any case.
• p.265 (VMC), top: the signature should also include \_ : \rightarrow \_ : B \times P \rightarrow P
• p.266 (VPA), bottom: r_m, s_m(i), c_m(i) should all be \in A^c, rather than \in A.
Process Algebra with Propositional Signals

J.C.M. Baeten
Department of Computer Science, Eindhoven University of Technology,
P.O.Box 513, 5500 MB Eindhoven, The Netherlands

J.A. Bergstra
Programming Research Group, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
and
Department of Philosophy, Utrecht University,
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

We consider processes that have transitions labeled with atomic actions, and states labeled with formulas over a propositional logic. These state labels are called signals. A process in a parallel composition may proceed conditionally, dependent on the presence of a signal in the process in parallel. This allows a natural treatment of signal observation.

Note: This research was supported in part by ESPRIT basic research action 7166, CONCUR2.

1. INTRODUCTION.
This paper can be viewed as a revision and simplification of [BAB92] in which we have introduced so-called signals as labels for states in processes (see also [BR90]). Though useful in a multitude of examples, it has turned out that the mechanism of signal observation of [BAB92] is quite complex. The approach taken was that actions observe signals. What we propose here is to require that the signals are propositions (i.e. elements of a boolean algebra; this is consistent with [BAB92]) and then to use tests to read off information from these signals. In this way, conditions in conditional expressions (written as φ ⇒ x, or x ≤ φ ⇒ y) and propositional signals are complementary. A mechanism to localise or hide the propositional signals is important. In summary, our development is based on the position:

\[ \text{the visible part (signal) of the state of a process is a proposition}. \]

Whatever the merits of this position, what we do establish is that it is a consistent one, and that it allows a wide range of examples.

The introduction of propositional signals in the context of process algebra occurs to us as a necessary step, it completes the picture that emerges if conditional process expressions are introduced. Indeed, consider an expression \( x \leftarrow \phi \rightarrow y \). If \( \phi \) is true or false, this is just \( x \) or \( y \). But in the more general case that \( \phi \) ranges over a class of propositions, what determines the meaning of \( x \leftarrow \phi \rightarrow y \)? An answer is: \( \phi \) is to be evaluated over a state. This leads one into state operators (as in [BAB88]) or global states (see [GRP94]), thus departing from the core of process algebra where every dynamic entity is a process.

So we feel that the primary motivation of this paper is a conceptual one and that additional motivation in terms of potential applications is both premature and superfluous. This is not meant to imply that we are pessimistic about applications. It rather is the case that we would propose to view
process algebra with propositional signals as a subject in pure logic at least initially. Many extensions or modifications can be imagined: first order signals, higher order signals, infinitary and non-classical logics for the entailment relation between signals and conditions, modal and temporal logics for processes with propositional signals.

We are not aware of any previous work aiming at objectives similar to our present ones. The present approach is also followed in [BEP94]. Clearly, our approach, based on ACP [BEK84] can be adapted to CCS [MIL80], MEUE [AUB84] or ATP [NiS94] without much effort. Adaptation to CSP [BRHR84] is more involved due to the different models, based on failure or ready sets.

ACKNOWLEDGEMENT: Helpful remarks by Jan Joris Vereijken (Eindhoven University of Technology) and Joris Boselie (University of Amsterdam) are appreciated.

2. BASIC PROCESS ALGEBRA WITH PROPOSITIONAL SIGNALS.

2.1 BPA WITH INACTION AND NONEXISTENCE.

Let A be a finite set. The elements of A will be called atomic actions. Every atomic action is an element of P, the sort of processes. There are also two binary operators on P, viz. + (alternative composition) and · (sequential composition). The core system BPA (Basic Process Algebra) over this signature has the axioms A1-5 of table 1 below (x,y,z ∈ P). The constant δ denoting inaction (or deadlock) is added to the language with axioms A6,7. In this paper, we introduce a new constant for process algebra, viz. ⊥. This constant stands for nonexistence, we will need it when we introduce signals further on. It is axiomatized by axioms NE1-3 of table 1 (x ∈ P, a ∈ A). Nonexistence stands for an inconsistent state of a process: such a state can never be exited (NE1,2) and also, it is impossible to enter such a state from a consistent state (NE3). This signature and these axioms together constitute the theory BPA⊥, Basic Process Algebra with inaction and nonexistence.

\[
\begin{align*}
    x + y &= y + x & \text{A1} \\
    (x + y) + z &= x + (y + z) & \text{A2} \\
    x + x &= x & \text{A3} \\
    (x + y) + z &= x + (y + z) & \text{A4} \\
    (x \cdot y) \cdot z &= x \cdot (y \cdot z) & \text{A5} \\
    x + \delta &= x & \text{A6} \\
    \delta \cdot x &= \delta & \text{A7} \\
    x + \bot &= \bot & \text{NE1} \\
    \bot \cdot x &= \bot & \text{NE2} \\
    a \cdot \bot &= \delta & \text{NE3}
\end{align*}
\]

TABLE 1. BPA⊥.

2.2 CONDITIONALS.

Besides the sort of processes P, we will have a second sort B. Elements of this sort are propositional logic formulas over a set of basic propositional variables P₁, ..., Pₙ with constants T, F (true, false) and operators ∨, ∧, →, ¬ (disjunction, conjunction, implication, negation). In derivations we can use identities of propositional logic. We use letters φ, ψ to range over B.

As in [BAB92], we use the guarded command \( \ldots .\rightarrow:. B \times P \rightarrow P \). The expression \( \phi .\rightarrow:x \) is read as if \( \phi \) then \( x \). We have the basic axioms of table 2 below, using the numbering of [BAB92].
2.3 Root and Terminal Signal Emission Operators.

The next operators to be introduced are the signal emission operators. \( \checkmark \) is the root signal emission operator and \( \checkmark \) is the terminal signal emission operator (we take this notation from [BEP94]). The intuition behind these operators is that both assign labels (signals) to the states of processes. Root signal emission places a signal at the root node of a process. Terminal signal emission places one and the same signal at each terminal node of a process. If one is interested solely in processes that emit signals exclusively in nonterminal states one may as well forget about the terminal signal emission operator. Leaving out all axioms involving terminal emission from the coming sections one will obtain an appropriate description of root signal emission. The following equations are added to BPA\(_L\) thus obtaining\( \text{BPAps} \) (BPA with Propositional Signals).

\[
\begin{align*}
\phi \xrightarrow{\checkmark} x \cdot y &= \phi \xrightarrow{\checkmark} (x \cdot y) & \text{RSE1} \\
\phi \xrightarrow{\checkmark} x + y &= \phi \xrightarrow{\checkmark} (x + y) & \text{RSE2} \\
\phi \xrightarrow{\checkmark} (\psi \xrightarrow{\checkmark} x) &= (\phi \wedge \psi) \xrightarrow{\checkmark} x & \text{RSE3} \\
\phi \xrightarrow{\checkmark} x &= x & \text{RSE4}
\end{align*}
\]

TABLE 3. Root signal emission.

The first axiom expresses the fact that the root of a sequential product is the root of its first component. Axiom RSE2 can be given in a more symmetric form as follows:

\[
(\phi \xrightarrow{\checkmark} x) + (\psi \xrightarrow{\checkmark} y) = (\phi \wedge \psi) \xrightarrow{\checkmark} (x + y).
\]

This equation depends on the fact that the roots of two processes in an alternative composition are identified. Therefore signals must be combined. The third axiom expresses the fact that there is no sequential order in the presentation of signals. Of course one might imagine that a sequential ordering on signals is introduced, but we think that the introduction of such a sequential ordering is far from obvious (it also leads to problems concerning the associativity of the parallel composition operator).

The combination of the signals is taking 'both' of them whereas \( x + y \) has to choose between \( x \) and \( y \). As an example, consider the following derivation:

\[
a \cdot ((\phi \xrightarrow{\checkmark} b) + (\neg \phi \xrightarrow{\checkmark} b)) = a \cdot ((\phi \wedge \neg \phi) \xrightarrow{\checkmark} b) = a \cdot (F \xrightarrow{\checkmark} b) = a \cdot \bot = \delta.
\]

The last axiom RSE8 is the signal inspection rule. If a signal \( \phi \) is emitted, then \( \phi \) holds in the current state (this is why the signal \( F \) denotes an inconsistent state). Note the following generalisation of RSE8:

\[
\phi \xrightarrow{\checkmark} (\psi \xrightarrow{\checkmark} x) = \phi \xrightarrow{\checkmark} (\phi \rightarrow (\psi \xrightarrow{\checkmark} x)) = \phi \xrightarrow{\checkmark} ((\phi \wedge \psi) \rightarrow x).
\]

The equations below regard terminal signal emission.

An interesting identity that follows is the following:

\[
\phi \xrightarrow{\checkmark} x = (\phi \xrightarrow{\checkmark} \delta) + x.
\]
This equation is indeed very useful for writing efficient process specifications mainly because it allows to a large extent to work with process algebra expressions that are not cluttered with signal emissions.

The axiom system BPAps, Basic Process Algebra with Propositional Signals, consists of all axioms from tables 1-4.

\[(x \cdot y) \downarrow \phi = x \cdot (y \downarrow \phi)\]  \hspace{1cm} (TSE1) \hspace{1cm} \downarrow F = \delta\]  \hspace{1cm} (TSE5)

\[(x + y) \downarrow \phi = x \downarrow \phi + y \downarrow \phi\]  \hspace{1cm} (TSE2) \hspace{1cm} \delta \downarrow \phi = \delta\]  \hspace{1cm} (TSE6)

\[(x \downarrow \phi) \downarrow \psi = x \downarrow (\phi \wedge \psi)\]  \hspace{1cm} (TSE3) \hspace{1cm} \perp \downarrow \phi = \perp\]  \hspace{1cm} (TSE7)

\[x \downarrow T = x\]  \hspace{1cm} (TSE4) \hspace{1cm} (\psi \rightarrow x) \downarrow \psi = \psi \rightarrow (x \downarrow \psi)\]  \hspace{1cm} (TSE8)

\[(x \downarrow \phi) \cdot y = x \cdot (\phi \downarrow \psi)\]  \hspace{1cm} (TRSE1) \hspace{1cm} (\phi \downarrow \psi) \downarrow \psi = (x \downarrow \psi) \downarrow \psi\]  \hspace{1cm} (TRSE2)

**Table 4. Remaining axioms of BPAps.**

2.4 BASIC TERMS.

Define a set of basic terms \(\mathcal{B}\) as follows.

i. \(\perp \in \mathcal{B}\)

ii. \(\phi \in \mathcal{B} - \{F\} \implies \phi \downarrow \delta \in \mathcal{B}\)

iii. \(\phi, \psi \in \mathcal{B} - \{F\}, \alpha \in A \implies (\phi \rightarrow \alpha \downarrow \psi) \in \mathcal{B}\)

Note that each basic term can be written as \(\perp\) or in the form:

\[\zeta \downarrow \delta + \sum_{i=1}^{n} \psi_i \rightarrow a_i \cdot x_i + \sum_{j=1}^{m} \psi_j \rightarrow b_j \downarrow \chi_j\]

or, equivalently,

\[\zeta \downarrow \delta + \left( \sum_{i=1}^{n} \psi_i \rightarrow a_i \cdot x_i + \sum_{j=1}^{m} \psi_j \rightarrow b_j \downarrow \chi_j \right)\]

When a basic term has this form, we call \(\zeta\) its root signal, and the subterms \(\psi_i \rightarrow a_i \cdot x_i, \psi_j \rightarrow b_j \downarrow \chi_j\) its summands.

2.5 BASIC TERM LEMMA. For all closed terms \(s\) there is a basic term \(t\) such that BPAps \(\vdash t = s\).

2.6 STRUCTURED OPERATIONAL SEMANTICS.

We proceed to give the semantics of BPAps using structured operational rules (SOS).

The semantics uses the following predicates and relations on closed terms:

- \(x \phi x'\) term \(x\) can do an a-step under condition \(\phi\) to term \(x'\)
- \(x \psi \psi\) term \(x\) can do a terminating a-step under condition \(\phi\) leaving terminal signal \(\psi\)
- \(s_\phi(x) = \phi\) the root signal of \(x\) is \(\phi\).

Plotkin-style rules for the step relations and step predicates are given in table 5, the rules for the root signal predicate are given in the form of axioms in table 6. This SOS specification is in the path format of [BAV93].
Based on this operational semantics involving conditions on the arrows comes a new definition for bisimulation. Instead of just requiring matching actions, we also require matching conditions; however, one transition on one side may have to be matched with several transitions on the other side, depending on the truth value of the propositional constants. Therefore, the following definition starts from the set of valuations of the propositional constants, i.e. all mappings \( v: \{ P_1, \ldots, P_n \} \rightarrow \text{BOOL} \). Each such mapping naturally extends to a mapping on all formulas. We write \( \phi = \psi \) (also in the rules above) iff for all valuations \( v \), \( v(\phi) = T \) iff \( v(\psi) = T \). Similarly, \( \phi \models \psi \) iff there is a valuation \( v \) with \( v(\phi) = T \) and \( v(\psi) = F \), or \( v(\phi) = F \) and \( v(\psi) = T \).

Then we say that a relation \( R \) on closed terms is a (strong) bisimulation when the following holds:

i. if \( xRy \) then \( Sp(x) = Sp(y) \)

ii. if \( xRy \) and \( x' \models \psi \), then for all valuations \( v \) such that \( v(Sp(x)) \wedge (x' \models \phi) \rightarrow T \), there is a condition \( \psi \) and an expression \( y' \) such that \( v(\psi) = T \), \( y \models y' \) and \( x'Ry' \)

iii. if \( xRy \) and \( y' \models \psi \), then for all valuations \( v \) such that \( v(Sp(y)) \wedge (y' \models \phi) \rightarrow T \), there is a condition \( \psi \) and an expression \( x' \) such that \( v(\psi) = T \), \( x \models x' \) and \( x'Ry' \)

TABLE 5. SOS rules.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x _{\phi,a}^* x' \rightarrow x' )</td>
<td>( x _{\psi,sp}^* x' \rightarrow x' )</td>
</tr>
<tr>
<td>( x <em>{\phi,a}^* x'</em>{,sp} \rightarrow x' )</td>
<td>( x <em>{\psi,sp}^* x'</em>{,sp} \rightarrow x' )</td>
</tr>
<tr>
<td>( x <em>{\phi,a}^* x'</em>{,sp} \rightarrow x' )</td>
<td>( x <em>{\psi,sp}^* x'</em>{,sp} \rightarrow x' )</td>
</tr>
<tr>
<td>( x <em>{\phi,a}^* x'</em>{,sp} \rightarrow x' )</td>
<td>( x <em>{\psi,sp}^* x'</em>{,sp} \rightarrow x' )</td>
</tr>
<tr>
<td>( x <em>{\phi,a}^* x'</em>{,sp} \rightarrow x' )</td>
<td>( x <em>{\psi,sp}^* x'</em>{,sp} \rightarrow x' )</td>
</tr>
<tr>
<td>( x <em>{\phi,a}^* x'</em>{,sp} \rightarrow x' )</td>
<td>( x <em>{\psi,sp}^* x'</em>{,sp} \rightarrow x' )</td>
</tr>
<tr>
<td>( x <em>{\phi,a}^* x'</em>{,sp} \rightarrow x' )</td>
<td>( x <em>{\psi,sp}^* x'</em>{,sp} \rightarrow x' )</td>
</tr>
</tbody>
</table>

TABLE 6. Root signal operator.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( sp(\bot) = F )</td>
<td>( RSO0 )</td>
</tr>
<tr>
<td>( sp(\bot) = T )</td>
<td>( RSO1 )</td>
</tr>
<tr>
<td>( sp(x + y) = sp(x) \wedge sp(y) )</td>
<td>( RSO2 )</td>
</tr>
<tr>
<td>( sp(x) = sp(x) )</td>
<td>( RSO3 )</td>
</tr>
<tr>
<td>( sp(y) \wedge sp(x) )</td>
<td>( RSO4 )</td>
</tr>
<tr>
<td>( sp(x) \mapsto sp(y) )</td>
<td>( RSO5 )</td>
</tr>
<tr>
<td>( sp(x) \mapsto sp(x) )</td>
<td>( RSO6 )</td>
</tr>
</tbody>
</table>
iv. if \( x \mathbin{R} y \) and \( x \overset{\phi}{\to} \psi \) then for all valuations \( \nu \) such that \( \nu(\mathcal{S}_p(x) \land \phi) = T \), there are conditions \( \phi', \psi' \) with \( \nu(\phi') = T, \psi = \psi' \) and \( y \overset{\phi'}{\to} \psi' \).

v. if \( x \mathbin{R} y \) and \( y \overset{\phi}{\to} \psi \) then for all valuations \( \nu \) such that \( \nu(\mathcal{S}_p(y) \land \phi) = T \), there are conditions \( \phi', \psi' \) with \( \nu(\phi') = T, \psi = \psi' \) and \( x \overset{\phi'}{\to} \psi' \).

We call two expressions \( x, y \) (strongly) bisimilar,notated \( x \equiv y \), if there is a (strong) bisimulation relating \( x \) and \( y \).

2.7 PROPOSITION. Bisimulation is an congruence relation on process expressions.
As a consequence, we can consider the algebraic structure \( \mathbb{P}/\mathbin{\equiv} \) of process expressions modulo bisimulation equivalence.

2.8 THEOREM (SOUNDNESS). \( \mathbb{P}/\mathbin{\equiv} \models \text{BPAPS} \).

For basic terms, there is a direct relation between syntax and semantics.

2.9 LEMMA. Let \( t \in \mathbb{B} \).

i. The root signal of \( t \) is \( \mathcal{S}_p(t) \).

ii. \( t \overset{\phi}{\to} \psi \)iff \( \phi :\to a \psi \) is a summand of \( t \).

2.10 THEOREM (COMPLETENESS). Let \( t, s \) be two closed BPAPS terms. Then \( x \equiv y \) implies \( \text{BPAPS} \models t \equiv s \). As a corollary, we have \( \mathbb{P}/\mathbin{\equiv} \models t \equiv s \iff \text{BPAPS} \models t \equiv s \) for all closed \( t, s \).

2.11 GLOBAL SIGNAL EMISSION.
In the next section, we will extend BPAPS with parallel composition. There, we will need as an extra operator the global signal emission operator, that adds a signal to each state of a process. We give axioms for this operator in table 7, and semantical rules in table 8. With the help of the global signal emission operator, we can define a notion of invariance:

**DEFINITION:** \( \phi \) is an invariant of \( x \) if \( \phi \mathbin{\downarrow} x = \phi \mathbin{\downarrow} x \).

\[
\begin{align*}
\phi \mathbin{\downarrow} v & \downarrow = \downarrow & \text{GSE0} \\
\phi \mathbin{\downarrow} v a & = \phi \mathbin{\downarrow} v a \mathbin{\downarrow} \phi & \text{GSE1} \\
\phi \mathbin{\downarrow} v (x + y) & = (\phi \mathbin{\downarrow} v x) + (\phi \mathbin{\downarrow} v y) & \text{GSE2} \\
\phi \mathbin{\downarrow} v (x \cdot y) & = (\phi \mathbin{\downarrow} v x) \cdot (\phi \mathbin{\downarrow} v y) & \text{GSE3}
\end{align*}
\]

**TABLE 7. Global signal emission.**

\[
\begin{align*}
x \overset{\phi}{\to} x', \psi \mathbin{\downarrow} \mathcal{S}_p (x) \mathbin{\Rightarrow} F & \quad \psi \mathbin{\downarrow} x \overset{\phi}{\to} \psi \mathbin{\downarrow} x' \quad \psi \mathbin{\downarrow} \mathcal{S}_p (x) \mathbin{\Rightarrow} F \quad \psi \mathbin{\downarrow} x \overset{\phi}{\to} \psi \mathbin{\downarrow} x'
\end{align*}
\]

**TABLE 8. Operational semantics of global signal emission.**

\[
\begin{align*}
\mathcal{S}_p (\phi \mathbin{\downarrow} v x) & = \phi \mathbin{\downarrow} \mathcal{S}_p (x)
\end{align*}
\]
2.12 Root Signal Operator and Root Signal Deletion Operator.

We used the root signal operator $sp$ in the operational semantics. We can also add this operator to the theory with the axioms of table 6. The operator $sp$ determines the root signal of a process. If $sp(x) = T$ we say that $x$ has a trivial root signal, otherwise $x$ has a non-trivial root signal. Processes that were studied until now in the context of process algebra always have a trivial root signal. We can also define an operator $p_p$, that removes the root signal from its argument. It remains to be seen if this operator is useful. Notice that the equation $sp(x \wedge \phi) = sp(x)$ is derivable:

$$sp(x \wedge \phi) = sp(x) = sp(x \wedge \phi \cdot y) = sp(x \cdot (\phi \wedge \psi)) = sp(x).$$

Also $x = sp(x)^{\wedge} p_p(x)$ will now be derivable for finite closed process expressions. As a rewrite rule it is useless, however, because it will immediately introduce an infinite loop.

$$
p_p(\bot) = \bot \quad \text{RSD0} \quad p_p(x \cdot y) = p_p(x) \cdot y \quad \text{RSD3}
$$

An important operator in applications is the signal hiding operator $\Lambda$, that hides a propositional constant $P$. We give axioms based on the structure of basic terms in table 10, and provide semantics in table 11. Example: $P \Lambda (a \cdot (P \wedge b) + a \cdot (-P \wedge b)) = a \cdot (\Lambda T \wedge b) + a \cdot (T \wedge b) = a \cdot b$. If we assume the linear time law $a \cdot (x + y) = a \cdot x + a \cdot y$, this leads to the unwanted identity $a \cdot b = \delta$ (combine with the example in 2.3). Thus, this theory only exists in a branching time setting.

The global signal emission operator of 2.11, the root signal operator and root signal deletion operator of 2.12 and the signal hiding operator here can all be eliminated from closed terms, using the axioms given. Thus, we have the basic term lemma also for this extended signature. It is not difficult to establish that the extended theory is a conservative extension of BPAs, and that the axiomatisation is sound and complete for the term model modulo bisimulation (use the recipe of [BAV94]).

$$P \Lambda \bot = \bot \quad \text{SH0} \quad P \Lambda (\phi^{\wedge} \delta) = (\phi[T/P] \vee \phi[F/P])^{\wedge} \delta \quad \text{SH1}
$$

$$P \Lambda (x + y) = P \Lambda (sp(x+y)^{\wedge} x) + P \Lambda (sp(x+y)^{\wedge} y) \quad \text{SH2}
$$

$$P \Lambda (\phi^{\wedge} \psi :: a \cdot x) = (\phi[T/P] \vee \phi[F/P])^{\wedge} ((\phi \wedge \psi)[T/P] :: a \cdot (P \Lambda x) + (\phi \wedge \psi)[F/P] :: a \cdot (P \Lambda x)) \quad \text{SH3}
$$

$$P \Lambda (\phi^{\wedge} \psi :: a \cdot \neg x) = (\phi[T/P] \vee \phi[F/P])^{\wedge} ((\phi \wedge \psi)[T/P] :: a \cdot \neg (x[T/P] \vee x[F/P])) + (\phi \wedge \psi)[F/P] :: a \cdot \neg (x[T/P] \vee x[F/P])) \quad \text{SH4}
$$

TABLE 10. Signal hiding.
TABLE II. Operational semantics of signal hiding.

2.14 ITERATION.
We will not deal with full recursion in this paper. It is enough to consider linear recursion and iteration.

For iteration, we use the operator \( \ast \) (Binary Kleene Star) of [BEBP94]. We present axioms in table 12, operational semantics in table 13. We have one extra axiom for the Kleene star and terminal signal emission.

\[
x'(x \ast y) + y = x \ast y
\]

\[
x'(y \cdot z) = (x \ast y) \cdot z
\]


\[
x'(x \ast y) + y = x \ast y
\]

\[
x'(y \cdot z) = (x \ast y) \cdot z
\]

TABLE 13. Operational semantics of binary Kleene star.

2.15 LINEAR RECURSION.
A recursion equation is linear if it is of the form:

\[
X = \zeta \sum_{i=1}^{n} \phi_i \rightarrow a_i X_i + \sum_{j=1}^{m} \psi_j \rightarrow b_j \chi_j
\]

where \( \zeta, \phi_i, \psi_j, \chi_j \in B - \{ F \}, a_i \in A, b_j \in A_B \), and the \( X, X_i \) are recursion variables. The semantics for processes given by systems of such equations is easily given: for an equation as above we have for each summand a transition, of one the forms

\[
X_{i} \rightarrow a_i X_i
\]

\[
X_{i} \rightarrow b_j \chi_j
\]

Recursion equations in the following can simply be brought into this form.
3. PARALLEL COMPOSITION.
In this section, we extend the basic theory of section 2 with parallel composition. First, we consider parallel composition without synchronisation or communication, the so-called free merge.

3.1 PAPS.
The theory PAPs, Process Algebra with Propositional Signals, extends BPAps with operators $\parallel$, $\|$ and $\|$, and the axioms of tables 7 and 14. Note: $\parallel x = (F^{\wedge} a) \parallel x = F^{\wedge} (a \parallel x) = \perp$, and $a \parallel x = (a^{\wedge} T) \parallel x = a^{\parallel} (T^{\parallel} x)$, where the last expression can be proven equal to $a^{\parallel} x$ for all closed terms.

3.2 SIGNAL INSPECTION.
Now we have all the ingredients necessary to describe the inspection of an emitted signal. A very simple example will serve to make the point. Let us consider a traffic light. The set of propositional constants is {green, yellow, red}.

$TL(green) = (green \land \neg yellow \land \neg red) \land change\cdot TL(yellow)$
$TL(yellow) = (\neg green \land yellow \land \neg red) \land change\cdot TL(red)$
$TL(red) = (\neg green \land \neg yellow \land red) \land change\cdot TL(green)$.

Now we describe a careful car driver.

$CD = approach\cdot ((\neg green \Rightarrow stop) \cdot (green \Rightarrow start) \cdot drive) + (green \Rightarrow drive))$.

Expression $TL(x) \parallel CD$ now describes a correct interaction between light and driver.

3.3 ACPps.
The theory ACPps, Algebra of Communicating Processes with Propositional Signals, extends PAPs with operators $\parallel$, $\parallel$, $\|$, and replaces the axioms of table 14 by the axioms of the root signal operator and the axioms in table 15 below. We assume given a partial commutative and associative binary function on $A$, the communication function $\gamma$. We provide the semantics of ACPps in table 16. The semantics of PAPs can be extracted, by omitting all parts referring to the communication merge operator.
\[
a \mid b = \gamma(a, b) \quad \text{if defined} \quad \text{CF1} \quad a \mid b = \delta \quad \text{otherwise} \quad \text{CF2}
\]
\[
x \| y = x \| y + y \| x + y \| y \quad \text{CM1} \quad a \cdot \chi (b \setminus \phi) = (a \mid b) \cdot (\phi \setminus x) \quad \text{CM6TS}
\]
\[
(a \setminus \phi) \| x = a \cdot (\phi \setminus x) \quad \text{CM2TS} \quad a \cdot x \| b = (a \mid b) \cdot (x \| y) \quad \text{CM7TS}
\]
\[
a \cdot x \| y = a \cdot (x \| y) \quad \text{CM3} \quad (x + y) \| z = x \| z + y \| z \quad \text{CM8}
\]
\[
(x + y) \| z = x \| z + y \| z \quad \text{CM4} \quad x \| y + z = x \| y + z \quad \text{CM9}
\]
\[
(\phi \setminus x) \| y = \phi \setminus (x \| y) \quad \text{CMRS1} \quad (\phi \setminus x) \| y = \phi \setminus (x \| y) \quad \text{CMRS2}
\]
\[
(\phi : \rightarrow x) \| y = \phi : \rightarrow (x \| y) \quad \text{CMGC1} \quad x \| (\phi \setminus y) = \phi \setminus (x \| y) \quad \text{CMRS3}
\]
\[
(a \setminus \phi) \mid b \cdot x = (a \mid b) \cdot (\phi \setminus x) \quad \text{CMTS} \quad (\phi : \rightarrow x) \| y = \phi : \rightarrow (x \| y) + s_p(y) \setminus x \quad \text{CM6TS}
\]
\[
(a \setminus \phi) \mid b \cdot x = (a \mid b) \cdot (\phi \setminus x) \quad \text{CMTS} \quad (\phi : \rightarrow x) \| y = \phi : \rightarrow (x \| y) + s_p(y) \setminus x \quad \text{CM6TS}
\]
\[
\delta_H(a) = a \quad \text{if } a \not\in H \quad \text{D1} \quad \text{CMGC3}
\]
\[
\delta_H(a) = \delta \quad \text{if } a \in H \quad \text{D2} \quad \delta_H(\phi \setminus x) = \phi \setminus \delta_H(x) \quad \text{DRS}
\]
\[
\delta_H(x + y) = \delta_H(x) + \delta_H(y) \quad \text{D3} \quad \delta_H(x \setminus \phi) = \delta_H(x) \setminus \phi \quad \text{DTS}
\]
\[
\delta_H(x \mid y) = \delta_H(x) \cdot \delta_H(y) \quad \text{D4} \quad \delta_H(\phi : \rightarrow x) = \phi : \rightarrow \delta_H(x) \quad \text{DGC}
\]

**TABLE 15.** Merge with communication and encapsulation.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x^{\phi,a} \rightarrow x', s_p)</td>
<td>((x | y) = F, s_p, \ (x' | y) = F)</td>
</tr>
<tr>
<td>(x | y^{\phi,a} \rightarrow x' | y)</td>
<td>(y | x')</td>
</tr>
<tr>
<td>(x^{\phi,a} \rightarrow x', s_p)</td>
<td>((x' | y) = F)</td>
</tr>
<tr>
<td>(x | y^{\phi,a} \rightarrow x' | y)</td>
<td>(y | x')</td>
</tr>
<tr>
<td>(x^{\phi,a} \rightarrow x', \psi, s_p)</td>
<td>((y | x') = F, s_p, \ (y | x') = F)</td>
</tr>
<tr>
<td>(x | y^{\phi,a} \rightarrow \psi )</td>
<td>(y, y | x, s_p, \rightarrow \psi, y)</td>
</tr>
</tbody>
</table>

**TABLE 16.** Semantics of ACPps.

\[
\delta_H(a) = a \quad \text{if } a \not\in H \\
\delta_H(a) = \delta \quad \text{if } a \in H \\
\delta_H(x + y) = \delta_H(x) + \delta_H(y) \\
\delta_H(x \mid y) = \delta_H(x) \cdot \delta_H(y)
\]

\[
s_p(x \| y) = s_p(x) \wedge s_p(y)
\]

\[
x \| y^{\phi,a} \rightarrow \chi, s_p, \ (\chi \| x \) = F, a \mid b = c\]

\[
s_p(x \| y) = s_p(x)
\]

\[
x^{\phi,a} \rightarrow x', y, s_p, \ (x' \| y) = F, s_p, \ (x' \| y') = F, a \mid b = c\]

\[
s_p(x \| y) = s_p(x) \wedge s_p(y)
\]

\[
x^{\phi,a} \rightarrow x', y, \psi, s_p, \ (x' \| y) = F, x \wedge s_p, \ (y') = F, a \mid b = c\]

\[
s_p(x \| y) = s_p(x)
\]

\[
x^{\phi,a} \rightarrow y', s_p, \ (x' \| y) = F, s_p, \ (x' \| y') = F, a \mid b = c\]

\[
s_p(x \| y) = s_p(x) \wedge s_p(y)
\]

\[
x^{\phi,a} \rightarrow y', s_p, \ (x' \| y) = F, x \wedge s_p, \ (y') = F, a \mid b = c\]

\[
s_p(x \| y) = s_p(x) \wedge s_p(y)
\]

\[
x^{\phi,a} \rightarrow y', s_p, \ (x' \| y) = F, x \wedge s_p, \ (y') = F, a \mid b = c\]

\[
s_p(x \| y) = s_p(x) \wedge s_p(y)
\]

\[
s_p(a) = a \quad \text{if } a \not\in H \\
\delta_H(x)^{\phi,a} = \delta_H(x)'\]

\[
s_p(\delta_H(x)) = s_p(x)
\]
4. STATE OPERATOR.
In this section, we extend any of the theories BPAps, PApS, ACPps with the state operator of [BAB88]. The interesting aspect here is, that we allow the state to be (partly) visible to the process, i.e. a state can emit a signal.

4.1. SYNTAX AND SEMANTICS.
Let us assume that a state operator in the sense of [BAB88] is given by a domain \( S \) and functions \( \text{act}: A \times S \rightarrow \mathbb{A} \cup \{\delta\} \) and \( \text{eff}: A \times S \rightarrow S \). The expression \( \lambda_s(x) \) with \( s \in S \) denotes process \( x \) working on the state space \( S \) with the current state being \( s \in S \).

We can assume that there is an additional function \( \text{sig}: S \rightarrow B \) which determines for each state the signal that is emitted by that state. The absence of signals is modeled by taking \( \text{sig}(s) = T \) of course.

Now the eight equations for the state operator are as shown in table 17, the operational semantics is given in table 18.

Using a state operator that generates signals one can define signaling processes in such a way that the equations need not contain any signal at all, thus considerably optimizing the notation. We will illustrate this in a simple example.

\[
\begin{align*}
\lambda_s(\perp) & = \perp \quad \text{SOS0} & \lambda_s(x + y) & = \lambda_s(x) + \lambda_s(y) \quad \text{SOS4} \\
\lambda_s(\delta) & = \text{sig}(s) \wedge \delta \quad \text{SOS1} & \lambda_s(\phi \wedge x) & = \phi \wedge \lambda_s(x) \quad \text{SOS5} \\
\lambda_s(\text{act}(a, s)) & = \text{sig}(\text{eff}(a, s)) \quad \text{SOS2} & \lambda_s(x \wedge \phi) & = \lambda_s(x) \wedge \phi \quad \text{SOS6} \\
\lambda_s(\text{act}(a, s) \wedge \lambda_s(\text{act}(a, s))) & = \text{sig}(\text{eff}(a, s)) \quad \text{SOS3} & \lambda_s(\phi \rightarrow x) & = \text{sig}(s) \wedge \phi : \rightarrow \lambda_s(x) \quad \text{SOS7}
\end{align*}
\]

TABLE 17. State operator generating signals.

\[
\begin{align*}
x^{\phi, a} & \rightarrow x', \text{sig}(s) \wedge s \quad \text{SOS0} & (x'') & = F, \text{sig}(\text{eff}(a, s)) \wedge s \quad (x'') & = F, \text{act}(a, s) = b \wedge \delta \\
\lambda_s(x) & \phi \perp \lambda_s(\text{act}(a, s)) \quad (x')
\end{align*}
\]

\[
\begin{align*}
x^{\phi, a} & \rightarrow \psi, \text{sig}(s) \wedge s \quad \text{SOS0} & (x'') & = F, \text{sig}(\text{eff}(a, s)) \wedge \psi \wedge F, \text{act}(a, s) = b \wedge \delta \\
\lambda_s(x) & \phi \perp \text{sig}(\text{eff}(a, s)) \wedge \psi
\end{align*}
\]

TABLE 18. Operational semantics.

4.2 EXAMPLE.
Let \( D \) be a finite alphabet of data, and let \( D^* \) be the collection of finite sequences over \( D \). The empty sequence is denoted by \( \varepsilon \) and adding an element \( d \) to the list \( x \) results in \( x^d \). The propositional constants are as follows: \( \text{on}\_\text{top}(d) \) for \( d \in D \), and \( \text{empty} \).

We will assume that these signals are exclusive, i.e. we will assume that the following formula always holds: \( \Phi = (\text{empty} \supset \land d \in D \text{ on}\_\text{top}(d)) \land \land d \in D (\text{on}\_\text{top}(d) \supset \land \land d \in D \text{ on}\_\text{top}(e)) \).

\( D^* \) will be the state space for a process that represents a stack over \( D \). The signal function \( \text{sig} \) is defined by \( \text{sig}(\varepsilon) = \text{empty}, \text{sig}(\varepsilon d) = \text{top}(d) \). The atomic actions are:
push_int(d), push(d) for \( d \in D \) (the suffix int denotes an intended action),
pop_int, pop.

The functions \( \text{act} \) and \( \text{eff} \) are given by:
\[
\begin{align*}
\text{act}(\text{push_int}(d), a) &= \text{push}(d) \\
\text{act}(\text{pop_int}, a) &= \text{pop}, \\
\text{eff}(\text{push_int}(d), a) &= ad \\
\text{eff}(\text{pop_int}, e) &= e, \\
\text{eff}(\text{pop_int}, od) &= \sigma.
\end{align*}
\]

(For \( \text{act} \) only those cases are given where \( \text{act} \) will not lead to \( \delta \).) The behavior of a stack over \( D \) is given by the following process definition.
\( \text{stack}(D) = \Phi \xrightarrow{\delta} \lambda e((\sum_{d \in D} \text{push_int}(d) + \text{pop_int})*\delta). \)

5. ABSTRACTION.
We provide axioms for silent step and abstraction in the setting of branching bisimulation of [GLW89].

5.1 ACP\(^{PS} \).
The theory \( ACP^{PS} \) extends \( ACP^{PS} \) by the addition of a special constant \( \tau \notin A \), the silent step, and a unary operator \( \tau \) for each \( I \subseteq A \), the abstraction operator. As axioms we have all axioms of \( ACP^{PS} \), with now \( a, b \in A \cup \{ 0, \tau \} \), minus the additional axioms of table 19 below.

\[
\begin{align*}
\tau \cdot (s_p(y) \xrightarrow{\phi} (\tau \cdot (y + z) + z)) &= \tau \cdot (y + z) & \text{BS} \\
\tau_1(x \cdot y) &= \tau_1(x) \cdot \tau_1(y) & \text{TI4} \\
\tau_1(a) &= a & \text{TI1} \\
\tau_1(\phi \xrightarrow{\phi} x) &= \phi \xrightarrow{\phi} \tau_1(x) & \text{TIRS} \\
\tau_1(\phi) &= \tau & \text{TI2} \\
\tau_1(x \cdot \neg \phi) &= \tau_1(x) \cdot \neg \phi & \text{TITS} \\
\tau_1(x + y) &= \tau_1(x) + \tau_1(y) & \text{TI3} \\
\end{align*}
\]

TABLE 19. Silent step and abstraction.

5.2 SEMANTICS.
The operational semantics now also has arrow labels of the form \( \phi, \tau \). In the previous rules, we now have \( a \in A \cup \{ \tau \} \). The additional rules for the abstraction operator are shown in table 20.

With this comes a new definition of bisimulation. In the following, \( x, y, x', y', \ldots \) range over terms and \( \phi, \psi, \ldots \) range over propositions.

A relation \( R \) on process expressions is a branching bisimulation when the following holds:
\[
\begin{align*}
i. & \quad \text{if } xRy \text{ then } s_p(x) = s_p(y) \\
ii. & \quad \text{if } xRy \text{ and } x \xrightarrow{\phi, \tau} x', \text{ then either:} \\
& \quad a. \quad a = \tau, \quad s_p(x) \supset \phi = T \text{ and } x'Rx', \text{ or:} \\
& \quad b. \quad \text{for all valuations } v \text{ such that } v(s_p(x) \land \phi) = T, \text{ there are propositions } \psi_1, \ldots, \psi_n \quad (n \geq 0) \text{ and } \psi', y_1, \ldots, y_n \text{ such that } s_p(x) \supset \psi_i = T \text{ for all } i, v(\psi) = T, \quad y \xrightarrow{\psi_1} y_1 \ldots \psi_n y_n \quad \psi \xrightarrow{\phi} y' \text{ and } xRy_i \text{ for all } i \text{ and } xRy' 
\end{align*}
\]

224
iii. if $xRy$ and $x \not\equiv y$, then for all valuations $v$ such that $v(s_p(x) \land \phi) = T$, there are propositions $\chi, \psi, \psi_1, \ldots, \psi_n$ ($n \geq 0$) and expressions $y_1, \ldots, y_n$ such that $\xi = \chi, s_p(x) \supset \psi_i = T$ for all $i$, $v(\psi) = T$, $y \equiv^v y_1 \ldots \equiv^v y_n$, $\psi \equiv^v \chi$ and $xRy_i$ for all $i$.

iv, v. like ii, iii with the roles of $x$ and $y$ interchanged.

We say a branching bisimulation $R$ satisfies the root condition for $x$ and $y$ if:

vi. if $x \not\equiv y'$, then for all valuations $v$ such that $v(s_p(x) \land \phi) = T$, there is a proposition $\psi$ and a term $y'$ such that $v(\psi) = T$, $y \equiv^v y'$ and $xRy'$.

vii. if $x \not\equiv y$, then for all valuations $v$ such that $v(s_p(x) \land \phi) = T$, there are propositions $\psi, \chi$ such that $v(\psi) = T$, $y \not\equiv^v \chi$ and $\xi = \chi$.

viii, ix. like vi, vii with the roles of $x$ and $y$ interchanged.

We call two expressions $x, y$ branching bisimilar, notated $x \equiv_b y$, if there is a branching bisimulation relating $x$ and $y$. Two expressions $x, y$ are rooted branching bisimilar, $x \equiv_{rb} y$, if there is a branching bisimulation that satisfies the root condition for $x$ and $y$.

\[
\frac{x \not\equiv \phi, a \in I}{\tau_1(x) \not\equiv \tau_1(x')} \quad \frac{x \not\equiv \phi, a \in I}{\tau_1(x) \not\equiv \psi} \quad \frac{x \not\equiv \phi, a \in I}{\tau_1(x) \not\equiv \tau_1(x')} \quad \frac{x \not\equiv \phi, a \in I}{\tau_1(x) \not\equiv \psi}
\]

\[sp(\tau_1(x)) = sp(x)\]

**TABLE 20. Semantics of ACP^ps.**

5.3 **THEOREM.** For all closed ACP^ps terms $t, s$ we have

\[ACP^ps \vdash t = s \iff t \equiv_{rb} s,\]

i.e. ACP^ps is a sound and complete axiomatisation of the bisimulation model.

6. **EXAMPLE: COMMUNICATING BUFFERS.**

In this example we study a system where both signal inspection and communication play a role. We will show that communication can be replaced by inspection. We start out from a standard specification of one element buffers, that in addition always signal the contents on the output port (all specifications can be simply brought into a form that uses iteration rather than recursion). The buffer $B_{ij}$ has input port $i$ and output port $j$, and can buffer messages from some finite set $D$. Let $\emptyset \not\in D$. The signal $show_j(d)$ means that message $d$ is offered at port $j$ ($d \in D$), $show_j(\emptyset)$ means that the buffer is empty.

\[B_{ij} = show_j(\emptyset)^\land \sum_{d \in D} read_i(d) \cdot B^y_{ij} \cdot d\]

\[B^y_{ij} = show_j(d)^\land \sum_{d \in D} send_j(d) \cdot B^y_{ij} \cdot d\]

\[X = \delta_H(B^{12} \upharpoonright B^{23})\]

where $send_2(d) \mid read_2(d) = comm_2(d)$ (communication gives $\delta$ otherwise),
and $H = \{\text{read}_2(d), \text{send}_2(d) : d \in D\}$.

The system $X$ obeys the usual specification of two coupled one-element buffers (as in [BAW90], page 106), after hiding all signals. As a first step in replacing communication by inspection, we omit the parametrisation of the communication action in favor of signal inspection. To make this correct, we need to require that signals are exclusive, formalised by proposition

$$\Phi_i = (\text{show}_i(0) \supset \mathcal{M}_{d \in D} \text{show}_i(d)) \land$$

$$\mathcal{M}_{d \in D} (\text{show}_i(d) \supset \neg \text{show}_i(0)) \land \mathcal{M}_{d \in D} \text{show}_i(e)).$$

$$C_i^d = \text{show}_i(0)^\diamond \sum_{d \in D} \text{show}_i(d) \rightarrow \text{read}_i C_d^i$$

$$C_d^i = \text{show}_i(d)^\diamond \text{send}_i C_i.$$

$$Y = (\Phi_1 \land \Phi_2 \land \Phi_3) \supset \partial_{H}(C_1 \parallel C_2),$$

where communication is given by $\text{send}_2 \perp \text{read}_2 = \text{comm}_2$.

After abstracting from actions and signals at port 2, we obtain the same system as before.

Next, we can do away with the synchronisation in favour of two extra signals at the connecting port. First, we consider the specification without extra actions:

$$E_{ij} = (\text{show}_j(0) \land \neg \text{flag}_i) \land \sum_{d \in D} (\text{show}_i(d) \land \neg \text{flag}_j) \rightarrow \text{read}_i E_d^i$$

$$E_d^i = (\text{show}_j(d) \land \text{flag}_i) \land (\text{show}_i(0) \land \neg \text{flag}_j) \rightarrow \text{send}_i E_d^j.$$

$$W = (\Phi_1 \land \Phi_2 \land \Phi_3) \supset E_1 \parallel E_2 \parallel E_3,$$

where this is the free merge, i.e. this is a specification in PAs.

Unfortunately, this system does not behave as a two-item buffer but as a one-item buffer. If we want the intended behaviour, we have to put in extra actions:

$$F_{ij} = (\text{read}_1 \land \text{show}_j(0) \land \neg \text{flag}_j) \land \sum_{d \in D} (\text{flag}_i \land \text{show}_i(d)) \rightarrow \text{read}_i F_d^i$$

$$F_d^i = (\neg \text{read}_1 \land \text{show}_j(d) \land \neg \text{flag}_j) \land (\text{read}_1 \land \neg \text{flag}_1) \rightarrow \text{send}_1 G_d^i,$$

$$G_d^i = (\neg \text{read}_1 \land \text{show}_j(d) \land \text{flag}_j) \land (\neg \text{read}_1 \land \neg \text{reset}_1) \rightarrow \text{reset}_1 F_d^i.$$

$$V = (\Phi_1 \land \Phi_2 \land \Phi_3) \supset F_1 \parallel F_2 \parallel F_3,$$

where this is the free merge, i.e. this is a specification in PAs. Some calculations show that this system shows the required behaviour, after hiding all signals at port 2 and the additional signals introduced, and abstracting from the action set $I = \{\text{read}_2, \text{send}_2, \text{comm}_2, \text{reset}_2, \text{reset}_3\}$.

7. CONCLUSION.

We conclude that we have described the interplay between the execution of actions of a process (giving the state changes, the dynamics of a process) and the propositions that hold in a state of a process (giving the static part of a process). The signal emitted by a state is a proposition that constitutes the
visible part of this state, and an action leading out of a state can be conditional on a proposition that
should hold in a state. In a parallel composition of two processes, an action executed by one process
can be conditional, depending on the signal emitted by the other process. This described a mechanism
called signal inspection or signal observation.

We have given some small examples. Further work would be to construct larger examples, and to
extend both logic and process theory, for instance with timing constructs.

REFERENCES.

[AUB84] D. AUSTRY & G. BOUDOL, Algebre de processus et synchronisation, Theoretical Computer

[BAB88] J.C.M. BAETEN & J.A. BERGSTRA, Global renaming operators in concrete process algebra,

[BAB92] J.C.M. BAETEN & J.A. BERGSTRA, Process algebra with signals and conditions, in:
Programming and mathematical method, Proc. Summer School, Marktoberdorf 1990 (M. Broy, ed.),

[BAV93] J.C.M. BAETEN & C. VERHOEF, A congruence theorem for structured operational semantics
477-492.

in computer science (S. Abramsky, D.M. Gabbay & T.S.E. Maibaum, eds.), Vol. IV, Syntactical
Methods, Chapter 2, pp. 149 - 268.


[BeBP94] J.A. BERGSTRA, I. BETHKE & A. PONSE, Process algebra with iteration and nesting, The

[BeK84] J.A. BERGSTRA & J.W. KLOP, Process algebra for synchronous communication,
Information & Control 60, 1984, pp. 109-137.

CSLI.


[GLW89] R.J. VAN GLABBEK & W.P. WEIJLAND, Branching time and abstraction in bisimulation
semantics. Extended abstract, in: Information Processing 89, IFIP World Congress, San Francisco

[GRP94] J.F. GROOTE & A. PONSE, Process algebra with guards (combining Hoare logic with

[NIS94] X. NICOLLIN & J. SIFAKIS, The algebra of timed processes ATP: theory and application,
A Simulation of a CIM Factory using a Parallel Object Oriented Language

André van Delft, Leen Stolk
Delftware Technology BV
Schijfstraan 17, 2625 KG Delft
E-mail: DELFT@dataweb.nl

Abstract

This paper is about a simulation program of a Computer Integrated Manufacturing plant, that involves dozens of conveyor belts, machines, controllers and a sophisticated communication protocol. The program has evolved from a specification written in the Process Algebra-based specification language PSF. It was actually implemented in Scriptic-C++, a parallel object-oriented programming language, also with roots in Process Algebra. The simulation has a graphical user-interface with fluent animation, and runs efficiently on current PCs. These characteristics were essential in the detection of a inherent deadlock caused by a non-trivial design error in the original specification.

Introduction

Computer Integrated Manufacturing is an emerging area that greatly enlarges the degrees of freedom in factory design and operation. The behaviour and efficiency of CIM installations cannot usually be expressed in simple terms of the constituing components. Their dependencies cause nonlinear relationships between component capacity and overall throughput. Physical deadlocks may occur when product streams mutually block one another. Logic deadlocks are possible when the controlling processes encounters unforeseen states.

CIM installations therefore require advanced techniques for analyzing and optimizing the behaviour, such as Process Algebra [BK84]. Several tools for working with Process Algebra exist, e.g., a simulation engine for the related formalism PSF [MV93]. However, that simulator falls short when applied to CIM systems of a realistic size, because the low speed and the rudimentary user interface.

This paper describes a project where the programming language Scriptic-C++ was used as an alternative tool to build a simulation of a CIM installation. This language may be regarded as an extension to the object-oriented language C++ with parallel constructs from Process Algebra. The project aimed to apply Scriptic-C++ for simulating a relatively large system, and to evaluate it as a tool for Process Algebra based research. During the project an already available specification of a flexible manufacturing plant was remapped from PSF into a Scriptic-C++ simulation program. The result runs an order of magnitude faster than the PSF based...
simulation. A graphical user interface lets a user control the simulation, and an animation shows at a glance the state of the simulated plant. These characteristics greatly help to understand and improve the mechanisms of the plant.

The next sections of this paper discuss Scriptic-C++, the simulated CIM installation and its specifications, the user interface of the simulation program, the results, and conclusions. The reader is assumed to be familiar with Process Algebra, PSF, and object-oriented programming.

**Scriptic-C++**

Scriptic-C++ is an extension to the object-oriented programming language C++, with contructs derived from Process Algebra, like PSF extends the Algebraic Specification Formalism (ASF) [D91, D92, GDW93]. Scriptic-C++ and PSF have more in common. Both have an ASCII-based lexical structure, like normal programming languages have, and unlike the average Process Algebra texts with the more special character fonts and styles. This makes it easy to edit specifications in PSF and Scriptic-C++, and tools processing these specifications can be simpler. Both PSF and Scriptic-C++ inherit data structuring and modularisation constructs from their ancestors (c.q. C++ and ASF). Both PSF and Scriptic-C++ support process expressions and refinements, named processes in PSF and scripts in Scriptic.

However, there are also major differences, because PSF aims to be a formalism for processes, whereas Scriptic-C++ aims to be a generally applicable programming language with object-oriented and process-oriented features. For PSF with a number of tools are available, such as a simulator and a proof assistant. See [V93] for a discussion and an overview.

Scriptic-C++ comes with typical programming utilities: a preprocessor which translates programs into C++, and a run-time system which should be linked with a Scriptic-C++ executable. The run-time system evaluates process expressions at run time and executes pieces of C++ code that stand for atomic actions. Scriptic has earlier been defined as an extension to the programming languages Pascal, Modula-2 and C. The C++ variant is more suitable for simulations because of its object-orientedness.

As a consequence of the different aims, Scriptic-C++ and PSF differ in the following details:

**Atomic Actions**

In PSF, atomic actions just need to be declared as such. A simulator may print their names, for instance in an action trace to show that they have happened. In Scriptic-C++, an atomic action is a C++ code fragment, visually enclosed by braces. It is not identified by a name. To get a working Scriptic-C++ program just showing action traces would require more work than a comparable PSF specification with the simulator. On the other hand, having code fragments as atomic actions makes Scriptic-C++ a generally applicable programming language.

Example: the following code fragment prints a message on the screen, using the C++ stream library with the standard output device cout and the output operator «:

```cpp
{cout << "A happens"}
```

If a code fragment is executed then this does not always imply that an atomic action has been executed conceptually: the code fragment may or may not fail, based on a boolean condition. The following code fragment will only succeed if the variable i is zero:

```cpp
{? i==0}
```
Choice, sequence, parallelism operator symbols

Scriptic applies different operator symbols for choice, sequence and parallelism than Process Algebra and PSF do, since it applies conventions from the C language and related tools. Therefore, sequence is denoted by a semicolon (;), as in C statement sequences, choice by a bar (|), like the bitwise or-operator in C and like the choice operator in Yacc, and parallelism by a comma, like in C declarations. Also, the comma binds stronger than the semicolon. It is well possible to make a Scriptic-C++ dialect that conforms more to operator symbols and precedences in Process Algebra.

And- and or-parallelism

Scriptic-C++ offers and-parallelism and or-parallelism. And-parallelism is like normal parallelism, except that all operands of the and-parallel operator (&) are terminated as soon as one of the operands enters an explicit deadlock situation. All operands of the or-parallel operator (+) are terminated as soon as one of the operands ends successfully.

Refinements

Scriptic-C++ process refinements are called scripts. These may have parameters, which may also return values, as in C reference parameters. These output parameters have a question mark suffix. This may be replaced by an exclamation mark, to force the return parameter to a specified value: if the actual value does not match, then an explicit deadlock is encountered so that the script call fails. Examples of four such calls:

A; B(0); C(i?); D(1!)

Communication

In Process Algebra and PSF a pair of atoms may be defined as communication, yielding another atom. Hiding the original two atoms is normally needed to prevent them from happening as normal actions. In Scriptic a pair of scripts may be defined together as a general script expression, which is more general than an atomic action. Hiding is not applicable as the first two scripts cannot happen in other contexts than communication. As an example, the following pair of communicating scripts can transfer an integer from one participant to the other:

P(int j), Q(int k?) = {k=j}; {cout<<"done"}

Communicating scripts may also be specified as arrays of a given size. The following specifies pairs of communicating scripts R[0],S[0] up to R[11],S[11]; each of these can transfer an integer:

R[12 i](int j), S[](int k?) = {k=j}; {cout<<"done"}

Scriptic-C++ also offers communication through channels, which require senders and receivers. A channel needs only a single identifier, instead of the two for shared scripts. Moreover, parameter transfer does not need to be programmed explicitly. Example:

aChannel<->(int i?) = {cout<<"aChannel("<<i<<")")

Here communication may actually occur if for instance a sender process calls aChannel<-(2) and a parallel receiver process calls aChannel->(i?). Channel arrays are also supported.
Control

Like PSF, Scriptic-C++ offers control through guards. The guarding expression is a boolean expression in C++ starting with a question mark, placed between braces, followed by a colon. Alternatively one can apply C-like if-else and switch statements. Three equivalent examples:

```
{? i==0}: A | {? i!=0}: B
if (i==0) A else B
switch(i) (case 0: A break; default: B break;)
```

There is a difference between a guarded expression and an if-expression without else-clause:

```
{? i==0}: A
if (i==0) A
```

The guarded expression becomes deadlock if the guard evaluates to false. The if-expression becomes a neutral process if the test evaluates to false. This neutral process depends on the context: it equals deadlock in the context of a choice or of or-parallelism, but it behaves like the empty process in the context of a sequence, or of normal parallelism, or of and-parallelism.

Iterations

Like PSF, Scriptic-C++ allows for recursion as a way to specify iterations, but recursion may consume considerable amounts of memory, and often it is less clear than explicit iterations. PSF offers a Kleene star (*) operator for sequential iterations. In contrast, Scriptic-C++ offers four special operands for iterations. These operands may be combined with any operator, so that not only sequential iterations are possible, but also alternative and parallel ones. Moreover, these operands may be refined in scripts.

Scriptic offers two C-like iteration constructs: for and while. Unlike in C, while and for now act as operands, so that they require an operator symbol next to them. There are two more basic iterators in Scriptic: the exit and the optional exit., written as .. and [...]. Activating the exit iterator terminates the iteration. Hence, while(exp) is equivalent to if (!exp).. (the exclamation mark is the logical negation operator in C++). Activating the optional exit makes it possible, but not mandatory, to terminate it (as if it is a choice between the exit and the neutral process).

The pseudo-variable pass keeps track of the pass count in the iteration, and counts from 0 on.

Examples:

```
script1 = while (isOk); B
EXIT = ..
SOME = [...]
TIMES(int i) = while (pass<i)
chooseOto9 = TIMES (10) | choose(pass)
doAllParallel = for (p=list; p; p=p->next), doit(p)
parseSentence = (SOME; parseWord); parseSentenceDelimiter
```

The last example is, apart from memory usage, equivalent to:

```
parsesentence = parseWord; parseSentence | parseSentenceDelimiter
```
Specification units

The main specification unit in Scriptic-C++ is the C++ class construct. A class can contain variables, functions and also scripts, like PSF modules can contain processes. Class items can either be specific for each object (i.e. instance of a class), or shared by all instances. The latter items are called static. They are more or less comparable to the items in PSF modules, since these modules do not have a notion of instantiation.

C++ has a multiple inheritance mechanism for classes. Scripts can be inherited like functions. This implies that, once defined in a class, scripts can have different implementations in subclasses.

State recording

Scriptic-C++ objects, functions and scripts record their state in instance variables. As PSF does not offer such variables, specifications tend to have longer parameter lists to functions and processes, and more function calls as parameters. This makes them less readable.

* Often one needs to express that a value is read from some source and then processed. PSF requires an explicit sum over all possible values that can be read, as in

\[ \text{aProcess} = \sum (i \text{ in } \text{aSetOfNumbers}, \text{read}(i).\text{process}(i)) \]

It is not possible to summarize in PSF over infinite data types such as integers. Scriptic-C++ offers an alternative using output parameters and variables that are local to the script, as in:

\[ \text{aScript} = \text{int } i; \text{read}(i?); \text{processDatum}(i) \]

If only elements out of a limited set of integers should be accepted, such as from 0 to 9, then one could try:

\[ \text{aScript} = \text{int } i; \text{for}(i=0; i<10; i++) | \text{read}(i?); \text{processDatum}(i) \]

However, this is erroneous. When activating the 10 alternatives, the script read is called with parameter values 0 to 9. Thereafter the variable i equals 10. After one of the 10 reads has succeeded, processDatum gets called with the unintentional parameter 10. This problem can be circumvented using the special loop counter pass:

\[ \text{aScript} = \text{int } i; \text{TIMES}(10) | \text{read}(\text{pass}?); \text{processDatum}(|\text{pass}) \]

Notion of time

PSF has no built-in notion of time. In Scriptic-C++, an atomic action, or code fragment, has an associated time duration. A subsequent action can only happen if the specified amount of time has elapsed. Here time does not denote real time but simulation time. Simulation time only elapses as far as this would be needed to get any atomic action to take place.

Example: the following code fragment would do nothing, except for lasting 1 simulation time unit:

\[ (\text{duration}=1) \]
This notion of time has little in common with discrete time process algebra. It is a closer to real time process algebra, since time is treated as a numeric entity. Real time process algebra concentrates on starting times of atomic actions, which end at the same moment as they start. Scriptic concentrates on durations of atomic actions. Once a Scriptic-C++ atomic action with a positive duration has started, it may even be terminated prematurely by another process (e.g. in an and-parallel or or-parallel context). It is comparable to a summation in real time process algebra over the time domain of sequences of two actions, with starting times that differ by the duration.

**Activation and deactivation**

In Scriptic-C++ activating a process may start with the execution of *activation code*: a special C++ code fragment which may make some initializations for the rest of the process. Likewise, deactivating a process (which completes or which is eliminated by an alternative), may cause the execution of *deactivation code*. Like atomic actions, activation code and deactivation code are specified within braces, but they come also with a trailing `<` symbol c.q. a preceding `>` symbol. Unlike atomic actions, activation codes and deactivation codes in alternative processes do not exclude each others.

Example: activating the following process starts with a message. Executing its code fragment yields another message, as does its deactivation:

A = {cout<<"A activated"} < {cout<<"A happens"} > {cout<<"A deactivated"}

PSF does not offer activation code and deactivation code. However, when the PSF simulator activates an atomic action, it may put that name into a list to present to the user. When deactivated, the name is removed from the list.

**Summary**

<table>
<thead>
<tr>
<th>Issue</th>
<th>PSF</th>
<th>Scriptic-C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic actions</td>
<td>Declared as such C++ code fragments</td>
<td>C++ code fragments</td>
</tr>
<tr>
<td>Choice, sequence, parallelism operator symbols</td>
<td>+ . ++</td>
<td>l ; . ,</td>
</tr>
<tr>
<td>And- and or-parallelism</td>
<td>Absent</td>
<td>+ &amp;</td>
</tr>
<tr>
<td>Refinements</td>
<td>Processes</td>
<td>Scripts</td>
</tr>
<tr>
<td>Communication</td>
<td>Shared atoms; hiding</td>
<td>Shared scripts, channels; no hiding</td>
</tr>
<tr>
<td>Control</td>
<td>Guards</td>
<td>Guards, if, switch</td>
</tr>
<tr>
<td>Iterations</td>
<td>recursion, Kleene star (*)</td>
<td>recursion, for, while, (optional) exit</td>
</tr>
<tr>
<td>Specification unit</td>
<td>module</td>
<td>class</td>
</tr>
<tr>
<td>State recording</td>
<td>using process choice (sum) over data and using parameters</td>
<td>variables</td>
</tr>
<tr>
<td>Notion of time</td>
<td>Absent</td>
<td>Atomic actions have a duration</td>
</tr>
<tr>
<td>Activation and deactivation</td>
<td>Absent</td>
<td>C++ code fragments</td>
</tr>
</tbody>
</table>

**The CIM Installation**

The simulated CIM installation is a model factory at the Cooperative Engineering Center (CEC) research department of Digital Equipment in Nieuwegein, The Netherlands. This model factory is in its turn a research tool to understand CIM aspects and to try-out new ideas for a real factory that produces electronic printed circuit boards, or PCBs. The model factory has dozens of parts such as machines and conveyor belts. The following is a summary from the descriptions in [HP92] and [BP93], with some simplifications:
Three kinds of PCBs are manufactured: product types A, B and C. These types differ on the kind of board and on the number and kind of components used. During manufacturing, one, two or three PCBs of the same type are operated on at the same time. Such a cluster is called a batch. The PCBs in a batch stay (conceptually) together during the entire production process.

Figure 1 shows the tracks of the batches. On the left are two raw material stores (RMSs). One contains empty boards for product type A, the other for types B and C. Board movers put these boards on the conveyor belt, after which these are treated as batches. A batch passes first through a screen printing station (SCP), which puts solder paste on the boards. Thereafter component placement stations CP1 and CP2 place components on the PCBs. A batch of type A passes both CP1 and CP2, type B passes CP1 only, and type C passes CP2 twice, so doing the tour twice. Board movers (BoMs, single vertical arrows) move batches to and from CP1 and CP2; batch movers (BaMs, double vertical arrows) move batches. A batch also passes the reflow clean station (RC). Then it enters an in-process store (IPS), to be treated as separate PCBs. For each product type A, B or C there is one IPS, which can store up to three PCBs. The PCBs stay there until a product order of the same type arrives. Then the requested number of PCBs of the product type is transported from the IPS as a new batch. It passes through the test repair station (TR) for a visual and electrical test. If the production of the batch is completed, it is sent to the warehouse. A C-batch that is only half way proceeds to the second side buffer (SSB) to go to CP2 for the second time.

There are several possible ways to organize the control of this factory, and several ways have been treated in different PSF specifications. [BP93] presents a centralized control process, while [HP92] applies 8 control zones (as shown in figure 1), that effectuate a decentralized control. The control zone approach is discussed in more detail since it was rebuilt into the Scriptic-C++ simulation. Each control zone is dominated by a controller. Control zone 2 is the most simple one:
The numbered connections between the parts of the model factory are called ports. Batch ports communicate batches between manufacturing units and transporting units. Data ports are adjacent to controllers, and communicate data. A request port is a special kind of data port, to communicate requests between controllers.

Manufacturing units and transporting units behave in general as follows: they send an available message to the controller (over a data port), and receive a go command back. Then a batch arrives over a batch port from another unit. The unit processes the batch and then sends a holding message to the controller, with the batch identification as parameter so that the controller can decide on the next processing step. Finally the batch is sent out to the controller, and the unit returns to its initial state. In Process Algebra notation:

\[
\text{send(available) \cdot read(go) \cdot read(batch) \cdot action \cdot send(hold, batch.id) \cdot send(batch)}
\]

A controller reads the hold message and determines to what unit it will direct the batch. It reads the available message from that next unit and returns a go message, so that that unit will take the batch over:

\[
\text{read(hold, batch.id) \cdot read(available) \cdot send(go)}
\]

A controller has one such a subprocess running in parallel for each of the manufacturing units and transporting units in its zone. Another subprocess of a controller is to pass product request messages through to the controller with the next lower number. There are 3 exceptions:

- at one end is the warehouse controller, where product orders from the environment come in. Each order is converted to requests, each being for at most 3 products.

- in the middle is the IPS controller. The in-process stores are assumed to be filled, so initially the IPS controller sends three requests to fill the store. Later it will send requests if products go out of the store. Incoming requests do not result in propagation to the next control zone, but they result in taking out the requested number of products from the proper store, for transport to the test repair station. An IPS tries to sent the number of products it contains in a present message, but the controller only reads this message if this is at least the requested number of products. If not enough products are in the store yet then more
products will have been requested by the IPS controller. When these products arrive, they enter the IPS and the proper number of products leaves right away.

- at the other end is the second-side controller, with a more complicated behaviour to prevent a deadlock by waiting C-products at SSB [VW92].

**Specifications**

The PSF specification of the model factory has about 50 modules. There are data modules describing batches, products, messages, ports and others, and process modules for the whole model factory, the control zones, the controllers, stations, and transport units.

Most of these modules have counterparts in classes in the Scriptic-C++ simulation program. However, the latter applies the C++ inheritance mechanism. Each control zone and each of its items (belts, machines etc) is modelled as a class which inherits from VisualObject.

A VisualObject is on the one hand something displayed on the screen, with its own boundary rectangle. On the other hand the user may click on it which causes an information window to appear. For that aspect VisualObject inherits from Window.

The inheritance relationship is shown below using the arrow symbol (→):

```
Window        → VisualObject
VisualObject  → Batch, ModelFactory, ControlZone, ControlZoneItem
ControlZone   → ControlZone1, ControlZone2, ..., ControlZone8
ControlZoneItem → Controller, Station, Transporter, In_process_store,
                  Raw_material_store, Second_side_buffer, Warehouse
Controller    → Second_side_controller, Screen_print_controller,
               Placement_controller_1, Placement_controller_2,
               Ips_controller, Bottleneck_controller,
               Test_repair_controller, Warehouse_controller
Station       → Screen_printing_station, Placement_station_1,
               Placement_station_2, Reflow_clean_station,
               Test_repair_station
Transporter   → Belt, Batch_mover, Board_mover
Port          → RPort, BPort, DPort
```

ControlZone is an abstract class with subclasses ControlZone1...ControlZone8. Each control zone has a local array items, with contains references to its controller, manufacturing units and transporting units. It has also a live script and arrays of communication scripts, for transferring batches and messages.

```plaintext
class ControlZone: public VisualObject
{
public:
int n;            /* index number of the control zone */
int nItems;       /* number of items in the control zone */
ControlZoneItem** items; /* items in the control zone */
virtual void initialize(); /* creates items */

ControlZone (char* name, int n, char* nameBox,
             short x1, short y1, short x2, short y2);
-ControlZone ();

... /* functions for drawing and user input */

scripts bat[20 i]<-->(BatchPtr b?);
  msg[20 i]<-->(MsgCode c?!, MsgData d?)
```

237
void ControlZone2::initialize()
{
  nItems =3;
  items =new ControlZoneItem* [nItems];
  items[0]=new Screen_print_controller("Screen-Print-Controller",_);
  items[1]=new Screen_printing_station("SP",b(2,0),b(2,1),d(2,1),_);
  items[2]=new Belt("Belt 4",b(2,1),b(0,0),b(3,0),b(3,1),d(2,2),d(3,0),_);
}

The model factory, the control zones and their items have to become active in the simulation.
For this purpose they get a live script. The model factory starts with a reset action, then it lets it
eight control zones live, and in parallel it generates orders to send to the warehouse. In turn, a
control zone lives by letting its items live in parallel:

ModelFactory::live = {reset();
  ( TIMES(8), {zones[pass+1]}->live), sendOrders
ControlZone ::live = TIMES(nItems), {items[pass]}->live

(Braces are needed syntax telling the Scriptic preprocessor that items[pass] denotes an object
instead of a script call.) The C++ despatch mechanism for virtual functions allows that the live
scripts of different types of items (belts, machines etc.) can have different implementations,
although these are called in a single loop.

A screen print station lives like any station by repeatedly sending and receiving messages and
batches, conforming to the earlier presented Process Algebra expression:

Station::live = MANY;
  sm(c_in,AVAIL );rm(c_in,GO!);rb(b_in);act(GO);smd(c_in,RO,ba);sb(b_ou)

MANY is an iterator that never stops, like for(;;). The scripts sb, rb, sm, rm, smd, rmd are
for sending and receiving batches and messages without or with batch data. ControlZoneItem
implements these efficiently using the channels msg and bat. One reason for this refinement is
conformance to the original PSF specification. rb and sb set and use the variable ba, present in
all ControlZoneItems, to the currently present batch. c_in, b_in, b_ou are variables denoting
the ports for communication with the controller and for inbound and outbound messages.

A belt has a somewhat more complicated life: it may receive and react to variants of the GO
message: GO_MM, GO_MS, GO_SM, GO_SS. Here M and S denote the main stream and the
side stream. E.g. after a GO_SM message the belt will receive a batch from the side input
stream, transport it, notify the controller, and send the batch to the main output stream.
The screen print controller controls both the screen print station and the belt. It directs empty A and B batches to the side output stream of the belt so that these go to component placement station 1. C boards will go through the main output stream. Non-empty A and B boards are not expected, and would result in an error message. Another parallel task by the controller is to transfer product requests from request port 6 to port 7.

```cpp
Screen_print_controller::live
= MsgData d;
(MANY; rm(c(2,0),HO!));
(MANY; rm(c(2,1),HO!));
if(d.batch->has(A,B_EMPTY)
    || d.batch->has(B,B_EMPTY))
    (rm(c(2,2),AVAIL!); sm(c(2,2),GO_MS))
else
    if(d.batch->has(C,B_EMPTY)
        || d.batch->has(C,B_HALF))
        (rm(c(2,2),AVAIL!); sm(c(2,2),GO_MM))
    else
        {ErrorUnexpectedMessage(d)}
),
transferRequests(6,7)
```

The **User Interface**

The main simulation window looks like:

![User Interface](image)

**figure 3**
The main script is

```plaintext
ModelFactory::mainscript = (MANY; live+restartCmd+pausings)
+ timer

ModelFactory::restartCmd = Command->(IDB_RESTART !); {SimTime=0.0}
```

This main script performs simulation runs until the user exits the program using the `file` menu. The script call `restartCmd` happens if the user clicks the mouse on the restart button (labeled II) in the main simulation window. It sets the simulation time variable `SimTime` back to 0. Then `restartCmd` completes successfully and the other two scripts, `live` and `pausings`, are terminated as well, since these run in an or-parallel expression.

The script call `pausings` never completes since it is an eternal loop of:

- let the user click the mouse on the pause button
- many times execute a dummy code fragment with a high priority, which effectively blocks all other activity
- let the user click the mouse on the pause button again, now to stop the pause

Apart from this there is a timer script which repeatedly executes a code fragment with duration 1. This gives the simulation program each simulation second the opportunity to update the simulation clock display in the main window.

The user may also adjust the `speed` and `steps` variables. The `steps` variable denotes the number of pixels into which the visual batch movement is divided. The smaller `steps`, the smoother the animation but also the more graphics computation power is needed. `Speed` is the maximum allowed ratio of elapsing simulation time and elapsing real time. The ratio is in practice also restricted by available processing power, the value of `steps`, and the number of moving batches. The Scriptic interface with the windowing system adds extra time delays as needed for the simulation not to go faster than `speed`.

![Warehouse](image)

*figure 4*

When the user clicks the mouse on a control zone item a dialog window will pop up with information on the item: the current batch, if any, and the ports. A scroll bar lets the user adjust the duration setting that a batch will spent at the item. Two other scroll bars specify the minimum gap and the maximum gap between the products of a single batch; a batch starts to move with the first product. The second product starts moving as soon as it lags the maximum gap behind, etc. If the first product slows down then the second will continue its higher speed until it lags only minimum gap behind, etc.
Transporting a batch is implemented by an iteration of moving it in the right direction until it has arrived at its destination. This has been programmed using the function \texttt{Batch::move}. The call to \texttt{Batch::move} sets the new and draws position, depending on the item's speed and the \texttt{step} variable. \texttt{Batch::move} returns the duration the move takes. If the duration is 0 the transport is ready and the iteration ends.

\begin{verbatim}
ControlZoneItem::transport(MsgCode destination)
 = int result;
 {result = ba->move(destination); duration = result}; while (result>0)
\end{verbatim}

Pressing \texttt{Info} button in the main window yields a Messages dialog, showing messages that processes try, manage and fail to send and to receive. The user may filter out message logs by code (e.g., \texttt{GO, GO\_MM}), by control zone, and by event (try, failure, actual communication):

![figure 5](image)

**Results**

**Animation**

The resulting program offers smooth animation. Batch movements do not flicker on the screen. One may change the speed of a conveyor belt by dragging the speed control in its information window. If one does so while a batch is transported over the belt then the speed change is instantly visible; the change is not effectuated only after the mouse button is released.

**Speed**

On a Pentium 90 MHz PC with SVGA graphics, it takes about 235 seconds to get 3 batches from the Raw Material Store to the In-Process Store (with the C batch making 1 loop), when the movement step size is 1 pixel. It takes 44 seconds with a 10 pixel step size. Speed seems to be limited by graphics, not by controlling activities in the scriptic run-time system. The program is an order of magnitude faster than a simulation with the PSF specification, where computation of single actions can take seconds up to minutes.

**IPS deadlock**

The simulation program has been tested with several kinds of input for product orders: some rather random, other where extreme cases: one with a sequence of orders for 3 C products, and one with a sequence of orders for 1 C product. With the latter the simulation unexpectedly entered an implicit deadlock situation. If only 1 product was in an in-process store, which was also ordered, and if a batch with another product just arrived for the in-process store, then the controller could send a hold message to the store too early. Then the control zone would enter a
deadlock situation, eventually blocking the entire factory. This deadlock was also present in the original PSF specification. The deadlock can therefore in principle be reproduced by the PSF specification, but this would currently take hours, if not days or weeks. It is very unlikely that extensive testing with the PSF simulator could have shown this deadlock if it was not known in advance.

Conclusion

Evaluating Specifications

Scriptic-C++ proved to be a valuable tool for evaluating specifications in Process Algebra. Using Scriptic-C++, a process simulation may run orders of magnitude faster than with PSF. Any Scriptic-C++ program may offer a graphical interface and animation, by the expense of additional down-to-earth C++ programming effort and increased processing overhead. The combination of speed and an advanced user interface has turned out to be essential for detecting a non-trivial specification error in the model factory specification.

Compared to PSF as a formalism, Scriptic-C++ offers object-orientedness and more expressiveness for iterations and communication. Because of state recording through variables in objects, functions and scripts less parameters and function calls as parameters are needed than in PSF. This appears to make specifications easier to make and to read, but at the same it excludes the possibility for static verification. This makes Scriptic-C++ a real programming language, as opposed to a formal specification language in the usual meaning. Strictly speaking however, Scriptic is a formal specification language, since it specifies behavior in a way that a computer can understand. It deserves the title executable specification language, but only as much as other programming languages (assembler, Pascal, C, C++, ...) do.

Scriptic-C++ is not meant to be a rigorous formal method. It does not need to be, since PSF already exists. Process Algebra researchers may use Scriptic-C++ as a supplement to PSF. Others may use it just for programming simulations and user interfaces.

It may be argued whether complex specifications like the model factory can or should be validated by simulation, rather than by formal techniques. The latter can in principle prove a system to be correct or incorrect, whereas simulation, and testing in general, can only show the presence of errors. However, complex specifications are often intractable for formal techniques. Moreover, if it would be possible to verify a specification against a set of correctness criteria, then these criteria may still be inadequate (and even the verification may be erroneous). Complete correctness criteria may be hidden and informal, residing inside the mind. Simulation with animation may therefore be helpful to determine whether specifications are correct.

This is illustrated by the unexpected deadlock detection in the in-process store. This detection was possible by the animation capabilities and speed of the program. The PSF simulator was able to reproduce it, but it is likely that without prior knowledge of the existence of the deadlock, an order of magnitude more testing effort should have been spent to detect the deadlock. With formal verification techniques the deadlock could not practically have been discovered.
Object oriented process modelling

In programming, objects and processes are related notions. Objects in object-oriented languages are usually passive entities that encapsulate data items, on which operations work that are available to the inner and outer world. Processes in parallel languages are concurrently active program threads, each with its own data. Several languages combine the two approaches. In general these offer a subset of the following features:

- passive objects (C++, Scriptic-C++, Smalltalk-80)
- active objects with a single associated process (POOL-T)
- active objects with multiple associated processes (Scriptic-C++)
- processes not associated to objects (Scriptic-C++, Smalltalk-80, Concurrent-C, Ada)

[PN91] distinguishes three main language approaches:

- Orthogonal: concurrent execution is independent of objects.
- Homogeneous: there is a single combined notion of process and object.
- Heterogeneous: both active and passive objects are provided

Scriptic-C++ could be classified as Orthogonal: it adds object-oriented and process-oriented features in a sense independently to C. This makes it allow for both active and passive objects, thus making it Heterogeneous as well.

The homogeneous languages, among which POOL-T [A87] may seem the most elegant at first sight. However, the practice of modelling calls for lightweight passive objects. Processes not associated to objects are also often convenient. And last but not least the use of active objects with multiple associated processes is often underestimated. An object can have several more or less independent aspects which may be modelled by parallel processes. This appeared in the current project for instance in the live scripts of the controllers and in the act script of an control zone item, which lets the item operate on a batch and transport it at the same time. Parallel aspect modeling was applied as well in other modelling exercises:

In a simulation of an administrative office, an office worker performs tasks during working hours, and he keeps his agenda:

```object
officeWorker::live = workingHours, performsTasks, agenda.live
```

In a simulation of a bank, each account has transactions, sends balance reports, and computes interest, with monthly or yearly intervals:

```object
bankAccount::live = transactions, sendBalanceReports, computeInterest
```

Future work

The main effort in building the simulation program was managing the user interface; the translation from PSF specifications was straightforward and relatively fast finished. Since this project new PSF specifications were made of the model factory with a centralized control. It is well possible to adjust the current Scriptic-C++ simulation in the same direction with modest effort.

Another idea is to remove all control zone items except for the controllers, and to have these communicate with the actual items in the model factory. The communication scripts will have to be adapted, so that they become more like existing scripts that read user input from the keyboard and the mouse.
References


Fischer's Protocol in Timed Process Algebra

Jan Joris Vereijken

Department of Computing Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

http://www.win.tue.nl/win/cs/fm/janjoris/
janjoris@acm.org

April 21, 1995

Abstract

Timed algebraic process theories can be developed with quite different purposes in mind. One can aim for theoretical results about the theory itself (completeness, expressiveness, decidability), or one can aim for practical applicability to non-trivial protocols. Unfortunately, these aims do not go well together. In this paper we take two theories, which are probably of the first kind, and try to find out how well suited they are for practical verifications.

We verify Fischer's protocol for mutual exclusion in the settings of discrete-time process algebra (ACP<sub>dt</sub>) and real-time process algebra (ACP<sub>rt</sub>). We do this by transforming the recursive specification into an equivalent linear specification, and then dividing out the maximal bisimulation relation. The required mutual exclusion result can then be found by reasoning about the obtained process graph.

Finally, we consider the ease of the verification, and ways to adapt the theory to make it more practical. It will turn out that the theories investigated are quite unsatisfactory when verifying real-life protocols.

1991 Mathematics Subject Classification: 68Q10, 68Q22, 68Q60.
Keywords: Fischer's protocol, process algebra, real time, discrete time, ACP, mutual exclusion, verification.

Note: These investigations were supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organisation for Scientific Research (NWO).

Contents

1 Introduction ......................................................... 2
2 Fischer's protocol .................................................. 3
   2.1 History of the protocol ...................................... 3
   2.2 Proof requirements .......................................... 3
   2.3 First informal description ................................... 4
   2.4 Second informal description ................................ 5
1 Introduction

In the past decade a lot of research has been done on algebraic process theories, the three most prominent ones being CCS [Mil89], CSP [Hoa85], and ACP [BW90]. Although these theories are well established in their untimed version, there is no consensus regarding versions of CCS, CSP, and ACP extended with time.

Admitted, there have been many proposals, some of them quite successful. But most of these timed theories were geared towards theoretical results. As a consequence, a lot of papers have been published regarding completeness results, expressivity results, decidability results, and so on, but almost no paper (one exception is [Hil95]) gives an actual verification of a real system, not even a toy system.

In this paper we will present such a more-or-less real system, namely Fischer's protocol for mutual exclusion [Fis85, Lam87]. We will try to prove the correctness of this protocol, which relies heavily on time aspects, using two timed ACP theories. First, we do a verification in a setting of discrete time, using the theory ACP_{dt} devised by Baeten and Bergstra [BB92a]. Secondly, a verification is done in a setting of dense time, using the theory ACP_{ur}. This theory was devised by Klusener [Klu93], and is closely related to ACP_{p}, the standard real-time extension of ACP by Baeten and Bergstra [BB91].
The whole point of this exercise lies in the following two questions. First, "How well suited are the current timed extensions of ACP to verify real-life systems?", and secondly, "With real-life verifications in mind, what modifications to the theory are desirable?".

Note that this paper supercedes the technical report [Ver94]; it is largely the same, but corrects some important errors and omissions.

2 Fischer's protocol

In this section we give a short history of Fischer's protocol and discuss the proof requirements. After that an informal description of the protocol is given, together with an informal correctness argument.

2.1 History of the protocol

The protocol we examine is a mutual exclusion protocol, first proposed by Fischer [Fis85], and later studied in [Lam87, SBM92, AL92a, JPXZ94]. None of these studies uses process algebra to prove correctness, they all rely on some form of temporal logic or Floyd-Hoare logic. Instead of using atomic test-and-set instructions or semaphores, as is nowadays often done to assure mutual exclusion, Fischer's protocol only assumes atomic reads and writes to a shared variable (when the first mutual exclusion protocols were developed in the late 1960s all exclusion protocols were of the "shared variable kind" [Dij65, Knu66, dB67, Lam74], later on researchers have more concentrated on the "semaphore kind" of protocol). Mutual exclusion in Fischer's Protocol is guaranteed by carefully placing bounds on the execution times of the instructions, leading to a protocol which is very simple, and relies heavily on time aspects. This makes it an ideal candidate for the purpose we have in mind, namely to try to verify (using process algebra) a not too difficult protocol which still has quite intricate timing aspects.

2.2 Proof requirements

What does one need to prove in the case of a mutual exclusion protocol? Strange as is it may seem, this is not all that clear. In the literature one sees requirements like:

- The actual property of mutual exclusion: only one component may be in its critical section at any time,
- Symmetry between the components,
- No assumptions about the execution times of statements (obviously not satisfied in our case!),
- Liveness: there should always be some process that is able to proceed,
- No starvation: it may not be that a component is permanently prohibited from entering its critical section,
- Various kinds of fairness: a component should get its fair share (in various senses) of being allowed to proceed into its critical section,
• Loosely connectedness: when one component deadlocks (outside its critical section), this should not affect the progress of the other components,

• Minimal overhead: the protocol should make a decision as soon it has enough information to do so,

and even more requirements. Some of these requirements are related (for example: symmetry guarantees most kinds of fairness), and each paper about mutual exclusion seems to have its own favorite subset of which ones to prove.

In the case of Fischer's protocol we choose, mostly following the earliest paper on mutual exclusion [Dij65], to prove the following three properties:

• Actual mutual exclusion between the two critical sections,

• Symmetry between the two components,

• No starvation.

We will not try to formalize these properties algebraically, as they do not lend themselves easily to this. This is more due to the shortcomings of the (current) algebraic approach than it is to unwillingness on our part; note for example that the above properties can indeed be very easily formalized using temporal logic. We will return to this subject in our conclusions.

2.3 First informal description

We will now describe the protocol in an informal way, giving an informal correctness argument. Assume the existence of a shared variable \( x \), to which atomic reads and writes are possible. Initially \( x \) equals zero. In Figure 1 we give Fischer's protocol expressed in pseudo-PASCAL. There are two components, running in parallel. The angle brackets ("(" , ")") denote atomicity.

**Component 1:**

```pascal
repeat
  repeat
    await \( x = 0 \); \( x = 1 \); \( \langle \text{delay} \rangle \); until \( x = 1 \);
  critical section 1; \( x = 0 \); until false;
```  

**Component 2:**

```pascal
repeat
  repeat
    await \( x = 0 \); \( x = 2 \); \( \langle \text{delay} \rangle \); until \( x = 2 \);
  critical section 2; \( x = 0 \); until false;
```  

Figure 1: Fischer's protocol, first informal version.

The protocol proceeds as follows. Initially, the value of the shared variable is 0. When component 1 observes that \( x \) is 0, it will write the value 1 to \( x \). After that, it waits for some
time, and if $x$ then still has the value 1, it is safe to enter the critical section. Component 2 works in a similar way (using 2 instead of 1), and both components run in parallel.

The mutual exclusion property of the protocol is based on the following observation. The delay operation causes component 1 to wait sufficiently long so that, if component 2 had read the value of $x$ in its await statement before the component 1 executed its $x = 1$ assignment, then component 2 will have completed the following $x = 2$ statement. Therefore, it can never happen that component 1 falls through its until statement, entering critical section 1, while component 2 is still about to execute its $x = 2$ assignment. This guarantees mutual exclusion. By symmetry, the argument also holds the other way around.

2.4 Second informal description

Let us try to make the reasoning from the previous section a bit more solid by exactly indicating the possible durations of the statements. First of all, the await statement may take anywhere between 0 and $\infty$ time units after $x$ becomes 0. The assignments $x = i$ are supposed to take between $a$ and $a'$ time units, and the delay statements between $d$ and $d'$ time units, for fixed non-negative values $a \leq a'$ and $d \leq d'$ over some totally-ordered time domain. Furthermore, assume that $a' < d$, i.e. the delay always takes longer than an assignment. For simplicity sake, the read actions $x = i$ are supposed to take 0 time units, and the critical section may take any time, including 0 time units. Writing $\langle \text{action} \rangle_t^r$ for an atomic action that happens between $t$ and $t'$ time units after it has been enabled, we arrive at the protocol of Figure 2.

**Component 1:**

```plaintext
repeat
  repeat
    await \langle x = 0 \rangle_0^\infty;
    \langle x = 1 \rangle_a^{a'};
    \langle \text{delay} \rangle_d^{d'};
  until \langle x = 1 \rangle_0^\infty;
  \langle \text{critical section 1} \rangle;
  \langle x = 0 \rangle_a^{a'};
until false;
```

**Component 2:**

```plaintext
repeat
  repeat
    await \langle x = 0 \rangle_0^\infty;
    \langle x = 2 \rangle_a^{a'};
    \langle \text{delay} \rangle_d^{d'};
  until \langle x = 2 \rangle_0^\infty;
  \langle \text{critical section 2} \rangle;
  \langle x = 0 \rangle_a^{a'};
until false;
```

Figure 2: Fischer's protocol, second informal version.

Remember we assumed that $0 \leq a \leq a' < d \leq d' < \infty$. Now we have that if component 2 falls through its await statement, it will complete its $x = 2$ assignment within at most $a'$ time units. If component 1 would have happened to complete its $x = 1$ assignment just after component 2 fell through its await, it will take component 1 at least $d$ time units to complete its delay. As $a' < d$, when component 1 reaches the until $\langle x = 1 \rangle$ statement, component 2 will have completed its $x = 2$ assignment. Therefore, the value of $x$ has stabilized, and component 2 can safely enter its critical section.

249
As a final remark: note that Fischer's protocol can be trivially generalized to any number \( n \geq 2 \) of components. This generalization, however, we will not examine.

3 A verification using discrete-time ACP

In this section we will prove the correctness of the protocol of Figure 2 on the preceding page for the special case where \( a = a' = 0 \) and \( d = d' = 1 \). We will use (a subset of) the discrete-time process algebra ACP\(_{dt}\) as described in [BB92a]. This is the simplest special case one can imagine. The resulting proof provides a clear illustration of all key issues, without becoming too cluttered with technicalities.

3.1 A short note on ACP\(_{dt}\)

First, we briefly describe the essential extensions that ACP\(_{dt}\) has to model time. Intuitively, ACP\(_{dt}\) is very much like plain ACP [BW90], with the exception that a process may have the ability to "let time pass for one time unit". If in an ACP\(_{dt}\) process some component still wants to do an action, it can, and no time passes. If however all components agree to let time pass, one unit of time will pass (in other words: the current time slice ends, or the clock ticks).

This willingness to let time pass is expressed with the discrete time unit delay operator \( \sigma_d \). The expression \( \sigma_d(X) \) denotes the process that can do no action anymore in the current time slice, but has the ability to let time pass for one time unit. If that unit of time finally "happens" (time does not really pass in a discrete setting, it just happens!), the process \( \sigma_d(X) \) turns into the process \( X \). This is denoted by \( \sigma_d(X) \xrightarrow{\sigma} X \), as if \( \sigma \) were a special action denoting a tick of the clock.

In Table 1 the operational semantics of BPA\(_{dt}\) is given. BPA\(_{dt}\) is the subset of ACP\(_{dt}\) that only has \( \cdot \), +, and \( \sigma_d \) as operators. For the full axiomatization and semantics of ACP\(_{dt}\) see [BB92a]. Besides the \( \sigma_d \) operator, we will only use one more special operator from ACP\(_{dt}\), namely the unbounded start delay of a process \( X \), denoted by \( [X]^\omega \). The
expression \([X]^\omega\) is the process that can do exactly the same things that \(X\) can, and it is also always willing to let time pass. This is shown in the operational semantics in Table 2.

\[
\frac{X \xrightarrow{a} X'}{[X]^\omega \xrightarrow{a} [X']^\omega} \quad \frac{X \xrightarrow{\sigma} \sqrt{X}}{[X]^\omega \xrightarrow{\sigma} \sqrt{[X]^\omega}}
\]

Table 2: Operational semantics of the unbounded start delay.

### 3.2 Abstraction in ACP\(_{dt}\)

Abstraction from internal actions in ACP\(_{dt}\) is defined as expected. It is axiomatized as shown in Table 3. If \(A\) denotes the set of all possible actions, we have that \(I \subseteq A, a \in A \cup \{\delta, \tau\}\), and \(\sigma \not\in A\). The variables \(x\) and \(y\) denote processes.

\[
\begin{align*}
\tau_I(a) &= a & \text{if } a \not\in I & \text{T11} \\
\tau_I(a) &= \tau & \text{if } a \in I & \text{T12} \\
\tau_I(x + y) &= \tau_I(x) + \tau_I(y) & \text{T13} \\
\tau_I(x \cdot y) &= \tau_I(x) \cdot \tau_I(y) & \text{T14} \\
\tau_I(\sigma_d(x)) &= \sigma_d(\tau_I(x)) & \text{T18}
\end{align*}
\]

Table 3: Axioms for the abstraction operator \(\tau_I\) in ACP\(_{dt}\).

There exists an operator \(\tau_\sigma\) that abstracts from time steps, but we do not describe it here because we will not need it. The non-consecutive numbering of the axioms is due to historical reasons.

### 3.3 Fischer's protocol formalized in ACP\(_{dt}\)

Having introduced the relevant operators for ACP\(_{dt}\), we are now ready to give a formal specification of Fischer's protocol FP\(_{dt}\), using ACP\(_{dt}\) in Figure 3 on the following page. As said before, we specify the special case where \(a = a' = 0\) and \(d = d' = 1\).

This specification can be understood intuitively in the following way. There are three processes running concurrently, namely \(A, B,\) and \(V\). The processes \(A\) and \(B\) model "Component 1" and "Component 2" of Figure 1 respectively, and the process \(V\) models the variable \(x\).

The process \(V\) can be in one of three states: \(V_0, V_1,\) or \(V_2\), corresponding to the possible values of \(x\). In any state \(V_i\), \(V\) is capable of sending the message \(x = i\), signaling that the value of \(x\) is currently \(i\), after which \(V\) will continue in state \(V_i\). Furthermore, \(V\) is in any state \(V_i\) capable of receiving the message \(x = j\) for any \(j\), indicating that \(x\) is being assigned with the value \(j\). After such an assignment \(V\) will continue in state \(V_j\). Finally, \(V\) is always capable of letting time pass. The process \(V\) constructed in this way behaves as a "variable server": if process \(A\) or \(B\) wants to assign a value \(i\) to \(x\) it performs the action
\[ A = A_0 \]
\[ A_0 = [r(x = 0)]^\omega \cdot A_1 \]
\[ A_1 = s(x = 1) \cdot A_2 \]
\[ A_2 = \sigma_d(A_3) \]
\[ A_3 = (r(x = 0) + r(x = 2)) \cdot A_0 + r(x = 1) \cdot A_4 \]
\[ A_4 = \text{EnterCS}_1 \cdot A_5 \]
\[ A_5 = \text{LeaveCS}_1 \cdot A_6 \]
\[ A_6 = s(x = 0) \cdot A_0 \]

\[ B = B_0 \]
\[ B_0 = [r(x = 0)]^\omega \cdot B_1 \]
\[ B_1 = s(x = 2) \cdot B_2 \]
\[ B_2 = \sigma_d(B_3) \]
\[ B_3 = (r(x = 0) + r(x = 1) + B_0 + r(x = 2) \cdot B_4 \]
\[ B_4 = \text{EnterCS}_2 \cdot B_5 \]
\[ B_5 = \text{LeaveCS}_2 \cdot B_6 \]
\[ B_6 = s(x = 0) \cdot B_0 \]

\[ V = V_0 \]
\[ V_0 = (r(x = 0) + s(x = 0)) \cdot V_0 + r(x = 1) \cdot V_1 + r(x = 2) \cdot V_2 + \sigma_d(V_0) \]
\[ V_1 = (r(x = 1) + s(x = 1)) \cdot V_1 + r(x = 0) \cdot V_0 + r(x = 2) \cdot V_2 + \sigma_d(V_1) \]
\[ V_2 = (r(x = 2) + s(x = 2)) \cdot V_2 + r(x = 0) \cdot V_0 + r(x = 1) \cdot V_1 + \sigma_d(V_2) \]

\[ y(r(\alpha), s(\alpha)) = c(\alpha) \text{ for } \alpha \in \{ x = i, x = i \mid i \in \{0, 1, 2\} \} \]

\[ H = \{ r(\alpha), s(\alpha) \mid \alpha \in \{ x = i, x = i \mid i \in \{0, 1, 2\} \} \} \]

\[ \text{FP}_{dt} = \partial_H(A \parallel B \parallel V) \]

Figure 3: Fischer’s protocol in discrete time (ACP_{dt}).

\[ s(x = i). \] If it wants to check if \( x \) has the value \( i \), it performs the action \( r(x = i) \). (The idea to construct the variable server in this way was taken from [Nie90].)

The process \( A \) is constructed as follows. First (in state \( A_0 \)) it waits for an undetermined amount of time till \( x \) is 0 ([\( r(x = 0) \)]^\omega). Then (in state \( A_1 \)) it sets \( x \) to 1 (\( s(x = 1) \)). After that (in state \( A_2 \)), it waits till the end of the time slice (\( \sigma_d(A_3) \)). When it has arrived in state \( A_3 \), it will examine the contents of \( x \), and either jump back to \( A_0 \) (if \( x = 0 \) or \( x = 2 \)), or continue with state \( A_4 \) (if \( x = 1 \)). Thereafter, it enters its critical section, leaves it again, and resets \( x \) back to 0 in state \( A_6 \), after which it repeats the whole procedure over again. The process \( B \) is constructed in the same way as \( A \) is. The entire protocol, \( \text{FP}_{dt} \), now consists of the processes \( A, B, \) and \( V \) running concurrently. Note that the assignment takes no time (\( a = a' = 0 \)) and the delay takes one time unit (\( d = d' = 1 \)).

### 3.4 The correctness of \( \text{FP}_{dt} \)

In order to prove the protocol \( \text{FP}_{dt} \) correct, we first rewrite the recursive equations of Figure 3 into an equivalent linear system of equations (i.e. one that does not contain the operators \( \parallel, \partial_H \), or \( \tau_I \) anymore). We arrive at the system of 32 equations given in Appendix A (where also the details of the linearization are given). Using this linear system we then construct the process graph of \( \text{FP}_{dt} \) given in Figure 4 on the next page (Note that all \( c(\ldots) \) edges remain unlabelled, as labelling them would obscure the picture, and that \text{EnterCS}_1 \) is abbreviated to E1. The other Enter and Leave actions are abbreviated in a
similar way).

Figure 4: The process graph of FPdt.

Now define the set $I$ of internal actions as:

$$I = \{ c(\alpha) \mid \alpha \in \{ x = i, x = i \mid i \in \{0, 1, 2\} \} \}$$

(i.e. all communication actions) and rename these actions into $\tau$, yielding the process graph of $\tau_1(FPdt)$. On this graph we compute the maximal rooted branching auto bisimulation, which gives us the equivalence classes $X_A, \ldots, X_H$ given below:

$$X_A = \{ X_0, X_8, X_{17}, X_{19}, X_{20}, X_{28}, X_{29} \}$$

$$X_B = \{ X_1, X_2, X_3, X_{21}, X_{30} \}$$

$$X_C = \{ X_{10}, X_{11}, X_{22}, X_{23} \}$$

$$X_D = \{ X_{12}, X_{13}, X_{14}, X_{15} \}$$

$$X_E = \{ X_{16}, X_{18} \}$$

$$X_F = \{ X_4, X_9, X_{24}, X_{31} \}$$

$$X_G = \{ X_5, X_6, X_{25}, X_{26} \}$$

$$X_H = \{ X_7, X_{27} \}$$
When we divide out this equivalence relation we arrive at the reduced process graph given in Figure 5. On this graph we will now be able to check all required properties very easily.

![Figure 5: The reduced process graph of \( \tau_I(FP_{dt}) \).](image)

The three requirements for mutual exclusion we chose in Section 2.2 are satisfied for the following reasons:

- **Actual mutual exclusion**: It can be easily seen from the graph of Figure 5 that an \( \text{EnterCS}_1 \) action is always immediately followed by a \( \text{LeaveCS}_1 \) action, and the same holds for \( \text{EnterCS}_2 \) and \( \text{LeaveCS}_2 \). Therefore, it cannot be the case that both components are in their critical section at the same time.

- **Symmetry**: As the graph is symmetrical with respect to the paths from the root, through the critical sections of the components, back to the root, it is clear that the protocol is symmetrical with respect to the components.

- **No starvation**: Whichever state the protocol is in, there is always a path leading to each component's critical section. Therefore, using fairness, it cannot be the case that one component is permanently prohibited from entering its critical section.

This completes the proof of the correctness of \( FP_{dt} \).

### 4 A verification using real-time ACP

In this section we will prove the correctness of the protocol of Figure 2, for the almost completely general case where \( 0 < a < a' < d < d' < 2a \) (this is only a very minor restriction of the general case, but it reduces the state-space significantly). We will use (a subset of) the real-time process algebra \( \text{ACP}_{ur} \) ("ACP with urgent actions and relative time") as described in [Klu93].
4.1 A short note on ACP$_{ur}$

Again, we briefly describe the essential extensions that ACP$_{ur}$ has to model time. To begin with, every action is postfixed with a non-negative real number between square brackets. One could have, for example, $a[3]$, meaning "the process that performs the action $a$ at exactly 3 time units from now". If the parameter is 0, e.g. $a[0]$, it means "execute the action immediately". Sequential composition and the choice operator behave as expected. So, for example, the process $a[2] \cdot b[3]$ started at time 4 does $a$ at time 6 and $b$ at time 9. The process $a[2] + b[3]$ started at time 2 can either do $a$ at time 4 or $b$ at time 5; if at any moment after 4 the action $a$ has not been observed, one knows that the choice has been made for $b$ at time 5. While this is all very intuitive, the exact semantics of ACP$_{ur}$ is still quite intricate, and we will make no attempt at making it formal here. Please see [Klu93] and [BB91].

The single most important construct in ACP$_{ur}$ is the integral construct. It is a generalized sum over an interval of time. For example, $\int_{t-1}^{t} a[t]$ denotes the process that can do the action $a$ at any time between 1 and 2 time units from now, inclusively. The upper bound may be infinity: $\int_{t}^{\infty} a$ denotes the process that can do $a$ once at some time, including right now, and all moments arbitrarily far into the future. (Please note that the integral construct as described here is a rather crude simplification of the one given by Klusener. This simplified one however is all we need.)

For the special cases where for an integral $\int_{v=p}^{q} P(v)$ we do not have $0 \leq p \leq q$, we will use the convention that:

$$\int_{v=p}^{q} P(v) = \begin{cases} \int_{v=0}^{q} P(v) & \text{if } p \leq 0 \text{ and } q \geq 0 \\ \delta[0] & \text{if } q < 0 \text{ or } p > q \end{cases}$$

This means that every integral can be written in a normal form in the following way.

$$\int_{v=p}^{q} P(v) = (0 \leq q \text{ and } p \leq q) \rightarrow \int_{v=\max(0,p)}^{q} P(v)$$

Here $\rightarrow$ is used to form a so called guarded command. For a boolean expression $\phi$ and a process $P$ the guarded command is defined, dependent on the current valuation $V$ of the free variables, as:

$$\phi \rightarrow P = \begin{cases} P & \text{if } [\phi \equiv \text{true}]_V \\ \delta[0] & \text{otherwise} \end{cases}$$

This normal form for integrals will be called guard-prefixed normal form. The usefulness of this normal form lies in the fact that is immediate clear from the guard expression whether or not the integral is empty. This will be important when defining abstraction operators.

4.2 Abstraction operators on ACP$_{ur}$

We define two abstraction operators. First, a general abstraction operator:

$$\tau_I : ACP_{ur} \to ACP_{ur}$$
that renames actions in \( I \) into \( \tau \)-actions, and secondly, a *time abstraction operator*:

\[
\tau_t : ACP_{ur} \rightarrow ACP
\]

that transforms a (timed) \( ACP_{ur} \)-term into an (untimed) \( ACP \)-term by “throwing away all timing information”. The general abstraction operator is axiomatized as shown in Table 4. We have that \( I \subseteq A \), \( a \in A \cup \{\delta, \tau\} \), \( v \in \mathbb{R}_{\geq 0} \), and \( \phi \) a boolean expression that may contain free variables. The variables \( x \) and \( y \) denote processes.

\[
\begin{align*}
\tau_t(a[v]) &= a[v] & \text{if } a \notin I & & \text{TIT1} \\
\tau_t(a[v]) &= \tau[v] & \text{if } a \in I & & \text{TIT2} \\
\tau_t(x + y) &= \tau_t(x) + \tau_t(y) & & \text{TIT3} \\
\tau_t(x \cdot y) &= \tau_t(x) \cdot \tau_t(y) & & \text{TIT4} \\
\tau_t(\phi \rightharpoonup x) &= \phi \rightharpoonup \tau_t(x) & & \text{TIT5}
\end{align*}
\]

Table 4: Axioms for the general abstraction operator \( \tau_t \) in \( ACP_{ur} \)

The time abstraction operator is axiomatized as shown in Table 5. We have that \( a \in A \cup \{\delta, \tau\} \), \( \phi \) a boolean expression that may contain free variables, and \( p, q, v \in \mathbb{R}_{\geq 0} \) with \( p \leq q \). The variables \( x \) and \( y \) denote processes, and \( FV(x) \) stands for the set of all free variables \( x \) contains.

\[
\begin{align*}
\tau_t(a[v]) &= a & & & & \text{TTT1} \\
\tau_t(x + y) &= \tau_t(x) + \tau_t(y) & & & & \text{TTT2} \\
\tau_t(x \cdot y) &= \tau_t(x) \cdot \tau_t(y) & & & & \text{TTT3} \\
\tau_t(\int_{v=p}^{q} a[v]) &= a & & & & \text{TTT4} \\
\tau_t(\int_{v=p}^{q} a[v] \cdot x) &= a \cdot \tau_t(x) & \text{if } v \notin FV(x) & & \text{TTT5} \\
\tau_t(\int_{v=p}^{q} a[v] \cdot x) &= a \cdot \tau_t(\int_{v=p}^{q} x) & \text{if } v \in FV(x) & & \text{TTT6} \\
\tau_t(\phi \rightharpoonup x) &= \tau_t(x) & \text{if } \exists v[\phi \equiv \text{true}]_v & & \text{TTT7}
\end{align*}
\]

Table 5: Axioms for the time abstraction operator \( \tau_t \) in \( ACP_{ur} \)

Axiom TTT7, where \( V \) denotes a valuation of the free variables, is formulated this way because we do not want that \( \tau_t(\phi \rightharpoonup x) \) would equal \( \tau_t(x) \) if there is no way to satisfy \( \phi \). On the other hand, we want that \( \tau_t(\phi \rightharpoonup x) \) does equal \( \tau_t(x) \) if \( \phi \) can be satisfied by some valuation of the free variables, even if there are other valuations that do not satisfy \( \phi \).

### 4.3 Expansion theorems for \( ACP_{ur} \)

Before we are able to do the actual verification, we first need to have expansion theorems, which become quite intricate as real time gets involved. We start by giving a version that assumes there is no communication at all:
An expansion theorem for ACP\_ur without communication:

\[
\left\| \sum_{i=1,\ldots,n} a_i[v_i] \cdot P_i \right\| = \sum_{i=1,\ldots,n} \int_{v_i=p_i}^{\min\{q_i, j=1,\ldots,n\}} a_i[v_i] \cdot \left( P_i \parallel \prod_{j=1,\ldots,n}^{q_j-v_j} a_j[v_j] \cdot P_j \right)
\]

This theorem can be understood in the following way. We have \(n\) processes of the form \(\int_{v_i=p_i}^{a_i[v_i]} a_i[v_i] \cdot P_i\) running in parallel, where \(a_i\) is an action, and \(P_i\) is an arbitrary process expression not containing free variables. The action \(a_i\) is enabled in the interval \([p_i, q_i]\) from the start of the process. How does this big merge of \(n\) processes expand? Obviously, the first thing that is going to happen is some action \(a_i\), so we sum over all those. At what time can this action happen? No earlier than the time when it gets enabled, but also not later than the minimum of the latest times of all other actions. This explains the bounds \(p_i\) and \(\min\{q_j, j=1,\ldots,n\}\) on the integral preceding \(a_i\). What remains is \(P_i\) running in parallel with the other components, albeit that \(v_i\) time units have passed; hence the \(p_j-v_j\) and \(q_j-v_j\) bounds on the inner integrals. To make this somewhat clearer we give the special case where \(n=2\):

\[
\left( \int_{v=p}^{q} a[v] \cdot P \right) \parallel \left( \int_{w=r}^{s} b[w] \cdot Q \right)
= \int_{v=p}^{\min\{q,s\}} a[v] \cdot \left( P \parallel \int_{w=r}^{s-v} b[w] \cdot Q \right) + \int_{w=r}^{\min\{a,s\}} b[w] \cdot \left( \int_{v=p-w}^{q-w} a[v] \cdot P \parallel Q \right)
\]

When we allow handshaking (communication with the restriction that \(a \mid b \mid c\) is always \(\delta\) for all actions \(a\), \(b\), and \(c\)) we arrive at the following theorem:

An expansion theorem for ACP\_ur with handshaking:

\[
\left\| \sum_{i=1,\ldots,n} \int_{v_i=p_i}^{q_i} a_i[v_i] \cdot P_i \right\| = \sum_{i=1,\ldots,n} \int_{v_i=p_i}^{\min\{q_i, j=1,\ldots,n\}} a_i[v_i] \cdot \left( P_i \parallel \prod_{j=1,\ldots,n}^{q_j-v_j} a_j[v_j] \cdot P_j \right) + \sum_{1 \leq i < j \leq n} \int_{v=p_{\max(p_i,p_j)}}^{\max(p_i,p_j)} (a_i \mid a_j)[v] \cdot \left( P_i \parallel P_j \parallel \prod_{k=1,\ldots,n}^{q_k-v_k} a_k[v_k] \cdot P_k \right)
\]

This theorem can be understood in much the same way as the previous one, although this time there are extra summands, corresponding to all possible (necessarily two-way)
communications. At what time can such a communication \((a_i, a_j)\) take place? The first possible time is when both \(a_i\) and \(a_j\) are enabled, so we get \(\max(p_i, p_j)\) as the lower bound of the integral. The upper bound is \(\min\{q_j \mid j = 1, \ldots, n\}\), just as it was in the previous theorem. Here, the special case for \(n = 2\) is as follows:

\[
\left(\int_{v=p}^{q} a[v] \cdot P \right) \parallel \left(\int_{w=r}^{s} b[w] \cdot Q \right) = \int_{v=p}^{\min(q,s)} a[v] \cdot \left( P \parallel \int_{w=r-v}^{s-v} b[w] \cdot Q \right) + \int_{w=r}^{\min(q,s)} b[w] \cdot \left( \int_{v=p-w}^{q-w} a[v] \cdot P \parallel Q \right) + \int_{u=\max(p,r)}^{\min(q,s)} (a \mid b) [u] \cdot (P \parallel Q)
\]

We will not give proofs for the above theorems, as that would go beyond the scope of the paper. For more general expansion theorems, and proofs, see [Klu93] and [Fok94].

### 4.4 Fischer's protocol formalized in ACP\(ur\)

We are now ready to give a formally specified implementation of Fischer’s protocol FP\(ur\), using ACP\(ur\) in Figure 6 on the next page. The protocol, FP\(ur\), consists just like in the discrete-time case of three concurrent processes: \(A\), \(B\), and \(V\). The process \(V\) models the variable \(x\) in such a way that assignments and reads can take place at any moment. The process \(A\) is also constructed much the same as in discrete time; the only difference being the execution time interval bounds on all actions. In state \(A_0\) we do a \(\int_{v=0}^{\infty} t(x = 0) [v]\), meaning that we will wait at least till \(x\) is 0, but possibly longer. After that, by performing \(\int_{v=a}^{d} s(x = 1) [v]\) we assign the value 1 to \(x\) somewhere within the interval \([a, a']\), and so on. The values \(a, a', d,\) and \(d'\) are positive constants such that \(a < a' < d < d' < 2a\).

Note that this time the “delay actions” are explicit modelled: \(i_1\) and \(i_2\).

### 4.5 The correctness of FP\(ur\)

Using the material from Sections 4.1 and 4.3 we are now able to rewrite the specification for FP\(ur\) in Figure 6 into an equivalent linear system of equations. The way to do this, however, is far more complex than it was in the case of FP\(dt\); see Appendix B for full details.

We ultimately arrive at the linear system of 51 equations given in Section B.2. Unfortunately, the process graph of FP\(ur\) induced by this linearization is too complex to draw it without obscuring most of the crucial parts, so we will not give it here. Continuing with the verification, we define the set of internal actions as:

\[
I = \{ c(\alpha) \mid \alpha \in \{ x = i, x = i \mid i \in \{0, 1, 2\} \} \} \cup \{i_1, i_2\}
\]

i.e. all communication actions, and \(i_1\) and \(i_2\) (the “delay” actions). As we have made sure that all guarded commands in the linearization can be satisfied, it is very easy to compute
A = A_0
A_0 = \int_{v=0}^{\infty} r(x = 0)[v] \cdot A_1
A_1 = \int_{v=d}^{d'} r(x = 1)[v] \cdot A_2
A_2 = \int_{v=d}^{d'} l_1[v] \cdot A_3
A_3 = (r(x = 0)[0] + r(x = 2)[0]) \cdot A_0 + r(x = 1)[0] \cdot A_4
A_4 = \text{EnterCS}_1[0] \cdot A_5
A_5 = \int_{v=0}^{\infty} \text{LeaveCS}_1[v] \cdot A_6
A_6 = \int_{v=d}^{d'} s(x = 0)[v] \cdot A_0

B = B_0
B_0 = \int_{v=0}^{\infty} r(x = 0)[v] \cdot B_1
B_1 = \int_{v=d}^{d'} r(x = 1)[v] \cdot B_2
B_2 = \int_{v=d}^{d'} l_2[v] \cdot B_3
B_3 = (r(x = 0)[0] + r(x = 1)[0]) \cdot B_0 + r(x = 2)[0] \cdot B_4
B_4 = \text{EnterCS}_2[0] \cdot B_5
B_5 = \int_{v=0}^{\infty} \text{LeaveCS}_2[v] \cdot B_6
B_6 = \int_{v=d}^{d'} s(x = 0)[v] \cdot B_0

V = V_0
V_0 = \int_{v=0}^{\infty} (r(x = 0)[v] + s(x = 0)[v]) \cdot V_0 + r(x = 1)[v] \cdot V_1 + r(x = 2)[v] \cdot V_2
V_1 = \int_{v=d}^{d'} (r(x = 1)[v] + s(x = 1)[v]) \cdot V_0 + r(x = 0)[v] \cdot V_1 + r(x = 2)[v] \cdot V_2
V_2 = \int_{v=d}^{d'} (r(x = 2)[v] + s(x = 2)[v]) \cdot V_2 + r(x = 0)[v] \cdot V_0 + r(x = 1)[v] \cdot V_1

\gamma(r(\alpha), s(\alpha)) = c(\alpha) \text{ for } \alpha \in \{ x = i, x = i \mid i \in \{0, 1, 2\} \}

H = \{ r(\alpha), s(\alpha) \mid \alpha \in \{ x = i, x = i \mid i \in \{0, 1, 2\} \}

\text{FP}_{ur} = \partial_H (A \parallel B \parallel V)

Figure 6: Fischer's protocol in real time (ACP_{ur}).
\[ \tau_t \circ \tau_t(F_{\text{pr}}) \]. On the process graph of \( \tau_t \circ \tau_t(F_{\text{pr}}) \) we then compute the maximal rooted branching auto bisimulation, which gives us the following equivalence classes:

\[ X_A = \{X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, X_{26}, X_{27}, X_{28}, X_{29}, X_{30}, X_{31}, X_{32}, X_{33}, X_{34}, X_{35}, X_{36}, X_{37}, X_{38}, X_{39}, X_{40}, X_{41}, X_{42} \} \]

\[ X_B = \{X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, X_{26}, X_{27}, X_{28}, X_{29}, X_{30}, X_{31}, X_{32}, X_{33}, X_{34}, X_{35}, X_{36}, X_{37}, X_{38}, X_{39}, X_{40}, X_{41}, X_{42}, X_{43}, X_{44}, X_{45}, X_{46}, X_{47} \} \]

\[ X_C = \{X_{18}, X_{21}, X_{24} \} \]

\[ X_D = \{X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22}, X_{23}, X_{24}, X_{25}, X_{26}, X_{27}, X_{28}, X_{29}, X_{30}, X_{31}, X_{32}, X_{33}, X_{34}, X_{35}, X_{36}, X_{37}, X_{38}, X_{39}, X_{40}, X_{41}, X_{42}, X_{43}, X_{44}, X_{45}, X_{46}, X_{47} \} \]

\[ X_E = \{X_{7}, X_{34}, X_{35} \} \]

When we divide out this equivalence relation, we arrive at the reduced process graph given in Figure 7. On this graph we can now check all required properties, in exactly the same way as we did earlier for \( \tau_t(F_{\text{pr}}) \):

- **Actual mutual exclusion:** It can be easily seen from the graph of Figure 7 that an EnterCS action is always immediately followed by a LeaveCS action, and the same holds for EnterCS and LeaveCS. Therefore, it cannot be the case that both components are in their critical section at the same time.

- **Symmetry:** As the graph is symmetrical with respect to the paths from the root, through the critical sections of the components, back to the root, it is clear that the the protocol is symmetrical with respect to the components.

- **No starvation:** Whichever state the protocol is in, there is always a path leading to each component's critical section. Therefore, using fairness, it cannot be the case that one component is permanently prohibited from entering its critical section.

We find that all properties hold, which completes the proof of the correctness of \( F_{\text{pr}} \).

5 Conclusions

As we have seen Fischer's protocol can be proven correct quite satisfactorily using algebraic techniques. This however does not mean too much; it is an almost trivial protocol, that has been solved time and time again using all kinds of formalisms. But it is not all
that bad either; see for example [SBM92] where an (incomplete) proof is given of Fischer's protocol. When it would be written out in full detail, that proof would be about as long and tedious as ours is [Sch94], and the same probably holds for a detailed temporal logic proof [Aba94].

Looking at our proof one observes that, although conceptually very clear and easy, the inner workings required a lot of bothersome and failure-prone computations. It seems valid to doubt whether all these calculations were really necessary. Much of their complexity results from the fact that the theories we used are based on bisimulation semantics, which preserves very many moments of choice, while that was not at all required for the proof we were constructing. Furthermore, the so-called "algebraic advantage", i.e. the ability to calculate with processes without having to write out the entire state space, is almost absent. This is disappointing; although the algebraic advantage manifests itself very clearly when verifying protocols that do not exhibit much internal parallelism, such as the Alternating Bit Protocol (see for example [BW90]), it seems to be lost in the verification of Fischer's protocol, which has very much internal parallelism. As a result, we get the worst of both worlds: the state explosion from naive model checking, and the complicated term rewriting from real-time process algebra...

Let us however not become too pessimistic. It has become clear that algebraic theories can indeed be applied to protocols with intricate real-time aspects. It would be unrealistic to expect that these theories, which were designed without much regard for their practical application, would in their unmodified form be splendidly suited for real-life verifications. There is a lot of room for tuning these theories towards more practicality. We see at least three directions in which we would like to proceed, preferably in all three at the same time.

- First of all, it would be nice to have a real-time process algebra that does not lean on bisimulation semantics only. It is simply not always a good idea to preserve all internal moments of choice. If we had a theory based on, say, ready semantics or failure semantics, we would probably gain some of our algebraic advantage back. But maybe abandoning bisimulation semantics altogether would be too crude. A more sophisticated and subtle approach could be to abstract only from those internal moments of choice we really want to abstract from. This could be implemented by introducing a special choice operator next to the ordinary $\tau$. For example, the delayed choice of [BM94] or a $\tau$-angelic choice.

- Secondly, it might be profitable to augment process algebra with a (limited) form of temporal logic. Looking at the linearization process of FP$_{ur}$ in Appendix B, it is clear that much of the calculations involved are needed to manage the exact ranges of the parameters of the variables. Or, on a conceptually higher level: the calculations get complicated because we do not have adequate tools for denoting the precise flow of time. When working with a hybrid process algebra-temporal logic theory (still predominantly algebraic!), these complications would probably not have arisen. See for example [BBB93], where this approach is investigated.

- Thirdly, we might just as well admit that real-time verifications are difficult, and probably will remain so for some years to come. Therefore, it may be advisable to have computer tools at our disposal. One could for example imagine an "process algebra calculator" which, like an ordinary arithmetic calculator, could assist
in performing complex calculations. This way we could put the human back into
the driver's seat of the verification process, instead of having him stumble in the
dark while juggling complex process terms. Assuming we can indeed exploit our al­
gebraic advantage, this approach could be very interesting. A combination of easy
process term manipulation and powerful algebraic techniques could lead to verifi­
cations that are both short and easy to construct.

Concluding, we can say that the verification of Fischer's protocol by algebraic means as
given in this paper is not very straightforward. But nonetheless, there are several promis­
ing directions for future research that could lead to new, algebraically oriented theories
that may perform much better for real-life verifications.

A The linearization of \( FP_{dt} \)

A.1 Methodology

Finding a linear specification for \( FP_{dt} \) is very easy (at least in principle): one just repeat­
edly applies the expansion theorem for \( ACP_{dt} \). All new states reached are numbered con­
secutively using process variables \( X_i \). Linearizing in a depth-first fashion we arrive at
the 32 processes \( X_0, \ldots, X_{31} \) given below in Section A.2. The protocol \( FP_{dt} \) itself corre­
sponds to the variable \( X_0 \).

Please note that the equations in the following section were produced by hand, and
therefore initially had several mistakes in them. To attempt to find these, we have per­
formed several computerized sanity checks on the hand-generated data file. For example,
it is obvious that the entire system is symmetrical with respect to the \( A \) and \( B \) compo­
nent, and this can be quite easily checked. This means that if there are any mistakes left,
it must be that we have made exactly the same mistake two times over, in a symmetrical
way. Some more checks were made, exploiting similar intuitively known facts about the
process graph, and by old-fashioned human proofreading.

Finally, the data file was automatically converted to "\( \text{\LaTeX} \)-format" by means of a com­
puter, in order to minimize the chances of mistakes in that step of the process.

A.2 The obtained linear system

\[
FP_{dt} = \partial_H(A \parallel B \parallel V) \\
= \partial_H(A_0 \parallel B_0 \parallel V_0) \\
= X_0
\]

\[
X_0 = \partial_H(A_0 \parallel B_0 \parallel V_0) \\
= c(x = 0) \cdot \partial_H(A_1 \parallel B_0 \parallel V_0) + c(x = 0) \cdot \partial_H(A_0 \parallel B_1 \parallel V_0) + \\
\sigma_d(\partial_H(A_0 \parallel B_0 \parallel V_0)) \\
= c(x = 0) \cdot X_1 + c(x = 0) \cdot X_2 + \sigma_d(X_0)
\]
\[ X_1 = \partial_H(A_1 \parallel B_0 \parallel V_0) \]
\[ = c(x = 1) \cdot \partial_H(A_2 \parallel B_0 \parallel V_1) + c(x = 0) \cdot \partial_H(A_1 \parallel B_1 \parallel V_0) \]
\[ = c(x = 1) \cdot X_{23} + c(x = 0) \cdot X_3 \]

\[ X_2 = \partial_H(A_0 \parallel B_1 \parallel V_0) \]
\[ = c(x = 0) \cdot \partial_H(A_1 \parallel B_1 \parallel V_0) + c(x = 2) \cdot \partial_H(A_0 \parallel B_2 \parallel V_2) \]
\[ = c(x = 0) \cdot X_3 + c(x = 2) \cdot X_4 \]

\[ X_3 = \partial_H(A_1 \parallel B_1 \parallel V_0) \]
\[ = c(x = 1) \cdot \partial_H(A_2 \parallel B_1 \parallel V_1) + c(x = 2) \cdot \partial_H(A_1 \parallel B_2 \parallel V_2) \]
\[ = c(x = 1) \cdot X_9 + c(x = 2) \cdot X_{10} \]

\[ X_4 = \partial_H(A_0 \parallel B_2 \parallel V_2) \]
\[ = \sigma_d(\partial_H(A_0 \parallel B_3 \parallel V_2)) \]
\[ = \sigma_d(X_5) \]

\[ X_5 = \partial_H(A_0 \parallel B_3 \parallel V_2) \]
\[ = c(x = 2) \cdot \partial_H(A_0 \parallel B_4 \parallel V_2) \]
\[ = c(x = 2) \cdot X_6 \]

\[ X_6 = \partial_H(A_0 \parallel B_4 \parallel V_2) \]
\[ = \text{EnterCS}_2 \cdot \partial_H(A_0 \parallel B_5 \parallel V_2) \]
\[ = \text{EnterCS}_2 \cdot X_7 \]

\[ X_7 = \partial_H(A_0 \parallel B_5 \parallel V_2) \]
\[ = \text{LeaveCS}_2 \cdot \partial_H(A_0 \parallel B_6 \parallel V_2) \]
\[ = \text{LeaveCS}_2 \cdot X_8 \]

\[ X_8 = \partial_H(A_0 \parallel B_6 \parallel V_2) \]
\[ = c(x = 0) \cdot \partial_H(A_0 \parallel B_0 \parallel V_0) \]
\[ = c(x = 0) \cdot X_0 \]

\[ X_9 = \partial_H(A_2 \parallel B_1 \parallel V_1) \]
\[ = c(x = 2) \cdot \partial_H(A_2 \parallel B_2 \parallel V_2) \]
\[ = c(x = 2) \cdot X_{24} \]
\[X_{10} = \partial_H(A_1 \parallel B_2 \parallel V_2)\]
\[= c(x = 1) \cdot \partial_H(A_2 \parallel B_2 \parallel V_1)\]
\[= c(x = 1) \cdot X_{11}\]

\[X_{11} = \partial_H(A_2 \parallel B_2 \parallel V_1)\]
\[= \sigma_d(\partial_H(A_3 \parallel B_3 \parallel V_1))\]
\[= \sigma_d(X_{12})\]

\[X_{12} = \partial_H(A_3 \parallel B_3 \parallel V_1)\]
\[= c(x = 1) \cdot \partial_H(A_4 \parallel B_3 \parallel V_1) + c(x = 1) \cdot \partial_H(A_3 \parallel B_0 \parallel V_1)\]
\[= c(x = 1) \cdot X_{13} + c(x = 1) \cdot X_{14}\]

\[X_{13} = \partial_H(A_4 \parallel B_3 \parallel V_1)\]
\[= \text{EnterCS}_1 \cdot \partial_H(A_5 \parallel B_3 \parallel V_1) + c(x = 1) \cdot \partial_H(A_4 \parallel B_0 \parallel V_1)\]
\[= \text{EnterCS}_1 \cdot X_{18} + c(x = 1) \cdot X_{15}\]

\[X_{14} = \partial_H(A_3 \parallel B_0 \parallel V_1)\]
\[= c(x = 0) \cdot \partial_H(A_4 \parallel B_0 \parallel V_1)\]
\[= c(x = 0) \cdot X_{15}\]

\[X_{15} = \partial_H(A_4 \parallel B_0 \parallel V_1)\]
\[= \text{EnterCS}_1 \cdot \partial_H(A_5 \parallel B_0 \parallel V_1)\]
\[= \text{EnterCS}_1 \cdot X_{16}\]

\[X_{16} = \partial_H(A_5 \parallel B_0 \parallel V_1)\]
\[= \text{LeaveCS}_1 \cdot \partial_H(A_6 \parallel B_0 \parallel V_1)\]
\[= \text{LeaveCS}_1 \cdot X_{17}\]

\[X_{17} = \partial_H(A_6 \parallel B_0 \parallel V_1)\]
\[= c(x = 0) \cdot \partial_H(A_0 \parallel B_0 \parallel V_0)\]
\[= c(x = 0) \cdot X_0\]

\[X_{18} = \partial_H(A_5 \parallel B_3 \parallel V_1)\]
\[= \text{LeaveCS}_1 \cdot \partial_H(A_6 \parallel B_3 \parallel V_1) + c(x = 1) \cdot \partial_H(A_5 \parallel B_0 \parallel V_1)\]
\[= \text{LeaveCS}_1 \cdot X_{19} + c(x = 1) \cdot X_{16}\]
\[ X_{19} = \partial_H(A_6 \parallel B_3 \parallel V_1) \]
\[ = c(x = 0) \cdot \partial_H(A_6 \parallel B_3 \parallel V_0) + c(x = 1) \cdot \partial_H(A_0 \parallel B_0 \parallel V_1) \]
\[ = c(x = 0) \cdot X_{20} + c(x = 1) \cdot X_17 \]

\[ X_{20} = \partial_H(A_0 \parallel B_3 \parallel V_0) \]
\[ = c(x = 0) \cdot \partial_H(A_1 \parallel B_3 \parallel V_0) + c(x = 0) \cdot \partial_H(A_0 \parallel B_0 \parallel V_0) \]
\[ = c(x = 0) \cdot X_{21} + c(x = 0) \cdot X_0 \]

\[ X_{21} = \partial_H(A_1 \parallel B_3 \parallel V_0) \]
\[ = c(x = 1) \cdot \partial_H(A_2 \parallel B_3 \parallel V_1) + c(x = 0) \cdot \partial_H(A_1 \parallel B_0 \parallel V_0) \]
\[ = c(x = 1) \cdot X_{22} + c(x = 0) \cdot X_1 \]

\[ X_{22} = \partial_H(A_2 \parallel B_3 \parallel V_1) \]
\[ = c(x = 1) \cdot \partial_H(A_2 \parallel B_0 \parallel V_1) \]
\[ = c(x = 1) \cdot X_{23} \]

\[ X_{23} = \partial_H(A_2 \parallel B_0 \parallel V_1) \]
\[ = \sigma_d(\partial_H(A_3 \parallel B_0 \parallel V_1)) \]
\[ = \sigma_d(X_{14}) \]

\[ X_{24} = \partial_H(A_2 \parallel B_2 \parallel V_2) \]
\[ = \sigma_d(\partial_H(A_3 \parallel B_3 \parallel V_2)) \]
\[ = \sigma_d(X_{25}) \]

\[ X_{25} = \partial_H(A_3 \parallel B_3 \parallel V_2) \]
\[ = c(x = 2) \cdot \partial_H(A_0 \parallel B_3 \parallel V_2) + c(x = 2) \cdot \partial_H(A_3 \parallel B_4 \parallel V_2) \]
\[ = c(x = 2) \cdot X_5 + c(x = 2) \cdot X_{26} \]

\[ X_{26} = \partial_H(A_3 \parallel B_4 \parallel V_2) \]
\[ = c(x = 2) \cdot \partial_H(A_0 \parallel B_4 \parallel V_2) + \text{Enter}CS_2 \cdot \partial_H(A_3 \parallel B_5 \parallel V_2) \]
\[ = c(x = 2) \cdot X_6 + \text{Enter}CS_2 \cdot X_{27} \]

\[ X_{27} = \partial_H(A_3 \parallel B_5 \parallel V_2) \]
\[ = c(x = 2) \cdot \partial_H(A_0 \parallel B_5 \parallel V_2) + \text{Leave}CS_2 \cdot \partial_H(A_3 \parallel B_6 \parallel V_2) \]
\[ = c(x = 2) \cdot X_7 + \text{Leave}CS_2 \cdot X_{28} \]
\( X_{28} = \partial_H(A_3 \parallel B_6 \parallel V_2) \)
\[ = c(x = 2) \cdot \partial_H(A_0 \parallel B_0 \parallel V_2) + c(x = 0) \cdot \partial_H(A_3 \parallel B_0 \parallel V_0) \]
\[ = c(x = 2) \cdot X_8 + c(x = 0) \cdot X_{29} \]
\( X_{29} = \partial_H(A_3 \parallel B_0 \parallel V_0) \)
\[ = c(x = 0) \cdot \partial_H(A_0 \parallel B_0 \parallel V_0) + c(x = 0) \cdot \partial_H(A_3 \parallel B_1 \parallel V_0) \]
\[ = c(x = 0) \cdot X_0 + c(x = 0) \cdot X_{30} \]
\( X_{30} = \partial_H(A_3 \parallel B_1 \parallel V_0) \)
\[ = c(x = 0) \cdot \partial_H(A_0 \parallel B_1 \parallel V_0) + c(x = 2) \cdot \partial_H(A_3 \parallel B_2 \parallel V_2) \]
\[ = c(x = 0) \cdot X_2 + c(x = 2) \cdot X_{31} \]
\( X_{31} = \partial_H(A_3 \parallel B_2 \parallel V_2) \)
\[ = c(x = 2) \cdot \partial_H(A_0 \parallel B_2 \parallel V_2) \]
\[ = c(x = 2) \cdot X_4 \]

**B The linearization of FPur**

**B.1 Methodology**

Finding a linear specification for FPur is a lot more difficult than it was in the discrete-time case. This is caused by the fact that most of the variables used in the linear recursive system are parameterized with a time stamp \( v \in \mathbb{R}^{\geq 0} \). So in order to perform calculations on these parameterized variables, a lot of bookkeeping has to be done. There are three things that have to be watched:

- First, when introducing new parameterized variables, one has to compute the range the parameter might take. The range will always take the form of a non-empty closed interval \([t, t'] \subseteq \mathbb{R}^{\geq 0}\). It is important that all points of this interval can actually be reached by the protocol.

For example, when expanding \( X_3(v) \) (where \( 0 \leq v \leq a' \)) on page 24, we need to introduce new parameterized variables \( X_9 \) and \( X_{10} \). The summand for \( X_9 \) is:

\[
(v \leq a' - a) \mathbin{:} \int_{u-a}^{a'-v} c(x = 1)[u] \cdot X_9(u + v)
\]

To determine the range the parameter of \( X_9 \) can take, we first observe that although \( 0 \leq v \leq a' \), the guarded command \((v \leq a' - a) \mathbin{:} \int_{u-a}^{a'-v} c(x = 1)[u] \cdot X_9(u + v)\) effectively limits this range to \( 0 \leq v \leq a' - a \). As \( u \) can range from \( a \) to \( a' - v \), this means that \( u + v \) ranges from \( a \) (when \( v = 0 \) and \( u = a \)) to \( a' \) (when \( u = a' - v \), and \( v \) is anything between \( 0 \) and \( a' - a \)). So, if we substitute the fresh variable \( v' \) for \( u + v \), the parameter range for \( X_9(v') \) is \( a \leq v' \leq a' \). This range is indicated just above the definition of \( X_9 \), on page 25.
• Secondly, two parameterized variables that both represent the same subprocess, but differ in the range which their parameter may take, have to be considered different variables, and get different names.

This happens for example between variables $X_{15}(v)$ and $X_{47}(v)$. Both are defined as:

$$\partial_H \left( \int_{w=d-v}^{d' - v} i_1[w] \cdot A_3 \parallel B_0 \parallel V_1 \right)$$

but for $X_{15}(v)$ we have $d - a' \leq v \leq d'$, while for $X_{47}(v)$ we have $d - a' + a \leq v \leq d'$.

• Thirdly, at some points in the calculation it is crucial that the information provided by the range of a parameter be used to simplify the equations involved. Often this manifests itself as a guarded command evaluating to false. These steps are the essence of the verification: the paths that are cut off because the guarded command never evaluates to true represent paths that would violate the safety properties, such as actual mutual exclusion, we are trying to prove.

This can be observed for example in the expansion of $X_9(v)$, where $a \leq v \leq a'$. There we arrive at a summand:

$$(v \leq a' - d) \rightarrow \int_{u=d}^{a' - v} i_1[u] \cdot \partial_H \left( A_2 \parallel \int_{w=a-v-u}^{d' - v - u} s(x = 2)[w] \cdot B_2 \parallel V_1 \right)$$

Because $a' < d$ (intuitively: a delay always takes longer than an assignment), $a' - d < 0$. As $0 < a \leq v$ we have $v > a' - d$, so the condition is always false. Therefore, the whole summand vanishes into $\delta$. This is just what we need, because choosing that summand would lead us onto a path where both components could enter the critical section simultaneously.

At this point, it becomes clear why we put in the extra restriction $d' < 2a$ (see page 10). Although this restriction is irrelevant for the correctness of the protocol, it does ensure that $d' - d < a$. This greatly reduces the size of the linear specification, as it intuitively cuts off all paths where the one component is delaying “forever” while the other component is repeatedly entering and leaving its critical section. See the expansion of $X_{26}(v)$ for an example of this.

After linearizing in a depth-first fashion we arrive at the 51 processes $X_0, \ldots, X_{50}$ given in Section B.2. The protocol $FP_{ur}$ itself corresponds to the variable $X_0$. We have made sure to indicate exactly the range the parameters may take. Furthermore, we ultimately put all integrals in guard-prefixed normal form.

As in the discrete-time case, several computerized sanity checks were made on the hand-generated data. This time we also had the opportunity to exploit the information provided by the parameter ranges. Translating the data file into "\text{T\LaTeX}-format" was again done automatically, although some final manual editing was necessary to achieve a satisfactory layout.

B.2 The obtained linear system

$$FP_{ur} = \partial_H (A \parallel B \parallel V) = \partial_H (A_0 \parallel B_0 \parallel V_0)$$
\[ = X_0 \]

\[ X_0 = \partial_H(A_0 \parallel B_0 \parallel V_0) \]
\[ = \int_{v=0}^{\infty} c(x = 0)\{v\} \cdot \partial_H(A_1 \parallel B_0 \parallel V_0) + \int_{v=0}^{\infty} c(x = 0)\{v\} \cdot \partial_H(A_0 \parallel B_1 \parallel V_0) \]
\[ = \int_{v=0}^{\infty} c(x = 0)\{v\} \cdot X_1 + \int_{v=0}^{\infty} c(x = 0)\{v\} \cdot X_2 \]
\[ = (\text{true}) :- \int_{v=0}^{\infty} c(x = 0)\{v\} \cdot X_1 + (\text{true}) :- \int_{v=0}^{\infty} c(x = 0)\{v\} \cdot X_2 \]

\[ X_1 = \partial_H(A_1 \parallel B_0 \parallel V_0) \]
\[ = \int_{v=a}^{a'} c(x = 1)\{v\} \cdot \partial_H(A_2 \parallel B_0 \parallel V_1) + \]
\[ \int_{v=0}^{a'} c(x = 0)\{v\} \cdot \partial_H\left( \int_{w=a-v}^{a'-v} s(x = 1)\{w\} \cdot A_2 \parallel B_1 \parallel V_0 \right) \]
\[ = \int_{v=a}^{a'} c(x = 1)\{v\} \cdot X_{41} + \int_{v=0}^{a'} c(x = 0)\{v\} \cdot X_{42}(v) \]
\[ = \int_{v=a}^{a'} c(x = 1)\{v\} \cdot X_{41} + \int_{v=0}^{a'} c(x = 0)\{v\} \cdot X_{42}(v) \]
\[ = (\text{true}) :- \int_{v=a}^{a'} c(x = 1)\{v\} \cdot X_{41} + (\text{true}) :- \int_{v=0}^{a'} c(x = 0)\{v\} \cdot X_{42}(v) \]

\[ X_2 = \partial_H(A_0 \parallel B_1 \parallel V_0) \]
\[ = \int_{v=0}^{a'} c(x = 0)\{v\} \cdot \partial_H\left( A_1 \parallel \int_{w=a-v}^{a'-v} s(x = 2)\{w\} \cdot B_2 \parallel V_0 \right) + \]
\[ \int_{v=a}^{a'} c(x = 2)\{v\} \cdot \partial_H(A_0 \parallel B_2 \parallel V_2) \]
\[ = \int_{v=0}^{a'} c(x = 0)\{v\} \cdot X_3(v) + \int_{v=a}^{a'} c(x = 2)\{v\} \cdot X_4 \]
\[ = (\text{true}) :- \int_{v=0}^{a'} c(x = 0)\{v\} \cdot X_3(v) + (\text{true}) :- \int_{v=a}^{a'} c(x = 2)\{v\} \cdot X_4 \]

For all \( v \), \( 0 \leq v \leq a' \):

\[ X_3(v) = \partial_H\left( A_1 \parallel \int_{w=a-v}^{a'-v} s(x = 2)\{w\} \cdot B_2 \parallel V_0 \right) \]
\[ = \int_{u=a}^{\min(a',a'-v)} c(x = 1)\{u\} \cdot \partial_H\left( A_2 \parallel \int_{w=a-v-u}^{a'-v-u} s(x = 2)\{w\} \cdot B_2 \parallel V_1 \right) + \]
\[ \int_{w=a-v}^{a'-v} c(x = 2)\{w\} \cdot \partial_H\left( \int_{u=a-w}^{a'-w} s(x = 1)\{u\} \cdot A_2 \parallel B_2 \parallel V_2 \right) \]
\[ = \int_{u=a}^{a'-v} c(x = 1)\{u\} \cdot \partial_H\left( A_2 \parallel \int_{w=a-v-u}^{a'-v-u} s(x = 2)\{w\} \cdot B_2 \parallel V_1 \right) + \]
\[ \int_{w=a-v}^{a'-v} c(x = 2)\{w\} \cdot \partial_H\left( \int_{u=a-w}^{a'-w} s(x = 1)\{u\} \cdot A_2 \parallel B_2 \parallel V_2 \right) \]
\[
\begin{align*}
&= \int_{u=a}^{a'-v} c(x = 1)[u] \cdot X_9(u + v) + \int_{w=a-v}^{a'-v} c(x = 2)[w] \cdot X_{10}(w) \\
&= (v \leq a' - a) \cdot \int_{u=a}^{a'-v} c(x = 1)[u] \cdot X_9(u + v) + \\
&\quad (\text{true}) \cdot \int_{w=\max(0,a-v)}^{a'-v} c(x = 2)[w] \cdot X_{10}(w) \\
&X_4 = \partial_H(A_0 \parallel B_2 \parallel V_2) \\
&= \int_{v=d}^{d'} i_2[v] \cdot \partial_H(A_0 \parallel B_3 \parallel V_2) \\
&= \int_{v=d}^{d'} i_2[v] \cdot X_5 \\
&= (\text{true}) \cdot \int_{v=d}^{d'} i_2[v] \cdot X_5 \\
&X_5 = \partial_H(A_0 \parallel B_3 \parallel V_2) \\
&= c(x = 2)[0] \cdot \partial_H(A_0 \parallel B_4 \parallel V_2) \\
&= c(x = 2)[0] \cdot X_6 \\
&X_6 = \partial_H(A_0 \parallel B_4 \parallel V_2) \\
&= \text{EnterCS}_2[0] \cdot \partial_H(A_0 \parallel B_5 \parallel V_2) \\
&= \text{EnterCS}_2[0] \cdot X_7 \\
&X_7 = \partial_H(A_0 \parallel B_5 \parallel V_2) \\
&= \int_{v=0}^{\infty} \text{LeaveCS}_2[v] \cdot \partial_H(A_0 \parallel B_6 \parallel V_2) \\
&= \int_{v=0}^{\infty} \text{LeaveCS}_2[v] \cdot X_8 \\
&= (\text{true}) \cdot \int_{v=0}^{\infty} \text{LeaveCS}_2[v] \cdot X_8 \\
&X_8 = \partial_H(A_0 \parallel B_6 \parallel V_2) \\
&= \int_{v=a}^{a'} c(x = 0)[v] \cdot \partial_H(A_0 \parallel B_0 \parallel V_0) \\
&= \int_{v=a}^{a'} c(x = 0)[v] \cdot X_0 \\
&= (\text{true}) \cdot \int_{v=a}^{a'} c(x = 0)[v] \cdot X_0 \\
\text{For all } v, a \leq v \leq a': \\
X_9(v) = \partial_H \left( A_2 \parallel \int_{v=a-v}^{a' - v} s(x = 2)[w] \cdot B_2 \parallel V_1 \right) \\
= \int_{u=d}^{\min(d',a'-v)} i_1[u] \cdot \partial_H \left( A_2 \parallel \int_{w=a-v-u}^{a'-v-u} s(x = 2)[w] \cdot B_2 \parallel V_1 \right) +
\end{align*}
\]
\begin{align*}
\min(d',\alpha'-v) \quad &c(x=2)[w] \cdot \partial_H \left( \int_{u=d-w}^{d'-w} i_1[u] \cdot A_3 \parallel B_2 \parallel V_2 \right) \\
= &\int_{u=d}^{\alpha'-v} i_1[u] \cdot \partial_H \left( A_2 \parallel \int_{u=d-w}^{d'-v-u} s(x=2)[w] \cdot B_2 \parallel V_1 \right) + \\
&\int_{u=d}^{\alpha'-v} c(x=2)[w] \cdot \partial_H \left( \int_{u=d-w}^{d'-w} i_1[u] \cdot A_3 \parallel B_2 \parallel V_2 \right) \\
= &\int_{u=d}^{\alpha'-v} i_1[u] \cdot \partial_H(\ldots) + \int_{w=a-v}^{\alpha'-v} c(x=2)[w] \cdot X_{28}(w) \\
= &\forall \alpha' \leq v \leq \alpha' \quad \forall v, 0 \leq v \leq \alpha': \\
X_{10}(v) &= \partial_H \left( \int_{w=a-v}^{\alpha'-v} s(x=1)[w] \cdot A_2 \parallel B_2 \parallel V_2 \right) \\
= &\min(d',\alpha'-v) \quad c(x=1)[w] \cdot \partial_H \left( A_2 \parallel \int_{u=d-w}^{d'-w} i_2[u] \cdot B_3 \parallel V_1 \right) + \\
&\min(d',\alpha'-v) \quad i_2[u] \cdot \partial_H \left( \int_{w=a-v}^{\alpha'-v-u} s(x=1)[w] \cdot A_2 \parallel B_3 \parallel V_2 \right) + \\
= &\int_{w=a-v}^{\alpha'-v} c(x=1)[w] \cdot \partial_H \left( A_2 \parallel \int_{u=d-w}^{d'-w} i_2[u] \cdot B_3 \parallel V_1 \right) + \\
&\int_{u=d}^{\alpha'-v} i_2[u] \cdot \partial_H \left( \int_{w=a-v}^{\alpha'-v-u} s(x=1)[w] \cdot A_2 \parallel B_3 \parallel V_2 \right) \\
= &\int_{w=a-v}^{\alpha'-v} c(x=1)[w] \cdot X_{11}(w) + \int_{u=d}^{\alpha'-v} i_2[u] \cdot \partial_H(\ldots) \\
= &\forall \alpha' \leq v \leq \alpha' \\
&\forall v, 0 \leq v \leq \alpha': \\
X_{11}(v) &= \partial_H \left( A_2 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) \\
= &\min(d',\alpha'-v) \quad i_1[u] \cdot \partial_H \left( A_3 \parallel \int_{w=d-v-u}^{d-v} i_2[w] \cdot B_3 \parallel V_1 \right) + \\
&\forall v, 0 \leq v \leq \alpha'
\end{align*}
For all \( v, d \leq v, d' \):

\[
\int_{w=d-v}^{\min(d', d'-v)} i_2[w] \cdot \partial_H \left( \int_{u=d-w}^{d'-w} i_1[u] \cdot A_3 \parallel B_3 \parallel V_1 \right)
\]

\[
= \int_{u=d}^{d'-v} i_1[u] \cdot \partial_H \left( A_3 \parallel \int_{w=d-v-u}^{d'-v-u} i_2[w] \cdot B_3 \parallel V_1 \right) + \int_{w=d-v}^{d'-v} i_2[w] \cdot \partial_H \left( \int_{u=d-w}^{d'-w} i_1[u] \cdot A_3 \parallel B_3 \parallel V_1 \right)
\]

\[
= \int_{u=d}^{d'-v} i_1[u] \cdot X_{12}(u + v) + \int_{w=d-v}^{d'-v} i_2[w] \cdot X_{13}(w)
\]

\[
= (v \leq d' - d) :\int_{u=d}^{d'-v} i_1[u] \cdot X_{12}(u + v) + (v > d) :\int_{w=d-v}^{d'-v} i_2[w] \cdot X_{13}(w)
\]

For all \( v, d \leq v, d' \):

\[
X_{12}(v) = \partial_H \left( A_3 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right)
\]

\[
= c(x = 1)[0] \cdot \partial_H \left( A_4 \parallel \int_{u=d-w}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) + (v \geq d) :\int_{w=d-v}^{d'-v} i_2[w] \cdot X_{14}(v)
\]

\[
= c(x = 1)[0] \cdot X_{12}(v) + (v \geq d) :\int_{w=d-v}^{d'-v} i_2[w] \cdot X_{14}
\]

\[
= c(x = 1)[0] \cdot X_{12}(v) + (v \geq d) :\int_{w=d-v}^{d'-v} i_2[w] \cdot X_{14}
\]

For all \( v, d - a' \leq v \leq d' \):

\[
X_{13}(v) = \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_3 \parallel V_1 \right)
\]

\[
= (v \geq d) :\int_{w=d-v}^{d'-v} i_1[w] \cdot \partial_H \left( A_4 \parallel B_3 \parallel V_1 \right) + c(x = 1)[0] \cdot \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_0 \parallel V_1 \right)
\]

\[
= (v \geq d) :\int_{w=d-v}^{d'-v} i_1[w] \cdot X_{14} + c(x = 1)[0] \cdot X_{15}(v)
\]

\[
X_{14} = \partial_H (A_3 \parallel B_3 \parallel V_1)
\]

\[
= c(x = 1)[0] \cdot \partial_H (A_4 \parallel B_3 \parallel V_1) + c(x = 1)[0] \cdot \partial_H (A_3 \parallel B_0 \parallel V_1)
\]

\[
= c(x = 1)[0] \cdot X_{14} + c(x = 1)[0] \cdot X_{15}
\]

For all \( v, d - a' \leq v \leq d' \):

\[
X_{15}(v) = \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_0 \parallel V_1 \right)
\]

\[
= \int_{w=d-v}^{d'-v} i_1[w] \cdot \partial_H (A_3 \parallel B_0 \parallel V_1)
\]

\[
= \int_{w=d-v}^{d'-v} i_1[w] \cdot X_{16}
\]

\[
= (true) :\int_{w=\max(0,d-v)}^{d'-v} i_1[w] \cdot X_{16}
\]
\[ X_{16} = \partial_H(A_3 \parallel B_0 \parallel V_1) \]
\[ = c(x = 1)[0] \cdot \partial_H(A_4 \parallel B_0 \parallel V_1) \]
\[ = c(x = 1)[0] \cdot X_{17} \]

\[ X_{17} = \partial_H(A_4 \parallel B_0 \parallel V_1) \]
\[ = \text{EnterCS}_1[0] \cdot \partial_H(A_5 \parallel B_0 \parallel V_1) \]
\[ = \text{EnterCS}_1[0] \cdot X_{18} \]

\[ X_{18} = \partial_H(A_5 \parallel B_0 \parallel V_1) \]
\[ = \int_{v=0}^{\infty} \text{LeaveCS}_1[v] \cdot \partial_H(A_6 \parallel B_0 \parallel V_1) \]
\[ = \int_{v=0}^{\infty} \text{LeaveCS}_1[v] \cdot X_{19} \]
\[ = (\text{true}) \iff \int_{v=0}^{\infty} \text{LeaveCS}_1[v] \cdot X_{19} \]

\[ X_{19} = \partial_H(A_6 \parallel B_0 \parallel V_1) \]
\[ = \int_{v=a}^{d'} c(x = 0) \cdot \partial_H(A_7 \parallel B_0 \parallel V_0) \]
\[ = \int_{v=a}^{d'} c(x = 0)[v] \cdot X_0 \]
\[ = (\text{true}) \iff \int_{v=a}^{d'} c(x = 0)[v] \cdot X_0 \]

\[ X_{20} = \partial_H(A_4 \parallel B_3 \parallel V_1) \]
\[ = \text{EnterCS}_1[0] \cdot \partial_H(A_5 \parallel B_3 \parallel V_1) + c(x = 1)[0] \cdot \partial_H(A_4 \parallel B_0 \parallel V_1) \]
\[ = \text{EnterCS}_1[0] \cdot X_{21} + c(x = 1)[0] \cdot X_{17} \]

\[ X_{21} = \partial_H(A_5 \parallel B_3 \parallel V_1) \]
\[ = \text{LeaveCS}_1[0] \cdot \partial_H(A_6 \parallel B_3 \parallel V_1) + c(x = 1)[0] \cdot \partial_H(A_5 \parallel B_0 \parallel V_1) \]
\[ = \text{LeaveCS}_1[0] \cdot X_{22} + c(x = 1)[0] \cdot X_{18} \]

\[ X_{22} = \partial_H(A_6 \parallel B_3 \parallel V_1) \]
\[ = c(x = 1)[0] \cdot \partial_H(A_7 \parallel B_0 \parallel V_1) \]
\[ = c(x = 1)[0] \cdot X_{19} \]

For all \( v, d \leq v \leq d' \):

\[ X_{23}(v) = \partial_H \left( A_4 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) \]
\[ = \text{EnterCS}_1[0] \cdot \partial_H \left( A_5 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) + \]
\[ (v \geq d) \iff i_2[0] \cdot \partial_H(A_4 \parallel B_3 \parallel V_1) \]
\[ = \text{EnterCS}_1[0] \cdot X_{24}(v) + (v \geq d) \iff i_2[0] \cdot X_{20} \]
\[ = \text{EnterCS}_1[0] \cdot X_{24}(v) + (\text{true}) \iff i_2[0] \cdot X_{20} \]

272
For all $v$, $d \leq v \leq d'$:

$$X_{24}(v) = \partial_H \left( A_5 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) +$$

$$\min \{ \max_{v}, d' \} \\text{LeaveCS}_1[u] \cdot \partial_H \left( A_6 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) +$$

$$\min \{ \max_{v}, d' \} \text{LeaveCS}_1[u] \cdot \partial_H \left( A_6 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) +$$

$$\min \{ \max_{v}, d' \} \\text{LeaveCS}_1[u] \cdot \partial_H \left( A_6 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) +$$

$$\min \{ \max_{v}, d' \} \left( \text{true} \right) \rightarrow \int_{u=0}^{d'-v} \text{LeaveCS}_1[u] \cdot X_{25}(u + v) + \int_{w=d-v}^{d'-v} i_2[w] \cdot X_{21}$$

For all $v$, $d \leq v \leq d'$:

$$X_{25}(v) = \partial_H \left( A_6 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_1 \right) +$$

$$\min \{ \max_{v}, d' \} \left( \text{true} \right) \rightarrow \int_{u=0}^{d'-v} c(x = 0)[u] \cdot \partial_H(...).$$

For all $v$, $0 \leq v \leq d' - d$:

$$X_{26}(v) = \partial_H \left( \int_{w=d-v}^{d'-v} s(x = 0)[w] \cdot A_0 \parallel B_3 \parallel V_1 \right) +$$

$$\left( \text{true} \right) \rightarrow \int_{w=0}^{d'-v} i_2[w] \cdot X_{26}(w)$$

$$\left( \text{false} \right) \rightarrow \int_{u=a}^{d'-v} c(x = 0)[u] \cdot \partial_H(...) + \left( \text{true} \right) \rightarrow \int_{w=0}^{d'-v} i_2[w] \cdot X_{26}(w)$$

For all $v$, $0 \leq v \leq d' - d$:

$$X_{26}(v) = \partial_H \left( \int_{w=d-v}^{d'-v} s(x = 0)[w] \cdot A_0 \parallel B_3 \parallel V_1 \right) +$$

$$\left( \text{true} \right) \rightarrow \int_{w=0}^{d'-v} i_2[w] \cdot X_{26}(w)$$
\[ c(x = 1)[0] \cdot \partial_H \left( \int_{w = a - v}^{d' - v} s(x = 0) \cdot A_0 \parallel B_0 \parallel V_1 \right) \]

\[ = (\forall x \geq a) :~ \neg c(x = 0)[0] \cdot \partial_H(...) + c(x = 1)[0] \cdot X_{27}(v) \]

\[ = (\text{false}) :~ \neg c(x = 0)[0] \cdot \partial_H(...) + c(x = 1)[0] \cdot X_{27}(v) \]

\[ = c(x = 1)[0] \cdot X_{27}(v) \]

For all \( v, 0 \leq v \leq d' - d \):

\[ X_{27}(v) = \partial_H \left( \int_{w = a - v}^{d' - v} s(x = 0) \cdot A_0 \parallel B_0 \parallel V_1 \right) \]

\[ = \int_{w = a - v}^{d' - v} c(x = 0)[w] \cdot \partial_H(A_0 \parallel B_0 \parallel V_0) \]

\[ = \int_{w = a - v}^{d' - v} c(x = 0)[w] \cdot X_0 \]

\[ = (\text{true}) :~ \neg \int_{w = a - v}^{d' - v} c(x = 0)[w] \cdot X_0 \]

For all \( v, 0 \leq v \leq a' - a \):

\[ X_{28}(v) = \partial_H \left( \int_{w = d - v}^{d' - v} i_1[w] \cdot A_3 \parallel B_2 \parallel V_2 \right) \]

\[ = \int_{w = d - v}^{d' - v} i_1[w] \cdot \partial_H \left( A_3 \parallel \int_{u = d - w}^{d' - w} i_2[u] \cdot B_3 \parallel V_2 \right) + \]

\[ \int_{u = d}^{\min(d' - v, d')} i_2[u] \cdot \partial_H \left( \int_{w = d - u}^{d' - u} i_1[w] \cdot A_3 \parallel B_3 \parallel V_2 \right) \]

\[ = \int_{w = d - v}^{d' - v} i_1[w] \cdot \partial_H \left( A_3 \parallel \int_{u = d - w}^{d' - w} i_2[u] \cdot B_3 \parallel V_2 \right) + \]

\[ \int_{u = d}^{\min(d' - v, d')} i_2[u] \cdot \partial_H \left( \int_{w = d - u}^{d' - u} i_1[w] \cdot A_3 \parallel B_3 \parallel V_2 \right) \]

\[ = \int_{w = d - v}^{d' - v} i_1[w] \cdot X_{29}(w) + \int_{u = d}^{d' - v} i_2[u] \cdot X_{30}(w) \]

\[ = (\text{true}) :~ \neg \int_{w = d - v}^{d' - v} i_1[w] \cdot X_{29}(w) + \]

\[ (\forall v \leq d' - d) :~ \int_{u = d}^{d' - v} i_2[u] \cdot X_{30}(u + v) \]

For all \( v, d - a' + a \leq v \leq d' \):

\[ X_{29}(v) = \partial_H \left( A_3 \parallel \int_{w = d - v}^{d' - v} i_2[w] \cdot B_3 \parallel V_2 \right) \]

\[ = c(x = 1)[0] \cdot \partial_H \left( A_0 \parallel \int_{w = d - v}^{d' - v} i_2[w] \cdot B_3 \parallel V_2 \right) + \]

\[ (\forall v \geq d) :~ i_2[0] \cdot \partial_H(A_3 \parallel B_3 \parallel V_2) \]

\[ = c(x = 1)[0] \cdot X_{40}(v) + (\forall v \geq d) :~ i_2[0] \cdot X_{31} \]
For all \( v, d \leq v \leq d' \):

\[
X_{30}(v) = \partial_H \left( \int_{w=d-v}^{d-v} i_1[w] \cdot A_3 \parallel B_3 \parallel V_2 \right) \\
= (v \geq d) \cdot -i_1[0] \cdot \partial_H(A_3 \parallel B_3 \parallel V_2) + \\
c(x = 2)[0] \cdot \partial_H \left( \int_{w=d-v}^{d-v} i_1[w] \cdot A_3 \parallel B_4 \parallel V_2 \right) \\
= i_1[0] \cdot \partial_H(A_3 \parallel B_3 \parallel V_2) + c(x = 2)[0] \cdot \partial_H(A_3 \parallel B_4 \parallel V_2) \\
= i_1[0] \cdot X_{31} + c(x = 2)[0] \cdot X_{32}(v)
\]

\[
X_{31} = \partial_H(A_3 \parallel B_3 \parallel V_2) \\
= c(x = 2)[0] \cdot \partial_H(A_0 \parallel B_3 \parallel V_2) + c(x = 2)[0] \cdot \partial_H(A_3 \parallel B_4 \parallel V_2) \\
= c(x = 2)[0] \cdot X_5 + c(x = 2)[0] \cdot X_{33}
\]

For all \( v, d \leq v \leq d' \):

\[
X_{32}(v) = \partial_H \left( \int_{w=d-v}^{d-v} i_1[w] \cdot A_3 \parallel B_4 \parallel V_2 \right) \\
= (v \geq d) \cdot -i_1[0] \cdot \partial_H(A_3 \parallel B_4 \parallel V_2) + \\
EnterCS_2[0] \cdot \partial_H \left( \int_{w=d-v}^{d-v} i_1[w] \cdot A_3 \parallel B_5 \parallel V_2 \right) \\
= (v \geq d) \cdot -i_1[0] \cdot X_{33} + EnterCS_2[0] \cdot X_{34}(v) \\
= (true) \cdot -i_1[0] \cdot X_{33} + EnterCS_2[0] \cdot X_{34}(v) \\
= i_1[0] \cdot X_{33} + EnterCS_2[0] \cdot X_{34}(v)
\]

\[
X_{33} = \partial_H(A_3 \parallel B_4 \parallel V_2) \\
= c(x = 2)[0] \cdot \partial_H(A_0 \parallel B_4 \parallel V_2) + EnterCS_2[0] \cdot \partial_H(A_3 \parallel B_5 \parallel V_2) \\
= c(x = 2)[0] \cdot X_6 + EnterCS_2[0] \cdot X_{34}
\]

For all \( v, d \leq v \leq d' \):

\[
X_{34}(v) = \partial_H \left( \int_{w=d-v}^{d-v} i_1[w] \cdot A_3 \parallel B_5 \parallel V_2 \right) \\
= \int_{\text{min}(d' - v, 0)}^{\text{min}(d' - v, w)} i_1[w] \cdot \partial_H(A_3 \parallel B_5 \parallel V_2) + \\
\int_{u=0}^{\text{min}(d' - v, w)} LeaveCS_2[u] \cdot \partial_H \left( \int_{w-d-v-u}^{d-v-u} i_1[w] \cdot A_3 \parallel B_6 \parallel V_2 \right) \\
= \int_{w=d-v}^{d-v} i_1[w] \cdot \partial_H(A_3 \parallel B_5 \parallel V_2) + \\
\int_{u=0}^{d-v} LeaveCS_2[u] \cdot \partial_H \left( \int_{w=d-v-u}^{d-v-u} i_1[w] \cdot A_3 \parallel B_6 \parallel V_2 \right) \\
= \int_{w=d-v}^{d-v} i_1[w] \cdot X_{35} + \int_{u=0}^{d-v} LeaveCS_2[u] \cdot X_{36}(u + v) \\
= (true) \cdot -\int_{w=0}^{d-v} i_1[w] \cdot X_{35} + (true) \cdot -\int_{u=0}^{d-v} LeaveCS_2[u] \cdot X_{36}(u + v)
\]
\[ X_{35} = \partial_H(A_3 \parallel B_5 \parallel V_2) \]
\[ = c(x = 2)[0] \cdot \partial_H(A_0 \parallel B_5 \parallel V_2) + \text{LeaveCS}_2[0] \cdot \partial_H(A_3 \parallel B_6 \parallel V_2) \]
\[ = c(x = 2)[0] \cdot X_7 + \text{LeaveCS}_2[0] \cdot X_{39} \]

For all \( v, d \leq v \leq d' \):
\[ X_{36}(v) = \partial_H \left( \int_{w=x-v}^{d'-v} I_1[w] \cdot A_3 \parallel B_6 \parallel V_2 \right) \]
\[ = \int_{w=x-v}^{d'-v} \left( \min(d' - v, a') \cdot c(x = 0)[u] \cdot \partial_H(\ldots) \right) \]
\[ + \int_{u=a}^{d'-v} c(x = 0)[u] \cdot \partial_H(\ldots) \]
\[ = \int_{w=x-v}^{d'-v} I_1[w] \cdot X_{37}(w) + \int_{u=a}^{d'-v} c(x = 0)[u] \cdot \partial_H(\ldots) \]
\[ = (\text{true}) \implies \int_{w=x-v}^{d'-v} I_1[w] \cdot X_{37}(w) \]
\[ + (v \leq d' - a) \implies \int_{u=a}^{d'-v} c(x = 0)[u] \cdot \partial_H(\ldots) \]
\[ = (\text{true}) \implies \int_{w=x-v}^{d'-v} I_1[w] \cdot X_{37}(w) \]

For all \( v, 0 \leq v \leq d' - d \):
\[ X_{37}(v) = \partial_H \left( \int_{w=x-v}^{d'-v} s(x = 0)[w] \cdot B_0 \parallel V_2 \right) \]
\[ = c(x = 2)[0] \cdot \partial_H \left( A_0 \parallel \int_{w=x-v}^{d'-v} s(x = 0)[w] \cdot B_0 \parallel V_2 \right) + \]
\[ (v \geq a) \implies c(x = 0)[0] \cdot \partial_H(A_3 \parallel B_6 \parallel V_2) \]
\[ = c(x = 2)[0] \cdot X_{38}(v) + (v \geq a) \implies c(x = 0)[0] \cdot \partial_H(\ldots) \]
\[ = c(x = 2)[0] \cdot X_{38}(v) \]

For all \( v, 0 \leq v \leq d' - d \):
\[ X_{38}(v) = \partial_H \left( A_0 \parallel \int_{w=x-v}^{d'-v} s(x = 0)[w] \cdot B_0 \parallel V_0 \right) \]
\[ = \int_{w=x-v}^{d'-v} c(x = 0)[w] \cdot \partial_H(A_0 \parallel B_0 \parallel V_0) \]
\[ = \int_{w=x-v}^{d'-v} c(x = 0)[w] \cdot X_0 \]
\[ (\text{true}) :\text{s} \sum_{v=a}^{a'-v} c(x=0)[w] \cdot X_0 \]

\[ X_{39} = \partial_H(A_3 \parallel B_0 \parallel V_2) \]
\[ = c(x=2)[0] \cdot \partial_H(A_0 \parallel B_0 \parallel V_2) \]
\[ = c(x=2)[0] \cdot X_8 \]

For all \( v, d - a' + a \leq v \leq d' \):

\[ X_{40}(v) = \partial_H \left( A_0 \parallel \sum_{w=d-v}^{d'-v} l_2[w] \cdot B_3 \parallel V_2 \right) \]
\[ = \sum_{w=d-v}^{d'-v} l_2[w] \cdot \partial_H(A_0 \parallel B_3 \parallel V_2) \]
\[ = \sum_{w=d-v}^{d'-v} l_2[w] \cdot X_5 \]
\[ = (\text{true}) :\text{s} \sum_{v=d}^{d'} l_2[w] \cdot X_5 \]

\[ X_{41} = \partial_H(A_2 \parallel B_0 \parallel V_1) \]
\[ = \sum_{v=d}^{d'} l_1[v] \cdot \partial_H(A_3 \parallel B_0 \parallel V_1) \]
\[ = \sum_{v=d}^{d'} l_1[v] \cdot X_{16} \]
\[ = (\text{true}) :\text{s} \sum_{v=d}^{d'} l_1[v] \cdot X_{16} \]

For all \( v, 0 \leq v \leq a' \):

\[ X_{42}(v) = \partial_H \left( \sum_{w=a-v}^{a'-v} s(x=1)[w] \cdot A_2 \parallel B_1 \parallel V_0 \right) \]
\[ = \sum_{w=a-v}^{a'-v} c(x=1)[w] \cdot \partial_H \left( A_2 \parallel \sum_{u=a-w}^{a'-w} s(x=2)[u] \cdot B_2 \parallel V_1 \right) \]
\[ + \sum_{u=a}^{\min(a'-v,a')} c(x=2)[u] \cdot \partial_H \left( \sum_{w=a-v-u}^{a'-v-u} s(x=1)[w] \cdot A_2 \parallel B_2 \parallel V_2 \right) \]
\[ = \sum_{w=a-v}^{a'-v} c(x=1)[w] \cdot \partial_H \left( A_2 \parallel \sum_{u=a-w}^{a'-w} s(x=2)[u] \cdot B_2 \parallel V_1 \right) + \]
\[ \sum_{u=a}^{a'-v} c(x=2)[u] \cdot \partial_H \left( \sum_{w=a-v-u}^{a'-v-u} s(x=1)[w] \cdot A_2 \parallel B_2 \parallel V_2 \right) \]
\[ = \sum_{w=a-v}^{a'-v} c(x=1)[w] \cdot X_{43}(w) + \sum_{u=a}^{a'-v} c(x=2)[u] \cdot X_{44}(u+v) \]
\[ = (\text{true}) :\text{s} \sum_{w=a-v}^{a'-v} c(x=1)[w] \cdot X_{43}(w) + \]
\[ (v \leq a' - a) :\text{s} \sum_{u=a}^{a'-v} c(x=2)[u] \cdot X_{44}(u+v) \]
For all \( v \), \( 0 \leq v \leq a' \):

\[
X_{43}(v) = \partial_H \left( A_2 \parallel \int_{w=a-v}^{a'-v} s(x = 2)[w] \cdot B_2 \parallel V_1 \right)
\]

\[
= \min_{u=d}^{(a',a'-v)} \left( \int_{w=a-v}^{a'-v} i_1[u] \cdot \partial_H \left( A_3 \parallel \int_{w=a-v}^{a'-v} s(x = 2)[w] \cdot B_2 \parallel V_1 \right) + \right)
\]

\[
= \int_{w=a-v}^{a'-v} c(x = 2)[w] \cdot \partial_H \left( A_3 \parallel \int_{u=d-w}^{a'-v} i_1[u] \cdot A_3 \parallel B_2 \parallel V_1 \right)
\]

\[
= \left( \int_{a'-d}^{a'-v} i_1[u] \cdot \partial_H(\ldots) + \int_{w=a-v}^{a'-v} c(x = 2)[w] \cdot X_{48}(w) \right)
\]

\[
= (v \leq a' - d) \rightarrow \int_{w=a-d}^{a-v} c(x = 2)[w] \cdot X_{48}(w)
\]

\[
= (true) \rightarrow \int_{w=a-v}^{a'-v} c(x = 2)[w] \cdot X_{48}(w)
\]

\[
= (false) \rightarrow \int_{a'-d}^{a'-v} i_1[u] \cdot \partial_H(\ldots) + (true) \rightarrow \int_{w=a-v}^{a'-v} c(x = 2)[w] \cdot X_{48}(w)
\]

\[
= (true) \rightarrow \int_{w=a-v}^{a'-v} c(x = 2)[w] \cdot X_{48}(w)
\]

For all \( v \), \( a \leq v \leq a' \):

\[
X_{44}(v) = \partial_H \left( \int_{w=a-v}^{a'-v} s(x = 1)[w] \cdot A_2 \parallel B_2 \parallel V_2 \right)
\]

\[
= \min_{u=d}^{(a',a'-v)} \left( \int_{w=a-v}^{a'-v} c(x = 1)[w] \cdot \partial_H \left( A_2 \parallel \int_{u=d-w}^{a'-v} i_2[u] \cdot B_3 \parallel V_1 \right) + \right)
\]

\[
= \int_{w=a-v}^{a'-v} i_2[u] \cdot \partial_H \left( \int_{w=a-v}^{a'-v} s(x = 1)[u] \cdot A_2 \parallel B_3 \parallel V_2 \right)
\]

\[
= \left( \int_{w=a-v}^{a'-v} c(x = 1)[w] \cdot \partial_H(\ldots) + \int_{u=d}^{a'-v} i_2[u] \cdot \partial_H(\ldots) \right)
\]

\[
= (true) \rightarrow \int_{w=0}^{a'-v} c(x = 1)[w] \cdot X_{45}(w) +
\]

\[
(\nu \leq a' - d) \rightarrow \int_{u=d}^{a'-v} i_2[u] \cdot \partial_H(\ldots)
\]

\[
= (true) \rightarrow \int_{w=0}^{a'-v} c(x = 1)[w] \cdot X_{45}(w) + (false) \rightarrow \int_{u=d}^{a'-v} i_2[u] \cdot \partial_H(\ldots)
\]

\[
= (true) \rightarrow \int_{w=0}^{a'-v} c(x = 1)[w] \cdot X_{45}(w)
\]
For all \( v, 0 \leq v \leq \alpha' - \alpha \):

\[
X_{45}(v) = \partial_H \left( A_2 \parallel \int_{w=d-v}^{d-v} i_2[w] \cdot B_3 \parallel V_1 \right)
\]
\[
= \int_{u=d}^{\min(d',d-v)} i_1[u] \cdot \partial_H \left( A_3 \parallel \int_{w=d-v}^{d-v-u} i_2[w] \cdot B_3 \parallel V_1 \right) + \\
\int_{w=d-v}^{\min(d',d-v)} i_2[w] \cdot \partial_H \left( \int_{u=d-w}^{d'-w} i_1[u] \cdot A_3 \parallel B_3 \parallel V_1 \right) +
\]
\[
= \int_{u=d}^{d'} i_1[u] \cdot \partial_H \left( A_3 \parallel \int_{w=d-v}^{d-v-u} i_2[w] \cdot B_3 \parallel V_1 \right) + \\
\int_{w=d-v}^{d'} i_2[w] \cdot \partial_H \left( \int_{u=d-w}^{d'-w} i_1[u] \cdot A_3 \parallel B_3 \parallel V_1 \right) +
\]
\[
= \int_{u=d}^{d'} i_1[u] \cdot X_{12}(u) + \int_{w=d-v}^{d'} i_2[w] \cdot X_{46}(w)
\]
\[
= (v \leq d' - d) : \int_{u=d}^{d'} i_1[u] \cdot X_{12}(v + u) + (\text{true}) : \int_{w=d-v}^{d'} i_2[w] \cdot X_{46}(w)
\]

For all \( v, d - \alpha' + \alpha \leq v \leq d' \):

\[
X_{46}(v) = \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_3 \parallel V_1 \right)
\]
\[
= c(x = 1)[0] \cdot \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_0 \parallel V_1 \right) + \\
(v \geq d) : \int_{w=d-v}^{d'-v} i_1[w] \cdot \partial_H(A_3 \parallel B_3 \parallel V_1)
\]
\[
= c(x = 1)[0] \cdot X_{47}(v) + (v \geq d) : \int_{w=d-v}^{d'-v} i_1[w] \cdot X_{14}
\]

For all \( v, d - \alpha' + \alpha \leq v \leq d' \):

\[
X_{47}(v) = \partial_H \left( \int_{w=d-v}^{d'} i_1[w] \cdot A_3 \parallel B_0 \parallel V_1 \right)
\]
\[
= \int_{w=d-v}^{d'} i_1[w] \cdot \partial_H(A_3 \parallel B_0 \parallel V_1)
\]
\[
= (\text{true}) : \int_{w=max(0,d-v)}^{d'-v} i_1[w] \cdot X_{16}
\]

For all \( v, 0 \leq v \leq \alpha' \):

\[
X_{48}(v) = \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_2 \parallel V_2 \right)
\]
\[
= \int_{w=d-v}^{\min(d',d-v')} i_1[w] \cdot \partial_H \left( A_3 \parallel \int_{u=d-w}^{d-w} i_2[u] \cdot B_3 \parallel V_2 \right) + \\
\int_{u=d}^{\min(d',d-v')} i_2[u] \cdot \partial_H \left( \int_{w=d-v}^{d'-v} i_1[w] \cdot A_3 \parallel B_3 \parallel V_2 \right) +
\]
\[
= \int_{w=d-v}^{d'-v} i_1[w] \cdot \partial_H \left( A_3 \parallel \int_{u=d-w}^{d-w} i_2[u] \cdot B_3 \parallel V_2 \right) +
\]

279
\[
\int_{u=d}^{d'-v} t_2[u] \cdot \partial_H \left( \int_{w=d-v-u}^{d'-v-u} i_1[w] A_3 \parallel B_3 \parallel V_2 \right) \\
= \int_{w=d-v}^{d'-v} i_1[w] \cdot X_{49}(w) + \int_{u=d}^{d'-v} t_2[u] \cdot X_{30}(u + v) \\
= (\text{true}) \text{ :- } \int_{w=d-v}^{d'-v} i_1[w] \cdot X_{49}(w) + (v \leq d' - d) \text{ :- } \int_{u=d}^{d'-v} t_2[u] \cdot X_{30}(u + v)
\]

For all \( v, d - a' \leq v \leq d' \):

\[
X_{49}(v) = \partial_H \left( A_3 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_2 \right) \\
= c(x=2)[0] \cdot \partial_H \left( A_0 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_2 \right) + \\
(v \geq d) \text{ :- } i_2[0] \cdot \partial_H (A_3 \parallel B_3 \parallel V_2) \\
= c(x=2)[0] \cdot X_{50}(v) + (v \geq d) \text{ :- } i_2[0] \cdot X_{31}
\]

For all \( v, d' - a \leq v \leq d' \):

\[
X_{50}(v) = \partial_H \left( A_0 \parallel \int_{w=d-v}^{d'-v} i_2[w] \cdot B_3 \parallel V_2 \right) \\
= \int_{w=d-v}^{d'-v} i_2[w] \cdot \partial_H (A_0 \parallel B_3 \parallel V_2) \\
= \int_{w=d-v}^{d'-v} i_2[w] \cdot X_5 \\
= (\text{true}) \text{ :- } \int_{w=\max(0,d-v)}^{d'-v} i_2[w] \cdot X_5
\]

Acknowledgments

We would like to thank Martin Abadi, Jos Baeten, Roland Bol, Jozef Hooman, Steven Klu­sener, Hans Mulder, Michel Reniers, Fred Schneider, and an anonymous referee for their various comments, suggestions, and fruitful discussion. Finally, we thank Jan Friso Groote for kindly providing the PASCAL source of his bisimulation-checking tool (which is de­scribed in [Gro91, §3]).

Bibliography


Formal Verification of a Leader Election Protocol in Process Algebra

Lars-åke Fredlund*
SICS, Box 1263, S-164 28 Kista, Sweden
fred@sics.se

Jan Friso Groote  Henri Korver†
Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
{jfg,korver}@phil.ruu.nl

Abstract
In 1982 Dolev, Klawe & Rodeh presented an $O(n \log n)$ unidirectional distributed algorithm for the circular extrema-finding (or leader-election) problem. At the same time Peterson came up with a nearly identical solution. In this paper, we bring the correctness of this algorithm to a completely formal level. This relatively small protocol, which can be described on half a page, requires a rather involved proof for guaranteeing that it behaves well in all possible circumstances. To our knowledge, this is one of the more advanced case-studies in formal verification based on process algebra.

1 Introduction
Experience teaches that distributed protocols are hard to define correctly. This is not only due to the inherent complexity of distributed systems, but it is also caused by the lack of adequate techniques to prove the correctness of such protocols. This means that there are no good ways of validating designs for distributed systems. The current approach to proving correctness of distributed systems generally uses stylized forms of hand waving that does not always avoid the intricacies and pitfalls that often appear in distributed systems. We are convinced that more precise proof techniques need to be used, which should allow for computer based proof checking. Concretely this means that a logic based approach should be taken.

The language $\mu$CRL (micro Common Representation Language) [13] has been defined as a combination of process algebra and (equational) data types to describe and verify distributed systems. In accordance with the philosophy outlined in the first paragraph this is a very precisely defined language provided with a logical proof system [14]. It is primarily intended to verify statements of the form

\[ \text{Condition} \rightarrow \text{Specification} = \text{Implementation}. \]

This system has been applied to verify a number of data transfer and distributed scheduling protocols of considerable complexity [3, 11, 12, 15]. It incorporates several old and new techniques [4, 3]. Due

*Research supported in part by the Swedish Research Council for Engineering Sciences (TFR) (project no. 221-92-722) and the HCM project EXPRESS.
†The investigations were (partly) supported by the Foundation for Computer Science in the Netherlands (SIoN) with financial support from the Netherlands Organization for Scientific Research (NWO).
to the logical nature of the proof system proofs can be verified by computer. Some sizable examples of proofs verified using the proof checker Coq [9] are reported in [14, 16].

If one develops a new technique then it is important that it is validated that the technique meets its purpose. For μCRL this means that it is applied to a wide range of distributed systems. In this paper we show its applicability on Dolev, Klawe and Rodeh's leader election or extrema finding protocol [10] that has been designed for a network with a unidirectional ring topology. At the same time, Peterson published a nearly identical version of this protocol, see [19]. This protocol is efficient, $O(n \log n)$, and highly parallel. As far as we know this is the first leader election protocol verified in a process algebraic style. In [7] a number of leader election protocols for carrier sense networks have been specified and some (informal) proof sketches are given in modal logic.

In Section 2 we specify Dolev, Klawe and Rodeh's leader election protocol formally in μCRL. The protocol is proven correct in Section 3 using a detailed argument. Appendix A summarises the proof theory for μCRL, and Appendix B defines the data types used in the specification and proof of the protocol.

Acknowledgements. We thank Frits Vaandrager for pointing out this protocol to us.

2 Specification and Correctness of the Leader Election protocol

We assume $n$ processes in a ring topology, connected by unbounded queues. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier $\text{id}$ (in the following assumed to be a natural number). The task of an algorithm for solving the leader election problem is then to make sure that eventually exactly one process will announce itself to be the leader.

In Dolev, Klawe and Rodeh's algorithm [10] each process in the ring carries out the following task:

Active:
\[
d := \text{id}
\]
\[
do\forever
\]
\[
\text{send}(d)
\]
\[
\text{receive}(e)
\]
\[
\text{if} \ e = \text{id} \text{ then begin}
\]
\[
\text{announce \_leader}
\]
\[
\text{stop}
\]
\[
\text{end}
\]
\[
\text{send}(e)
\]
\[
\text{receive}(f)
\]
\[
\text{if} \ e > \max(d, f) \text{ then } d := e \text{ else goto Relay}
\]
\[
\text{end}
\]

Relay:
\[
do\forever
\]
\[
\text{receive}(d)
\]
\[
\text{send}(d)
\]
\[
\text{end}
\]

The intuition behind the protocol is as follows. In each round the number of selectable processes decreases, if there are more than two active processes around. During each round every active process, i.e., a process in state Active, receives two different values. If the first value is larger than the second value and its own value, then it stays active. In this case its anti-clockwise neighbour will become a relay process. So, from every set of active neighbours, one will die in every round. Furthermore, the
maximal value among the identifiers will never be lost in the ring network, it will traverse the ring in messages, or be stored in a variable in a process, until only one active process remains. If only one active process is left, i.e., not in state Relay, then the leader-in-spe sends its own value of \( d \) to itself, and then declares itself a leader.

As the attentive reader may have noticed, there is a simpler way to elect a leader. For example, it would be sufficient for a process to receive just one value, i.e., the value \( (e) \) of its direct neighbour. In this case, only two values instead of three values have to be compared (\( e > d \) instead of \( e > \max(d, f) \)). However, this approach is not so efficient as one may need \( 2n^2 + 2n \) actions before a leader is selected. The protocol described earlier is faster. It is bounded by \( 2n \log n + 2n \) actions because in every round at least one process becomes inactive.\(^1\) For an explanation of these complexity bounds one is referred to [10].

Below we formalise the processes and their configuration in the ring as described above in \( \mu \text{CRL} \).

\[
\text{proc \ Active}(i, d, n:\text{Nat}) = \\
\quad s(i, d) \sum_{e: \text{Nat}} r(i-n, 1, e) \{ \text{leader} \delta \triangleq \text{eq}(d, e) \triangleright s(i, e) \\
\quad \sum_{f: \text{Nat}} r(i-n, 1, f) \text{Active}(i, e, n) \triangleq e > \max(d, f) \triangleright \text{Relay}(i, n) \}
\]

Here a process in the imperative description with value ident for \( d \) corresponds to \( \text{Active}(i, \text{ident}, n) \). Intuitively the \( \mu \text{CRL} \) process first sends the value of the variable \( d \) to the next process in the ring \( (s(i, d)) \) via a queue, which is described below. Then it reads a new value \( e \) from the queue connected to the preceding process in the ring by an action \( r(i-n, 1, e) \). The notation \( -n \) stands for subtraction modulo \( n \), which is defined in Appendix B. Consequently, it executes a then-if-else test denoted by \( \triangleright \). The notation \( \triangleright \) stands for subtraction modulo \( n \). If the variables \( d \) and \( e \) are equal, expressed by \( \text{eq}(d, e) \), then the process declares itself leader by executing the action \( \text{leader} \). Otherwise the value of \( e \) is sent \( (s(i, e)) \) and a value \( f \) is read \( (r(i-n, 1, f)) \). Now, if \( e \) is larger than both \( d \) and \( f \) the process repeats itself with \( e \) replacing \( d \). Otherwise, the process becomes a relay process (denoted by \( \text{Relay}(i, n) \)).

In order to prove the correctness of the protocol we must be precise about the behaviour of the queues that connect the processes. We assume that the queues have infinite size and deliver data in a strict first in first out fashion without duplication or loss. In the queue data is stored in a data queue \( q \) which is specified in Appendix B. Note that the behaviour of the queue is straightforward; it reads data via \( r(i, d) \) at process \( i \) and delivers it via \( s(i+n, 1) \) at process \( i+n \) \((+n \) is addition modulo \( n \)).

\[
\text{proc \ Q}(i, n:\text{Nat}, q:\text{Queue}) = \sum_{d: \text{Nat}} r(i, d) Q(i, n, i(d, q)) + \\
\quad s(i+n, 1, ioe(q)) Q(i, n, iioe(q)) \triangleq \text{not empty}(q) \triangleright \delta
\]

It remains to connect all processes together. First we state that send actions \( s \) communicate with receive actions \( r \). Then, using the processes \( \text{Spec} \) and \( \text{Spec} \) we combine the processes with the queues, and assign a unique number to them. The process \( \text{Spec}(n) \) represents a ring network of \( n \) processes interconnected by queues.

\[
\text{act \ c : \ Nat \times \ Nat} \\
\text{com \ rs = c} \\
\text{proc \ Spec}(m, n:\text{Nat}) = \text{Active}(m-1, m-1, n) \parallel Q(m-1, n, q_0) \parallel \text{Spec}(m-1, n) \triangleq m > 0 \triangleright \delta \\
\text{Spec}(n:\text{Nat}) = r(\epsilon) \delta_{(r, s)}(\text{Spec}(n, n))
\]

Since the protocol is supposed to select exactly one leader after some internal negotiation we formulate correctness by the following formula, where ‘\( '=' \) is to be interpreted as ‘behaves the same’:

\(^1\) By \( \log n \), we mean \( \log_2 n \).
Theorem 2.1. For all \( n : \text{Nat} \)
\[ n > 0 \rightarrow \text{Spec}(n) = \tau \text{leader} \delta \]

The theorem says that in a ring with at least one process exactly one leader will be elected after some internal activity.

However, as experience shows the correctness reasoning above is too imprecise to serve as a proof of correctness of the protocol. Many, often rather detailed arguments, are omitted. Actually, the protocol does not have to adhere to the rather synchronous execution suggested by the word 'rounds', but is highly parallel. One can even argue that given the large number of rather 'wild' executions of the protocol, the above description makes little sense. Therefore, we provide in the next sections a completely formalised proof, where we are only interested in establishing correctness of the protocol and not in proving its efficiency.

3 A proof of the protocol

The proof strategy for proving the correctness theorem consists of a number of distinct steps. First in Section 3.1 we define a linear representation of the specification in which the usage of the parallel composition operator in the original specification is replaced by a tabular data structure encoding the states of processes in the network, and actions with guards that check the contents of the data structure. The linearised specification is proven equivalent to the original specification in Lemma 3.3.

Then, in Section 3.3, we define a (focus) condition on the tabular data structure such that if the condition holds then no internal computation is any longer possible in the protocol, i.e., no \( \tau \) steps can be made \([3]\). The focus condition is used in Lemma 3.10, in Section 3.6, to separate the proof that the linear specification can be proven equivalent to a simple process into two parts. Lemma 3.10 together with Lemma 3.3 then immediately proves the correctness theorem of the protocol, i.e., Theorem 2.1. The proof of Lemma 3.10 makes use of the Concrete Invariant Corollary (see Appendix A and \([4]\)), i.e., a number of invariance properties are defined (in Section 3.4) on the tabular data structure such that regardless which execution step the linear specification performs, the properties remain true after the step if they were true before the execution of the step. These invariants are used to prove the equality between the linear specification and the simple process in Lemma 3.10. In order to make use of the Concrete Invariant Corollary we have to show that the linear specification can only perform finitely many consecutive \( \tau \) steps. This is proven in Section 3.5.

3.1 Linearisation

As a first step the leader election protocol is described as a \( \mu \)CRL process in a state based style, as this is far more convenient for proving purposes. The state based style very much resembles the Unity format \([6, 8]\) or the I/O automata format \([17]\). Following \([6]\) we call this format the Unity format or a process specification in Unity style. Inspection of the processes \textit{Active} and \textit{Relay} indicates that there are 7 different major states between the actions. The states in \textit{Active} are numbered 0,1,2,3,6 and those in \textit{Relay} get numbers 4 and 5. The processes \textit{Active} and \textit{Relay} can then be restated as follows:

\[
\text{proc } \text{Act}(i, d, e, n, s: \text{Nat}) = \\
\quad s(i, d) \text{Act}(i, d, e, n, 1) \triangleq \text{eq}(s, 0) \triangleright \delta + \\
\quad \bigvee_{e: \text{Nat}} r(i - n 1, e) \text{Act}(i, d, e, n, 2) \triangleq \text{eq}(s, 1) \triangleright \delta + \\
\quad \text{leader} \text{Act}(i, d, e, n, 6) \triangleq \text{eq}(d, e) \text{ and } \text{eq}(s, 2) \triangleright \delta + \\
\quad s(i, e) \text{Act}(i, d, e, n, 3) \triangleq \text{not eq}(d, e) \text{ and } \text{eq}(s, 2) \triangleright \delta + \\
\quad \bigvee_{f: \text{Nat}} r(i - n 1, f) \text{Act}(i, d, e, n, 0) \triangleq e \triangleright \text{max}(d, f) \text{ and } \text{eq}(s, 3) \triangleright \delta + \\
\quad \bigvee_{f: \text{Nat}} r(i - n 1, f) \text{Act}(i, d, e, n, 4) \triangleq e \leq \text{max}(d, f) \text{ and } \text{eq}(s, 3) \triangleright \delta + \\
\quad s(i, d) \text{Act}(i, d, e, n, 4) \triangleq \text{eq}(s, 3) \triangleright \delta + \\
\]
Lemma 3.1. For all $i, d, e, n$, we have:

$$\text{Active}(i, d, n) = \text{Act}(i, d, e, n, 0)$$

$$\text{Relay}(i, n) = \text{Act}(i, d, e, n, 4)$$

Proof. The proof of this lemma is straightforward, using the Recursive Specification Principle (RSP), but note that it uses $a(p \lor c \lor q) = a p \lor c \lor a q$ as well as the distributivity of $\Sigma$ over $\lor$.

We now put the processes and queues in parallel. As we work towards the Unity style, we must encode the states of the individual processes in a data structure. For this we take a table (or indexed queue) with an entry for each process $i$. This entry contains values for the variables $d, e, s$ and the contents of the queue in which process $i$ is putting its data. Furthermore, it contains a variable of type Bool, which plays a role in the proof. The data structure has the name Table and is defined in Appendix B.

We put the processes and queues together in three stages. First we put all processes together, using $\Pi_{Act}$ and $X_{Act}$ below. Then we put all queues together, via $\Pi_{Q}$ and $X_{Q}$.

Finally, we combine $X_{Ad}$ and $X_{Q}$ obtaining the process $X$ which is a description in Unity style of the leader election protocol.

The leader election protocol in Unity form is given below and will be the core process of the proof.

Note that in many cases verification of a protocol only starts after the process below has been written down. In the description of $X$ most details of the description are directly reflected in corresponding behaviour of the constituents $X_{Act}$ and $X_{Q}$. However, there is one difference. It appears that in the protocol two kinds of messages travel around. One that one could call active and others that one could call passive. The active messages contain numbers that may replace the current value of the $d$-variable of its receiver. The passive messages are not essential for the correctness of the protocol, but only used to improve its speed. For the correctness of the protocol it is important to know that the maximum identification number is always somewhere in an active position and no identification number is present twice in active positions. In order to distinguish active from passive messages, we...
have added a boolean \( b \) to each position in the queues. Where if \( b = t \) the position is active, and if \( b = f \) the position is passive. When processes become Relays then they also act as a queue. Therefore, we have also added a boolean \( b \) to the process parameters, to indicate the status of the message that a process in state 5 is holding. The equation below is referred to by (I) in the remainder of the proof.

\[
\text{proc } X(T: \text{Table}, n: \text{Nat}) =
\begin{align*}
&\sum_{j: \text{Nat}} \tau \left( \text{upd}_{s}(1, j, \text{in}_{q}(\text{get}_{d}(j, T), t, j, T)), n \right) \circ \text{eq}(\text{get}_{s}(j, T), 0) \text{ and } j < n \triangleright \delta + \\
&\sum_{j: \text{Nat}} \tau \left( \text{unq}(j - n, 1, \text{upd}_{s}(\text{toe}(j - n, 1, T), j, \text{upd}_{s}(2, j, T)), n) \right) \circ \text{eq}(\text{get}_{s}(j, T), 1) \text{ and not empty}(j - n, 1, T) \text{ and } j < n \triangleright \delta + \\
&\sum_{j: \text{Nat}} \tau \left( \text{leader}(j, \text{upd}_{s}(6, j, T)) \circ \text{eq}(\text{get}_{s}(j, T), 2) \text{ and eq}(\text{get}_{d}(j, T), \text{get}_{s}(j, T)) \right) \text{ and } j < n \triangleright \delta + \\
&\sum_{j: \text{Nat}} \tau \left( \text{unq}(j - n, 1, \text{upd}_{s}(\text{get}_{v}(j, T), j, \text{upd}_{s}(0, j, T)), n) \right) \circ \text{eq}(\text{get}_{s}(j, T), 3) \text{ and not empty}(j - n, 1, T) \text{ and } j < n \triangleright \delta + \\
&\sum_{j: \text{Nat}} \tau \left( \text{unq}(j - n, 1, \text{upd}_{s}(4, j, \text{upd}_{s}(\text{toe}(j - n, 1, T), T)), n) \right) \circ \text{eq}(\text{get}_{s}(j, T), 4) \text{ and not empty}(j - n, 1, T) \text{ and } j < n \triangleright \delta + \\
&\sum_{j: \text{Nat}} \tau \left( \text{unq}(\text{in}_{q}(\text{get}_{s}(j, T), \text{get}_{s}(j, T), j, \text{upd}_{s}(4, j, T)), n) \circ \text{eq}(\text{get}_{s}(j, T), 5) \right) \text{ and } j < n \triangleright \delta \\
\end{align*}
\]

Definition 3.2. The function \( \text{init} : \text{Nat} \rightarrow \text{Table} \), which is used for denoting the initial state of the protocol, is defined as follows:

\[
\text{init}(n) = \text{if}(\text{eq}(n, 0), t, \text{in}(n - 1, n - 1, 0, 0, f, q_0, \text{init}(n - 1))).
\]

See also Appendix B.5.

Lemma 3.3. For all \( T : \text{Table} \) and \( m, n : \text{Nat} \)

1. \( \text{UniqueIndex}(T) \rightarrow \Pi_{\text{Act}}(T, n) = X_{\text{Act}}(T, n), \)
2. \( \text{UniqueIndex}(T) \rightarrow \Pi_{\text{Q}}(T, n) = X_{\text{Q}}(T, n), \)
3. \( \text{UniqueIndex}(T) \land \text{test}(j, T) = j < n \rightarrow \text{Spec}(T, n) = X(T, n), \)
4. \( \text{Spec'}(m, n) = \Pi_{\text{Act}}(\text{init}(m), n) \parallel \Pi_{\text{Q}}(\text{init}(m), n), \)
5. \( \text{Spec}(n) = \text{Spec}(\text{init}(n), n), \)
6. \( \text{Spec}(n) = X(\text{init}(n), n), \)

Proof.

1. A standard expansion using induction on \( T \) (cf. [16]).
2. Again a straightforward expansion.
3. \( \text{Spec}(T, n) = \tau(c) \partial_{\{r,s\}}(X_{\text{Act}}(T, n) \parallel X_{\text{Q}}(T, n)) \rightarrow \text{Spec}(T, n) \parallel X_{\text{Act}}(T, n) \parallel X_{\text{Q}}(T, n) \). Now expand \( X_{\text{Act}}(T, n) \parallel X_{\text{Q}}(T, n) \) and apply hiding. The equations obtained in this way match those of \( X(T, n) \), except that \( j < n \) is replaced by \( \text{test}(j, T) \)' or \( \text{test}(j, T) \) and \( \text{test}(j - n, 1, T) \). As \( X \) is convergent (proven in Lemma 3.7) it follows with the Concrete Invariant Corollary [4] that \( \text{Spec}(T, n) \) and \( X(T, n) \) are equal. The invariant \( \text{test}(j, T) = j < n \) is used and easy to show true.

290
4. By induction on \( m \), using associativity and commutativity of the merge.

5. Directly from the previous case, i.e. Lemma 3.3.4.

6. Directly using cases 3 and 5.

\[ \square \]

### 3.2 Notation

In the sequel we will for certain property formulas \( \phi(j) \) write

\[ \forall_{j<n}\phi(j) \text{ for } \phi'(0,n) \text{ and } \forall_{i<j<n}\phi(j) \text{ for } \phi'(i+1,n) \]

and

\[ \exists_{j<n}\phi(j) \text{ for } \phi''(0,n) \text{ and } \exists_{i<j<n}\phi(j) \text{ for } \phi''(i+1,n) \]

and

\[ \sum_{j<n}\phi(j) \text{ for } \phi'''(0,n) \]

where \( \phi'(j,n) \), \( \phi''(j,n) \) and \( \phi'''(j,n) \) are defined by:

\[ \phi'(j,n) = \text{if}(j \geq n, t, \phi(j) \text{ and } \phi'(j+1,n)) \]

\[ \phi''(j,n) = \text{if}(j \geq n, f, \phi(j) \text{ or } \phi''(j+1,n)) \]

\[ \phi'''(j,n) = \text{if}(j \geq n, 0, \text{if}(\phi(j), 1, 0) + \phi'''(j+1,n)) \]

Note that if we can prove that

\[ (j < n \text{ and } \phi(j)) \rightarrow \psi(j) \]

then we can also show that

\[ \forall_{j<n}\phi(j) \rightarrow \forall_{j<n}\psi(j) \text{ and } \exists_{j<n}\phi(j) \rightarrow \exists_{j<n}\psi(j) \]

Also note that:

\[ \text{not } (\forall_{j<n}\phi(j)) = \exists_{j<n}\text{not } \phi(j) \text{ and } \text{not } (\exists_{j<n}\phi(j)) = \forall_{j<n}\text{not } \phi(j) \]

### 3.3 Focus Condition

The focus condition \( FC : Table \times Nat \rightarrow Bool \) indicates at which points the leader election protocol cannot do \( \tau \)-steps. This means it can either do nothing, or do a leader action.

\[ FC(T, n) = \]

\[ \forall_{j<n} \text{ not eq}(get_e(j, T), 0) \text{ and } \]

\[ \text{not eq}(get_e(j, T), 1) \text{ or empty}(j - n 1, T) \text{ and } \]

\[ \text{not eq}(get_t(j, T), 2) \text{ or eq}(get_t(j, T), get_e(j, T)) \text{ and } \]

\[ \text{not eq}(get_t(j, T), 3) \text{ or empty}(j - n 1, T) \text{ and } \]

\[ \text{not eq}(get_t(j, T), 4) \text{ or empty}(j - n 1, T) \text{ and } \]

\[ \text{not eq}(get_t(j, T), 5) \]
3.4 Some invariants of $X$

In this section we state four invariants ($Inv_1, \ldots, Inv_4$) of the process $X(T, n)$ that are used in Section 3.6 to prove the correctness of the protocol. We prove that the predicates below are indeed invariance properties in a traditional manner. First we show that they hold in the initial state of the protocol, i.e., for invariant $Inv_i$ we show $Inv_i(init(n), n)$. Then for each protocol step (there are eight such steps in the linearised process $X$) we show that if both the precondition of the step holds and the predicate holds in the state before the protocol step, then the predicate holds also in the state that is the result of performing the step. For example, to prove that $Inv_2$ is an invariance property we need to establish that the first step in $X$ preserves the property, i.e., that

$$eq(get_s(j, T), 0) \text{ and } j < n \text{ and } Inv_2(T, n) \rightarrow Inv_2(upd_s(1, j, ind_t(get_d(j, T), t, j, T)), n)$$

where $T$ is a tabular data structure. This entails proving a large number of rather trivial lemmas, such as:

$$qsizes(upd_s(1, j, T), n) = qsizes(T, n)$$

We omit here the rather long and tedious details of these proofs. In order to establish that $Inv_3$ and $Inv_4$ are indeed invariants we first have to prove additional statements on the behaviour of the protocol, i.e., $Inv_5, Inv_6, Inv_7, Inv_8$ and $Inv_9$ in Sections 3.4.5 to 3.4.9 respectively.

3.4.1 Acceptable states

Each process is in one of the states $0, \ldots, 6$:

$$Inv_1(T, n) = \forall j < n \leq get_s(j, T) \leq 6$$

3.4.2 Bound on the number of messages in queues

Invariant $Inv_2$ expresses the property that the number of processes in state 1 or 3 is equal to the number of processes in state 5 plus the number of messages in message channels.

$$Inv_2(T, n) = eq(nproc(T, 1, n) + nproc(T, 3, n), nproc(T, 5, n) + qsizes(T, n))$$

where

$$nproc(T, s, n) = \sum_{j<n} eq(get_s(j, T), s), 1, 0)$$

$$qsizes(T, n) = \sum_{j<n} size(get_s(j, T))$$

3.4.3 Termination of one process implies termination of all processes

Invariant $Inv_3$ expresses that if a process is in state 6, then all processes are either in state 4 or state 6. It is provable using invariant $Inv_9$.

$$Inv_3(T, n) = (\exists_{j<n} eq(get_s(j, T), 6)) \rightarrow \forall_{j<n} eq(get_s(j, T), 4) \text{ or } eq(get_s(j, T), 6)$$

3.4.4 Max is preserved

In the initial state, $init(n)$, the maximal identifier in the ring is equal to $n - 1$. Invariant $Inv_4$ expresses that this value can not be lost. The invariants $Inv_5, Inv_6, Inv_7$ are needed to establish $Inv_4$.

$$Inv_4(T, n) = \exists_{j<n} ActiveNode(n - 1, j, T) \text{ or } ActiveChan(n - 1, get_q(j, T))$$

where
ActiveNode\((k, j, T) = \)
\((eq(get, (j, T), 0) \text{ and } eq(get_d(j, T), k)) \text{ or } \)
\((eq(get_s(j, T), 2) \text{ or } eq(get(j, T), 3) \text{ or } eq(get_s(j, T), 6)) \text{ and } eq(get(j, T), k)) \text{ or } \)
\((eq(get_s(j, T), 5) \text{ and } get_b(j, T) \text{ and } eq(get_d(j, T), k))\)

ActiveChan\((k, q) = \text{if( empty(q), f, (hd_b(q) and eq(k, hd(q))) or ActiveChan(k, tl(q)))}\)

An identifier has not been lost if it can in the future be received by another process and replace the value of the \(d\) variable of that process. Identifiers can be stored either in a variable, in a process (ActiveNode) or in a channel (ActiveChan).

3.4.5 Trivial facts

Inv_5 formulates two trivial protocol properties, that all identifiers are less than \(n\) (less than or equal to the maximal identifier \(n - 1\)), and that the values of variables \(d\) and \(e\) differs when a process is in state 3.

\(Inv_5(T, n) = \)
\(\forall_{j < n} \text{Bounded}_q(get_e(j, T), n) \text{ and } \)
\(get_d(j, T) < n \text{ and } get_e(j, T) < n \text{ and } \)
\(\text{if(eq(get_s(j, T), 3), not eq(get_d(j, T), get_e(j, T)), t)}\)

where

\(Bounded_q(q, n) = \text{if( empty(q), t, hd(q) < n and Bounded}_q(tl(q), n))\)

3.4.6 Consecutive identifiers are distinct

Inv_6 guarantees that when an identifier in an active position follows an identifier in a passive position, the identifiers are distinct. This invariant depends on Inv_5 and Inv_7.

\(Inv_6(T, n) = \forall_{j < n} \text{Cons}_q(j, T, n)\)

where

\(\text{Cons}_q(j, T, n) = \)
\(\text{Cons}_q(get_e(j, T), j, T, n) \text{ and } \)
\(\text{if(eq(get_s(j, T), 5) and not get_b(j, T), } \)
\(\text{Neq}_q(get_d(j, T), get_e(j, T), j, T, n), \)
\(\text{if(eq(get_s(j, T), 1) or eq(get_s(j, T), 2), Eq}_q(get_d(j, T), get_e(j, T), j, T, n), t)\)}

Cons_q(q, j, T, n) =
\(\text{if( empty(q), t, Cons}_q(tl(q), j, T, n) \text{ and if(hd}(q), t, Neq}_q(hd(q), tl(q), j, T, n)))\)

Neq(q, j, T, n) =
\(((eq(get_s(j, T), 2) \text{ or } eq(get_s(j, T), 3)) \text{ and not eq(get_e(j, T), k)}) \text{ or } \)
\((eq(get_s(j, T), 4) \text{ and Neq}_q(k, get_s(j, T), j, T, n)) \text{ or } \)
\((eq(get_s(j, T), 5) \text{ and get_b(j, T) and not eq(get_d(j, T), k)})\)

Neq_q(k, q, j, T, n) = \text{if( empty(q), Neq}(k, j + n 1, T, n), hd_b(q) and not eq(hd(q), k))

Eq(q, k, j, T, n) =
\(((eq(get_s(j, T), 2) \text{ or } eq(get_s(j, T), 3)) \text{ and eq(get_e(j, T), k)}) \text{ or } \)
\((eq(get_s(j, T), 4) \text{ and Eq}_q(k, get_s(j, T), j, T, n)) \text{ or } \)

Eq(k, j, T, n) =
\(((eq(get_s(j, T), 2) \text{ or } eq(get_s(j, T), 3)) \text{ and eq(get_e(j, T), k)}) \text{ or } \)
\((eq(get_s(j, T), 4) \text{ and Eq}_q(k, get_s(j, T), j, T, n)) \text{ or } \)
\[ \text{Eqq}(k, q, j, T, n) = \frac{i}{\text{empty}(q), \text{Eq}(k, j + n, T, n), \text{hd}(q) \text{ and } \text{eq}(\text{hd}(q), k)} \]

3.4.7 Active and passive messages

The invariant \( \text{Inv}_7 \) characterises the relation between neighbour processes and channel contents.

\[ \text{Inv}_7(T, n) = \forall j < n \text{ Alt}(j, T, n) \]

where

\[ \text{Alt}(j, T, n) = \]
\[ \text{if}(\text{eq}(\text{get}_e(j, T), 0) \text{ or } \text{eq}(\text{get}_e(j, T), 3) \text{ or } \text{eq}(\text{get}_e(j, T), 4) \text{ and not } \text{get}_b(j, T)) \text{ or } \text{eq}(\text{get}_e(j, T), 5) \text{ and } \text{get}_b(j, T)), \]
\[ \text{secondary}(\text{get}_q(j, T), j, T, n), \]
\[ \text{primary}(\text{get}_q(j, T), j, T, n)) \]

\[ \text{primary}(q, j, T, n) = \]
\[ \text{if}(\text{empty}(q), \]
\[ \text{eq}(\text{get}_e(j + n, 1, T), 2) \text{ or } \text{eq}(\text{get}_e(j + n, 1, T), 3) \text{ or } \text{eq}(\text{get}_e(j + n, 1, T), 6) \text{ or } \]
\[ ((\text{eq}(\text{get}_e(j + n, 1, T), 4) \text{ or } \text{eq}(\text{get}_e(j + n, 1, T), 5)) \text{ and } \text{get}_b(j + n, 1, T)), \]
\[ \text{hd}(q) \text{ and } \text{secondary}(\text{tl}(q), j, T, n)) \]

\[ \text{secondary}(q, j, T, n) = \]
\[ \text{if}(\text{empty}(q), \]
\[ \text{eq}(\text{get}_e(j + n, 1, T), 0) \text{ or } \text{eq}(\text{get}_e(j + n, 1, T), 1) \text{ or } \]
\[ ((\text{eq}(\text{get}_e(j + n, 1, T), 4) \text{ or } \text{eq}(\text{get}_e(j + n, 1, T), 5)) \text{ and not } \text{get}_b(j + n, 1, T)), \]
\[ \text{not } \text{hd}(q) \text{ and } \text{primary}(\text{tl}(q), j, T, n)) \]

The intuition for the rather complex looking invariant is to capture the protocol property that there are two kinds of messages sent: active messages which are received by the following process as values on the e variable and which can subsequently replace the d value of the process. The passive messages are received as values on the f variables (state 3) and will not replace the original d value of the process.

The \text{Alt} property guarantees that an active message can never be received as a passive message (or vice versa), i.e., neighbour processes and channels are always kept synchronised by the protocol. \text{Inv}_7 is needed to establish the invariants \text{Inv}_8 and \text{Inv}_4, to guarantee that identifiers are neither duplicated nor lost.

In order to prove \text{Inv}_7 in particular the following two lemmas are useful. For example, \text{Lemma}_71 allows us to conveniently prove that \text{secondary}(\text{get}_q(j, T), j, T) implies \text{secondary}(\text{get}_q(j, T'), j, T), assuming that the channels \text{get}_q(j, T) and \text{get}_q(j, T') are identical.

\[ \text{Lemma}_71(T, T', n) = \]
\[ \forall j < n \text{eq}(\text{get}_q(j, T), \text{get}_q(j, T')) \rightarrow \]
\[ \text{secondary}(\text{get}_q(j, T), j, T, n) \rightarrow \text{secondary}(\text{get}_q(j, T'), j, T', n) \leftarrow \]
\[ \left( \text{even}(\text{size}(\text{get}_q(j, T))) \rightarrow \text{secondary}(q_0, j, T, n) \rightarrow \text{secondary}(q_0, j, T', n) \right) \text{ and } \]
\[ \left( \text{not}(\text{even}(\text{size}(\text{get}_q(j, T)))) \rightarrow \text{primary}(q_0, j, T, n) \rightarrow \text{primary}(q_0, j, T', n) \right) \]

Similarly, \text{Lemma}_72 is convenient for proving that the transition from state 3 to state 4 preserves the invariant:
Lemma 7.2

Lemma 7.2(T, n) =
\[ \forall j < n | \text{Alt}(j, T) \text{ and not empty}(j, T) \text{ and secondary}(get_q(j, T), j, T) \text{ and eq}(get_s(j + n, 1, T), 3) \rightarrow \text{not toe}(j, T) \]

3.4.8 Uniqueness of identifiers

Inv9 expresses the fact that identifiers can occur in at most one "active" position in the ring of processes. It is provable with the help of Inv7.

\[ \text{Inv}_9(T, n) = \forall k < \text{Count}(T, k, n) \leq 1 \]

where

\[ \text{Count}(T, k, n) = \sum_{j < n} \text{if}(\text{ActiveNode}(k, j, T), 1, 0) + \sum_{j < n} \text{ActiveChanOcc}(k, \text{get}_q(j, T)) \]

\[ \text{ActiveChanOcc}(k, q) = \]
\[ \text{if}(\text{empty}(q), 0, \text{if}(hds(q) \text{ and eq}(k, hd(q)), 1, 0) + \text{ActiveChanOcc}(k, tl(q))) \]

Intuitively, the definition of Count counts the number of times an identifier occurs in an "active" position, i.e., in a position such that the identifier can be transmitted and received by another process and later replace the d value of that process. An identifier in an active position can either occur in a variable (ActiveNode) or in a channel (ActiveChanOcc).

3.4.9 Identifier travel creates relay processes

Inv9 points out that if two processes contain the same identifier \( k \) then the processes in between are guaranteed to be in state 4 and the connecting channels all empty. It is provable using Inv8.

\[ \text{Inv}_9(T, n) = \forall k < n | \text{end}(\text{get}_s(i, T), 1) \text{ or end}(\text{get}_s(i, T), 2) \text{ and eq}(\text{get}_d(i, T), k) \rightarrow \]
\[ \forall j < n | \text{eq}(\text{get}_s(j, T), 0) \rightarrow \text{not eq}(\text{get}_d(j, T), k) \text{ and} \]
\[ \forall j < n | \text{ActiveNode}(k, j, T) \rightarrow \text{empty}(\text{get}_q(i, T)) \text{ and EmptyNodes}(i, j, n, T) \text{ and} \]
\[ \forall j < n | \text{ActiveChan}(k, \text{get}_q(j, T)) \rightarrow \]
\[ \text{eq}(hd(j, T), k) \text{ and hds}(j, T) \text{ and} \]
\[ \text{if}(\text{eq}(i, j), 0, \text{eq}(\text{get}_s(j, T), 4) \text{ and empty}(\text{get}_q(i, T)) \text{ and EmptyNodes}(i, j, n, T)) \]

where

\[ \text{EmptyNode}(j, T) = \text{eq}(\text{get}_s(j, T), 4) \text{ and empty}(j, T) \]

\[ \text{EmptyNodes}(i, j, n, T) = \]
\[ \text{if}(i < j, \forall i < j \text{EmptyNode}(i, T), (\forall i < j \text{EmptyNode}(i, T)) \text{ and} \]
\[ (\forall i < n \text{EmptyNode}(i, T)) \]

3.5 Convergence of the protocol

In this section we prove that the linear process \( X \) is convergent, i.e., that we can find a decreasing measure on the data parameter over the \( r \)-steps in the \( X \) process operator. This result implies that all sequences of \( r \)-steps are finite, which is a necessary condition for applying the Concrete Invariant Corollary. We prove that the function Meas defined below is a decreasing measure, and thus proving convergence.

\[ \text{Meas}(T, n) = \]
\[ \sum_{j < n} \left[ \text{if}(\text{eq}(\text{get}_s(j, T), 0), (n - \text{get}_d(j, T) + 2) 6 n^3, \right. \]
\[ \left. \text{if}(\text{eq}(\text{get}_s(j, T), 1) \text{ or eq}(\text{get}_s(j, T), 2), (1 + n - \text{get}_d(j, T)) 6 n^2 + 3 n^3, \right. \]
\[ \left. \text{if}(\text{eq}(\text{get}_s(j, T), 3), (1 + n - \text{get}_d(j, T)) 6 n^3, 0)) \right) \right]\]
\[
\sum_{j<n} \sum_{k<size(getq(j,T))} \text{Term}(j,T,k) + \\
\sum_{j<n} \text{if}(eq(getq(j,T),5), 1 + \text{Term}(j + n, T, size(getq(j,T))), 0).
\]

\[
\text{Term}(j,T,st) = \\
\text{if}(eq(st,1) \text{ and } (eq(getq(j,T),0) \text{ or } eq(getq(j,T),1))) \text{ or } (eq(st,0) \text{ and } getq(j,T) \leq 3], 1, \\
2 + \text{Term}(j + n, T, size(getq(j,T))) + st + \\
\text{if}(eq(getq(j,T),5), 1, 0) - \text{if}(eq(getq(j,T),1,3), 1, 0)).
\]

We have a sequence of theorems that are useful to show that \(Meas(T,n)\) shows that all \(\tau\)-sequences in \(X\) are finite.

**Lemma 3.4.** If \(n > 0\), \(0 \leq j, k < n\) and

\[
st < \sum_{i=j}^{k} \text{if}(eq(getq(i,T),1) \text{ or } eq(getq(i,T),3), 1, 0) - \\
\text{if}(eq(getq(i,T),5), 1, 0) - size(getq(i - n, T))) + size(getq(j - n, T)).
\]

then

1. \(\text{Term}(j,T,st) \leq 2(k - n, j) + 1.\)
2. If \(getq(j,T) = 5\) and \(T' = inq(d,b,j,upd_{(4,j,T)})\) then \(\text{Term}(i,T',st) = \text{Term}(i,T,st).\)
3. If \(getq(j,T) = 4, \text{st} < size(getq(i,T)), i \neq j, T' = untoe(j - n,1,upd_{(5,j,T)})\) then \(\text{Term}(i,T',st) \leq \text{Term}(i,T,st + if(eq(i,j),1,0)).\)

**Proof.** All statements are proven by induction on \((k - n, j).\)

**Corollary 3.5.**

1. For \(n > 0\) and \(0 \leq k < n\) we find \(\text{Term}(k,T,st) < 2n\) provided \(\text{st} < size(getq(k,T)).\)
2. If \(getq(j,T) = 5, T' = inq(d,b,j,upd_{(4,j,T)})\) and \(\text{st} < size(getq(i,T))\) then \(\text{Term}(i,T',st) \leq \text{Term}(i,T,st).\)
3. If \(getq(j,T) = 4, \text{st} < size(getq(i,T)), i \neq j, T' = untoe(j - n,1,upd_{(5,j,T)})\) then \(\text{Term}(i,T',st) \leq \text{Term}(i,T,st).\)

**Proof.** Respectively, instantiate case 1 of Lemma 3.4 with \(j = k + n, 1;\) case 1 with \(k = l - n, 1\) and \(j = l + n, 1;\) case 2 with \(j = k + n, 1\) and at last case 3 with \(j = k + n, 1.\)

**Lemma 3.6.**

\[
\begin{align*}
\text{getq}(j,T) = 0 & \rightarrow Meas(upd_{(1,j,T),n}) + 3n^3 \leq Meas(T,n) \\
\text{getq}(j,T) = 1 & \rightarrow Meas(upd_{(2,j,T),n}) = Meas(T,n) \\
\text{getq}(j,T) = 2 & \rightarrow Meas(upd_{(3,j,T),n}) + 3n^3 \leq Meas(T,n) \\
\text{getq}(j,T) = 3 & \rightarrow Meas(upd_{(4,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 0 & \rightarrow Meas(inq_{(getq(j,T),b,j,T),n}) < Meas(T,n) + 3n^3 \\
\text{getq}(j,T) = 2 & \rightarrow Meas(upd_{(4,j,T),n}) < Meas(T,n) + 3n^3 \\
\text{getq}(j,T) = 1 & \rightarrow Meas(upd_{(4,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 3 & \rightarrow Meas(upd_{(5,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 4 & \rightarrow Meas(upd_{(4,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 5 & \rightarrow Meas(upd_{(4,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 3 & \rightarrow Meas(upd_{(5,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 4 & \rightarrow Meas(upd_{(5,j,T),n}) < Meas(T,n) \\
\text{getq}(j,T) = 5 & \rightarrow Meas(upd_{(5,j,T),n}) < Meas(T,n)
\end{align*}
\]
**Theorem 3.7.** \( X \) is convergent.

**Proof.** This follows as with the help of Lemma 3.6 it is straightforward to see that \( \text{Meas}(T, n) \) is a decreasing measure.

**Remark 3.8.** The measure \( \text{Meas} \) is certainly not optimal. It suggest that the algorithm requires about \( 6n^3(n + 2) \) actions to select a leader. This is a very rough measure, looking at the far sharper bound in [10] suggests that the bound can actually be improved to \( 4n \log_2 n + 2n \) actions.

### 3.6 Final calculations

We now prove the following crucial lemma that links the leader action to \( X \). But first we provide an auxiliary function that expresses that no process \( j < n \) is in state 6.

**Definition 3.9.**

\[
\text{nonsix}(T, n) = \forall j < n \text{not} eq(get_j(T), 6).
\]

**Lemma 3.10.** The invariants \( \text{Inv}_1(T, n), \ldots, \text{Inv}_4(T, n) \) imply:

\[
X(T, n) = \text{(leader} \delta \land \text{nonsix}(T, n) \land \delta) \land FC(T, n) \land \tau (\text{leader} \delta \land \text{nonsix}(T, n) \land \delta).
\]

**Proof.** We show assuming the invariants \( \text{Inv}_1(T, n), \ldots, \text{Inv}_4(T, n) \) that

\[
\forall T, n : \text{Nat.} (\text{leader} \delta \land \text{nonsix}(T, n) \land \delta) \land FC(T, n) \land \tau (\text{leader} \delta \land \text{nonsix}(T, n) \land \delta)
\]

is a solution for \( X \) in (I). As (I) is convergent, the lemma follows from the Concrete Invariant Corollary (see [4]). First suppose \( FC(T, n) \) holds. This means that we must show that

\[
\text{leader} \delta \land \text{nonsix}(T, n) \land \delta = \sum_{j \in \text{Nat.}} \text{leader} \delta \land \text{nonsix} \text{upd}_1(6, j, T) \land \delta \land \text{eq}(get_j(T), 2) \land \text{eq}(getd_j(T), get_j(T)) \land j < n \land \delta.
\]

Note that it follows from \( FC(T, n) \) that the other summands of (I) may be omitted. As \( \text{nonsix} \text{upd}_1(6, j, T) \) f. equation (1) reduces to:

\[
\text{leader} \delta \land \text{nonsix}(T, n) \land \delta = \sum_{j \in \text{Nat.}} \text{leader} \delta \land \text{eq}(get_j(T), 2) \land \text{eq}(getd_j(T), get_j(T)) \land j < n \land \delta.
\]

Now assume \( \text{nonsix}(T, n) \). From \( FC(T, n) \) and \( \text{Inv}_1(T, n) \) is follows that

\[
\forall j < n 1 \leq get_j(T) \leq 4.
\]

First we show that \( \exists j < n \text{eq}(get_j(T), 2) \) and \( \text{eq}(getd_j(T), get_j(T)) \). Now suppose

\[
\exists j < n \text{eq}(get_j(T), 1) \) or \( \text{eq}(getd_j(T), 3).
\]

Hence, using \( \text{Inv}_2(T, n) \) and \( n\text{proc}(T, 1, n) + n\text{proc}(T, 3, n) > 0 \) and (3), it follows that \( \text{qsize}(T, n) > 0 \). Hence, \( \exists j < n \text{size}(j - n, 1, T) > 0 \). Hence, using the focus condition and \( \text{Inv}_1(T, n) \):

\[
\exists j < n \text{eq}(get_j(T), 2) \) and \( \text{eq}(getd_j(T), get_j(T)) \).
\]

Now suppose

\[
\exists j < n \text{eq}(get_j(T), 1) \) or \( \text{eq}(getd_j(T), 3).
\]

Hence, using (3) it follows that

\[
\forall j < n \text{eq}(get_j(T), 2) \) or \( \text{eq}(getd_j(T), 4).
\]
Now assume
\[ \forall j \in \mathbb{N} \neg eq(get_s(j, T), 4). \]
But this contradicts Inv\(_4\)(T, n) in conjunction with Inv\(_2\)(T, n). Hence, using (4) it follows that
\[ \exists j \in \mathbb{N} eq(get_s(j, T), 2). \]
From this and FC(T, n) it follows that
\[ \exists j \in \mathbb{N} eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)). \]
Hence, using SUM3 (see appendix) the right-hand side of (2) has a summand
\[ \text{leader}\delta. \]
(5)
But using some straightforward calculations (5) has the right-hand side of (2) as a summand. Hence, if nonsix(T, n) then (2) is equivalent to
\[ \text{leader}\delta = \text{leader}\delta \]
which is clearly a tautology. Now assume not nonsix(T, n). Hence, \[ \exists j \leq n \neg eq(get_s(j, T), 6). \]
Using Inv\(_3\)(T, n) it follows that
\[ \text{leader}\delta \triangleleft \text{leader}\delta \]
or\[ eq(get_s(j, T), 6). \]
Hence (2) reduces to
\[ \delta = \delta \]
which is clearly true. Now suppose the focus condition does not hold, i.e., not FC(T, n). We find (where we use that \( n > 0 \) and Milner’s second r-law (T2)):
\[ \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \triangleleft j < n \triangleright \delta +
\sum_{j: \mathbb{N}} \text{leader}\delta \\
\triangleleft \text{nonsix}(T, n) \text{ and } eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta \]
(6)
Now note that it follows from Inv\(_3\)(T, n) that if \( \exists j \leq n \neg eq(get_s(j, T), 2) \), then nonsix(T, n). So, (6) reduces to:
\[ \sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =
\sum_{j: \mathbb{N}} \tau (\text{leader}\delta \triangleleft \text{nonsix}(T, n) \triangleright \delta) \]
\[ \sum_{j: \mathbb{N}} \text{leader}\delta \triangleleft eq(get_s(j, T), 2) \text{ and } eq(get_d(j, T), get_e(j, T)) \text{ and } j < n \triangleright \delta =\]
Finally we are ready to prove that the main theorem of the paper holds, i.e.,

$$n > 0 \rightarrow \text{Spec}(n) = \tau\text{leader} \delta$$

**Proof.** Using Lemma 3.3 we know

$$\text{Spec}(n) = X(\text{init}(n), n).$$

From Lemma 3.10 it then follows that

$$\text{Spec}(n) = (\text{leader} \delta \triangleleft \text{nonsiz}(\text{init}(n), n) \triangleright \delta) \uplus \text{FC}(\text{init}(n), n) \uplus \tau(\text{leader} \delta \triangleleft \text{nonsiz}(\text{init}(n), n) \triangleright \delta).$$

However, $\text{FC}(\text{init}(n), n)$ is not true if $n > 0$ while $\text{nonsiz}(\text{init}(n), n)$ is true. Therefore

$$n > 0 \rightarrow \text{Spec}(n) = \tau\text{leader} \delta$$

is true. \qed

## 4 Conclusion

We have outlined a formal proof of the correctness of the leader election or extrema finding protocol of Dolev, Klawe and Rodeh in $\mu$CRL. The proof is now ready to be proof checked conform [2, 12, 16, 20].

It is shown that process algebra, in particular $\mu$CRL, is suited to prove correctness of non-trivial protocols. A drawback of the current verification is that it is rather complex and lengthy. A possible lead towards improvement is given by Frits Vaandrager in [21], where by using the notion of confluence (see e.g. [18]) one only needs to consider one trace to establish correctness. Currently we are formalising this notion in [5]. We expect that using this idea our proof can be simplified significantly.
A. An overview of the proof theory for $\mu$CRL

We provide here a very short account of the axioms that have been used. We also give the Concrete Invariant Corollary for referencing purposes.

All the process algebra axioms used to prove the leader election protocol can be found in Table 1-6. We do not explain the axioms (see [1, 4, 14]) but only include them to give an exact and complete overview of the axioms that we used. Actually, the renaming axioms are superfluous, but have been included for completeness.

Besides the axioms we have used the Concrete Invariant Corollary [4] that says that if two processes $p$ and $q$ can be shown a solution of a well founded recursive specification using an invariant, then $p$ and $q$ are equal, for all starting states where the invariant holds. It is convenient to use linear process operators, which are functions that transform a parameterised process into another parameterised process. If such an operator is well founded, it has a unique solution, and henceforth defines a process. Note that if a linear process operator is applied to a process name, it becomes a process in Unity format.

Definition A.1. A linear process operator $\Psi$ is an expression of the form

$$\lambda p:D \rightarrow \mathcal{P}. \lambda d:D. \Sigma_{i \in I} \Sigma_{e_i,D_i} f_i(d,e_i) \cdot p(g_i(d,e_i)) \cdot b_i(d,e_i) \cdot \delta$$

for some finite index sets $I, I'$, actions $c_i, c_i'$, data types $D_i, D_i', D_j$, and $D_j'$, functions $f_i : D_i \rightarrow D_i', b_i : D_i \rightarrow \text{bool}$, $g_i : D_i \rightarrow D_i'$, $b_i'(d,e_i) = b_i(d,e_i) \cdot \delta$.

Definition A.2. A linear process operator (LPO) $\Psi$ written in the form above is called convergent iff there is a well-founded ordering $<$ on $D$ such that $g_i(d,e_i) < d$ for all $d \in D_i, i \in I$ and $e_i \in D_i$ with $c_i = \tau$ and $b_i(d,e_i)$.

Corollary A.3 (Concrete Invariant Corollary). Assume

$$\Phi = \lambda p:D \rightarrow \mathcal{P}. \lambda d:D. \Sigma_{j \in J} \Sigma_{e_j,D_j} f_j(d,e_j) \cdot p(g_j(d,e_j)) \cdot b_j(d,e_j) \cdot \delta$$

is a LPO. If for some predicate $I : D \rightarrow \text{bool}$

$$\lambda pd. \Phi pd \cdot I(d) \cdot \delta$$

is convergent, and

$$I(d) \land b_j(d,e_j) \rightarrow I(g_j(d,e_j))$$

for all $j \in J$, $d \in D$ and $e_j \in D_j$,

i.e. $I$ is an invariant of $\Phi$, and for some $q : D \rightarrow \mathcal{P}, q' : D \rightarrow \mathcal{P}$ we have

$$I(d) \rightarrow q(d) = \Phi qd,$$

$$I(d) \rightarrow q'(d) = \Phi q'd,$$

then

$$I(d) \rightarrow q(d) = q'(d).$$
Table 1: The axioms of ACP in $\mu$CRL.

$\text{CM1} \quad x \parallel y = z \parallel y + x \parallel z$  
$\text{CM2} \quad a \parallel x = a \cdot x$ 
$\text{CM3} \quad a \cdot x \parallel y = a \cdot (x \parallel y)$ 
$\text{CM4} \quad (x + y) \parallel z = x \parallel z + y \parallel z$ 
$\text{CM5} \quad a \cdot x \parallel b = (a \parallel b) \cdot x$ 
$\text{CM6} \quad a \parallel b \cdot x = (a \parallel b) \cdot x$ 
$\text{CM7} \quad a \cdot x \parallel b = (a \parallel b) \cdot (x \parallel y)$ 
$\text{CM8} \quad (x + y) \parallel z = x \parallel z + y \parallel z$ 
$\text{CM9} \quad x \parallel (y + z) = x \parallel y + x \parallel z$

$\text{CD1} \quad \delta \parallel x = \delta$ 
$\text{CD2} \quad x \parallel \delta = \delta$ 
$\text{CT1} \quad \tau \parallel x = \delta$ 
$\text{CT2} \quad x \parallel \tau = \delta$

Table 2: Axioms of Standard Concurrency (SC).

$\text{TID} \quad \tau_I(\delta) = \delta$ 
$\text{TIT} \quad \tau_I(\tau) = \tau$ 
$\text{TII} \quad \tau_I(n(\tilde{t})) = n(\tilde{t})$ if $n \notin I$ 
$\text{TII} \quad \tau_I(n(\tilde{t})) = \tau$ if $n \in I$ 
$\text{TI3} \quad \tau_I(x + y) = \tau_I(x) + \tau_I(y)$ 
$\text{TI4} \quad \tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$

Table 3: Axioms for abstraction.
\[ SUM1 \quad \sum_{d:D}(p) = p \quad \text{if } d \text{ not free in } p \]
\[ SUM2 \quad \sum_{d:D}(p) = \sum_{e:D}(p[e/d]) \quad \text{if } e \text{ not free in } p \]
\[ SUM3 \quad \sum_{d:D}(p) = \sum_{d:D}(p) + p \]
\[ SUM4 \quad \sum_{d:D}(p_1 + p_2) = \sum_{d:D}(p_1) + \sum_{d:D}(p_2) \]
\[ SUM5 \quad \sum_{d:D}(p_1 \cdot p_2) = \sum_{d:D}(p_1) \cdot p_2 \quad \text{if } d \text{ not free in } p_2 \]
\[ SUM6 \quad \sum_{d:D}(p_1 \perp p_2) = \sum_{d:D}(p_1) \perp p_2 \quad \text{if } d \text{ not free in } p_2 \]
\[ SUM7 \quad \sum_{d:D}(p_1 | p_2) = \sum_{d:D}(p_1) | p_2 \quad \text{if } d \text{ not free in } p_2 \]
\[ SUM8 \quad \sum_{d:D}(p \cdot t) = \cdot t(\sum_{d:D}(p)) \]
\[ SUM9 \quad \sum_{d:D}(p \cdot t) = \sum_{d:D}(p t) \quad \text{if } d \text{ not free in } p \]
\[ SUM10 \quad \frac{p_1 = p_2}{\sum_{d:D}(p_1) = \sum_{d:D}(p_2)} \quad \text{provided } d \text{ not free in } \mathcal{D} \]

Table 4: Axioms for summation.

\[ COND1 \quad x \triangleright y = x \]
\[ COND2 \quad x \triangleright y = y \]
\[ BOOL1 \quad \neg(t = f) \]
\[ BOOL2 \quad \neg(b = t) \rightarrow b = f \]

Table 5: Axioms for the conditional construct and Bool.

\[ T1 \quad x \tau = x \]
\[ T2 \quad \tau x = \tau x + x \]

Table 6: Some \( \tau \)-laws.
B Data types

B.1 Booleans

sort Bool
cons t, f :→ Bool
func not : Bool → Bool
    and, or, eq : Bool × Bool → Bool
var b : Bool
rew not t = f
    not f = t
    t and b = b
    f and b = f
    t or b = t
    for b = b
    eq(t, t) = t
    eq(f, f) = t
    eq(t, f) = f
    eq(f, t) = f

For every sort S we assume a selector function if:

func if : Bool × S × S → S
var s, t : S
rew if(t, s, t) = s
    if(f, s, t) = t

B.2 Natural numbers

sort Nat
cons 0 := Nat
    S : Nat → Nat
func P : Nat → Nat
    even : Nat → Bool
    +, ÷, *, max : Nat × Nat → Nat
    eq, ≥, ≤, <, > : Nat × Nat → Bool
var n, m : Nat
rew P(0) = 0
    P(Sn) = n
    even(0) = t
    even(S(0)) = f
    even(S(S(n))) = even(n)
    n + 0 = n
    n + Sm = S(n + m)
    n ÷ 0 = n
    n ÷ Sm = P(n ÷ m)
    n * 0 = 0
    n * Sm = n + (n * m)
    max(n, m) = if(n ≥ m, n, m)
    eq(0, 0) = t
eq(0, Sn) = f
eq(Sn, 0) = f
eq(Sn, Sm) = eq(n, m)
n ≥ 0 = t
0 ≥ Sn = f
Sn ≥ Sm = n ≥ m
n ≤ m = m ≥ n
n > m = n ≥ Sm
n < m = Sn ≤ m

We write 1 for S(0), 2 for S(S(0)), etc, in the standard manner.

B.3 Modulo arithmetic

func mod : Nat x Nat → Nat
  +, - : Nat x Nat x Nat → Nat
var k, m, n : Nat
rew m mod 0 = m
m mod Sn = if(m ≥ Sn, (m - Sn) mod Sn, m)
k +n, m = (k + m) mod n
k -n, m = if(k mod n ≥ m mod n, k mod n - m mod n, n ~ (m mod n - k mod n))

B.4 Queues

We use two kind of queues which are subtly different. The first is of sort Queue with the usual operations. The second is of sort Queueb which is similar to Queue except that a boolean is added for technical purposes. The specification of Queueb is given below. We do not present the data type Queue here because it can be considered as a simple instance of Queueb as follows: omit the functions hd_b, toe_b and remove all boolean arguments. For example, in : Nat x Bool x Queueb → Queueb corresponds with in : Nat x Queue → Queue.

sort Queueb
cons q0 : Queueb
  in : Nat x Bool x Queueb → Queueb
func rem : Nat x Queueb → Queueb
  il, union : Queueb → Queueb
  con : Queueb x Queueb → Queueb
  hd, toe : Queueb → Nat
  hd_b : Queueb → Bool
  toe_b : Queueb → Bool
  eq : Queueb x Queueb → Bool
  empty : Queueb → Bool
  test : Nat x Queueb → Bool
  size : Queueb → Nat
var d, e : Nat
b, c : Bool
q, r : Queueb
rew rem(d, q0) = q0
rem(d, in(e, b, q)) = if(eq(d, e), q, in(e, b, rem(d, q)))
tl(q₀) = q₀

tl(in(d, b, q₀)) = q

untoe(q₀) = q₀

untoe(in(d, b, q₀)) = q₀

untoe(in(d, b, in(e, c, q₀))) = in(d, b, untoe(in(e, c, q₀)))

con(q₀, q) = q

con(in(d, b, q), r) = in(d, b, con(q, r))

hd(q₀) = 0

hd(in(d, b, q)) = d

hdb(q₀) = false

hdb(in(d, b, q)) = b

toe(q₀) = 0

toe(in(d, b, q₀)) = d


toe(in(d, b, in(e, c, q₀))) = toe(in(e, c, q₀))

hdb(q₀) = false

hdb(in(d, b, q₀)) = b


toe(in(d, b, in(e, c, q₀))) = toe(in(e, c, q₀))

eq(q₀, q₀) = t

eq(q₀, in(d, b, q₀)) = f

eq(in(d, b, q₀), q₀) = f

eq(in(d, b, q₀), in(e, c, r)) = eq(d, e) and eq(b, c) and eq(q, r)

empty(q₀) = eq(size(q₀), 0)

test(d, q₀) = f

test(d, in(e, b, q₀)) = or(eq(d, e), test(d, q₀))

size(q₀) = 0

size(in(d, b, q₀)) = size(q₀) + 1

B.5 Protocol states

sort Table

cons t₀ : Table

    in : Nat × Nat × Nat × Nat × Bool × Queue × Table → Table

func init : Nat → Table

    get₀ : Nat × Table → Nat

    get₁ : Nat × Table → Bool

    get₂ : Nat × Table → Queue

    upd₀, upd₁, upd₂ : Nat × Nat × Table → Table

    upd₂ : Bool × Nat × Table → Table

    upd₃ : Queue × Nat × Table → Table

    test : Nat × Table → Bool

    in₀ : Nat × Nat × Table → Table

    hd : Nat × Table → Nat

    hdb : Nat × Table → Bool

    hd₁ : Table → Nat

    toe : Nat × Table → Nat

    toe₀ : Nat × Table → Bool

    untoe : Nat × Table → Table

empty : Nat × Table → Bool

tl : Table → Table

305
rem : Nat × Table → Table
UniqueIndex : Table → Bool

var d, e, s, v, i, j, n : Nat
T : Table
b, b' : Bool
q, q' : Queueb

rew init(n) = if(eq(n, 0), t₀, in(n - 1, n - 1, 0, 0, f, q₀, init(n - 1)))
getₐ(i, t₀) = 0
getₐ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), d, getₐ(i, T))
getₑ(i, t₀) = 0
getₑ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), e, getₑ(i, T))
getₛ(i, t₀) = 0
getₛ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), s, getₛ(i, T))

getₒ(i, to) = 0
getₒ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), e, getₒ(i, T))
getₒ(i, to) = 0
getₒ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), s, getₒ(i, T))

getₐ(i, to) = f
getₐ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), b, getₐ(i, T))

getₐ(i, t₀) = q₀
getₐ(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), q, getₐ(i, T))

updₐ(v, i, t₀) = in(i, v, 0, 0, 0, f, q₀, t₀)
updₐ(v, i, in(j, d, e, s, b, q, T)) = if(eq(i, j), in(i, v, e, s, b, q, T), in(j, d, e, s, b, q, updₐ(v, i, T)))

updₑ(v, i, t₀) = in(i, 0, 0, 0, f, q₀, t₀)

updₑ(v, i, in(j, d, e, s, b, q, T)) = if(eq(i, j), in(i, 0, 0, 0, f, q₀, t₀), in(j, d, e, s, b, q, updₑ(v, i, T)))

updₛ(v, i, t₀) = in(i, 0, 0, 0, 0, f, q₀, t₀)

updₛ(v, i, in(j, d, e, s, b, q, T)) = if(eq(i, j), in(j, d, e, s, b, q, T), in(j, d, e, s, b, q, updₛ(v, i, T)))

updₒ(b', i, t₀) = in(i, 0, 0, 0, 0, 0, b', q₀, t₀)
updₒ(b', i, in(j, d, e, s, b, q, T)) = if(eq(i, j), in(j, d, e, s, b', q, T), in(j, d, e, s, b, q, updₒ(b', i, T)))

updₒ(q', i, t₀) = in(i, 0, 0, 0, 0, 0, f, q', t₀)
updₒ(q', i, in(j, d, e, s, b, q, T)) = if(eq(i, j), in(j, d, e, s, b, q', T), in(j, d, e, s, b, q, updₒ(q', i, T)))

test(i, t₀) = t

test(i, in(j, d, e, s, b, q, T)) = eq(i, j) or test(i, T)

untoe(i, T) = updₒ(untoe(getₒ(i, T)), i, T)
hd(i, T) = hdl(getₒ(i, T))
hdₒ(i, T) = hdₒ(getₒ(i, T))
hdₒ(t₀) = 0
hdₒ(in(j, d, e, s, b, q, T)) = j
toe(i, T) = toe(getₒ(i, T))
toeₒ(i, T) = toeₒ(getₒ(i, T))

untoe(i, T) = updₒ(untoe(getₒ(i, T)), i, T)
empty(i, T) = empty(getₒ(i, T))
dl(t₀) = t₀
dl(in(j, d, e, s, b, q, T)) = T
rem(i, t₀) = t₀
rem(i, in(j, d, e, s, b, q, T)) = if(eq(i, j), T, in(j, d, e, s, b, q, rem(i, T)))
UniqueIndex(t₀) = t

UniqueIndex(in(j, d, e, s, b, q, T)) = not test(j, T) and UniqueIndex(T)

306
References


Process Algebra Semantics of $\varphi$SDL

J.A. Bergstra$^{1,3}$ C.A. Middelburg$^{2,3}$

$^1$Programming Research Group, University of Amsterdam
P.O. Box 41882, 1009 DB Amsterdam, The Netherlands

$^2$Department of Network & Service Control, KPN Research
P.O. Box 421, 2260 AK Leidschendam, The Netherlands

$^3$Department of Philosophy, Utrecht University
P.O. Box 80126, 3508 TC Utrecht, The Netherlands

E-mail: janb@fw.iuva.nl - keesm@phil.ruu.nl

Abstract

A new semantics of an interesting subset of the specification language SDL is given by a translation to a discrete-time variant of process algebra in the form of ACP extended with data as in $\mu$CRL. The strength of the chosen subset, called $\varphi$SDL, is its close connection with full SDL, despite its dramatically reduced size. Thus, we are able to concentrate on solving the basic semantic issues without being in danger of having to turn the results inside out in order to deal with full SDL. Novel to the presented semantics is that it relates the time used with timer setting to the time involved in waiting for signals and delay of signals.

Keywords & Phrases: discrete time, process algebra, semantics, specification language, asynchronous communication, delay, timers, ACP, SDL.


---

*This paper is to be presented at the 2nd Workshop on Algebra of Communicating Processes, May 1995.
1 Introduction

A process algebra semantics of $\varphi$SDL is presented. $\varphi$SDL is roughly a subset of Basic SDL. The following simplifications have been made:

- blocks are removed and consequently channels and signal routes are merged – making channel to route connections obsolete;
- variables are treated more liberal: all variables are revealed and they can be viewed freely;
- timer setting is regarded as just a special use of signals;
- timer setting is based on discrete time.

Besides, $\varphi$SDL does not deal with the specification of abstract data types. An algebraic specification of all data types used in an $\varphi$SDL specification is assumed as well as an initial algebra semantics for it. The pre-defined data types Boolean and Natural, with the obvious interpretation, should be included; and besides, Pld and Time should be included as copies of Natural.

We decided to focus in $\varphi$SDL on the behavioural aspects of SDL. We did so for the following two reasons. Firstly, the structural aspects of SDL are mostly of a static nature and therefore not very relevant from a semantic point of view. Secondly, the part of SDL that deals with the specification of abstract data types is well understood – besides, it can easily be isolated and treated as a parameter. Because it will largely be a routine matter, we also chose to postpone the addition of procedures, syntypes with a range condition and process types with a bound on the number of instances that may exist simultaneously. For similar reasons, the any expression is omitted. Services are not supported by $\varphi$SDL for other reasons: the semantics of services is hard to understand, ETSI forbids for this reason their use in European telecommunication standards (see [19]), and the SDL community currently discusses its usefulness (see [16]).

Apart from the data type definitions, all SDL system definitions without usage of procedures, services, syntypes with a range condition, process types with a bound on the number of instances that may exist simultaneously, and the any expression can be transformed to $\varphi$SDL system definitions. The transformation concerned has, apart from some minor adaptations, already been given. The first part of the transformation is the mapping for the shorthand notations of SDL which is given informally in the ITU/TS Recommendation Z.100 [21] and defined in a fully precise manner in its Annex F.2 [23]. The second and final part is essentially the mapping extract-dict which is defined in its Annex F.3 [24]; $\varphi$SDL system definitions can actually be viewed as textual presentations of the

---

1 This subset is called $\varphi$SDL, where $\varphi$ stands for flat, as it does not cover the structural aspects of SDL. Throughout the paper, we will write SDL for the version of SDL defined in [21], the ITU/TS Recommendation Z.100 published in 1992.

2 The following is also worth noticing: (1) ETSI discourages the use of abstract data types other than the pre-defined ones in European telecommunication standards (see [19]); (2) ASN.1 [20] is widely used for data type specification in the telecommunications field, and there is an emerging ITU/TS Recommendation, Z.105, for combining SDL and ASN.1 (see [25]).
extracted \textit{Entity-dicts} which are interpreted instead of the SDL system definitions proper.

The semantics of \(\varphi\text{SDL}\) agrees with the semantics of SDL as far as reasonably possible. This means in the first place that obvious errors in [24] have not been taken over. For example, the intended effect of SDL’s create and output actions may sometimes be reached with interruption according to [24] – allowing amongst other things that a process ceases to exist while a signal is sent to it without any delay. Secondly, the way of dealing with time is considered to be unnecessarily complex and inadequate in SDL and has been adapted as explained below.

In SDL, \textbf{Time} and \textbf{Duration}, the pre-defined sorts of absolute time and relative time, are both copies of the pre-defined sort \textbf{Real} (intended to stand for the real numbers, but in fact standing for the rational numbers, see [22]). When a timer is set, a real expiration time must be given. However, the time considered is the system time which proceeds actually in a discrete manner: the system receives ticks from the environment which increase the system time with a certain amount (how much real time they represent is left open). Therefore, the timer is considered to expire when the system receives the first tick that indicates that its expiration time has passed. So nothing is lost by adopting in \(\varphi\text{SDL}\) a discrete time approach, using copies of \textbf{Natural} for \textbf{Time} and \textbf{Duration}, where the time unit can be viewed as the time between two ticks but does not really rely upon the environment. This much simpler approach also allows us to remove the original inadequacy to relate the time used with timer setting to the time involved in waiting for signals by processes and in delay of signals in channels.

We had to make our own choices with respect to time in \(\varphi\text{SDL}\), because the time related aspects of SDL are virtually left out completely in the ITU/TS recommendation Z.100. Our choices were based on communications with various practitioners from the telecommunications field using SDL. In particular, the communications with Leonard Pruitt [18] provided convincing practical justification for the premise of our choices: provided time is divided into sufficiently large time slices, an SDL process will only enter a next time slice if there are no more signals to consume for it in the current time slice. Ease of adaptation to other viewpoints on time in SDL is guaranteed relatively well by using a discrete-time variant of process algebra, essentially \(\text{ACP}_{dt}\) (see [2]), as the basis of the presented semantics.

The language \(\varphi\text{SDL}\) and the presented semantics for it are primarily intended for work on advanced analysis tools for systems modelled using SDL. However, it can also serve to gain a better insight into the semantic aspects of proposed simplifications, and other future changes, of SDL.

The structure of this paper is as follows. First of all, we give an overview of \(\varphi\text{SDL}\) (Section 2). Next, we give a brief summary of the ingredients of process algebra which make up the basis for the semantics of \(\varphi\text{SDL}\) presented in this paper (Section 3). Then, we describe specifics on the operator used to formalize execution of a process in a state (Section 4). After that, we present the process algebra semantics of \(\varphi\text{SDL}\) (Section 5). Finally, we make some additional remarks about the work reported on in this paper as well as some remarks about
related work (Section 6). Besides, there are appendices about notational conventions used (Appendix A) and details about the contexts used to model scope in the presented semantics (Appendix B).

2 Overview of \( \varphi \)SDL

This section gives an overview of \( \varphi \)SDL. Its syntax is described by means of production rules in the form of an extended BNF grammar (the extensions are explained in Appendix A). The meaning of the language constructs of the various forms distinguished by these production rules is explained informally. Some peculiar details, inherited from full SDL, are left out to improve the comprehensibility of the overview. These details will, however, be made mention of in Section 5, where a process algebra semantics of \( \varphi \)SDL is presented.

2.1 System definition

First of all, the \( \varphi \)SDL view of a system is explained in broad outline.

Basically, a system consists of processes which communicate with each other and the environment by sending and receiving signals via signal routes. A process proceeds in parallel with the other processes in the system and communicates with these processes in an asynchronous manner. This means that a process sending a signal does not wait until the receiving process consumes it, but it proceeds immediately. A process may also use local variables for storage of values. A variable is associated with a value that may change by assigning a new value to it. A variable can only be assigned new values by the process to which it is local, but it may be viewed by other processes. Processes can be distinguished by unique addresses, called pid values (process identification values), which they get with their creation.

A signal can be sent from the environment to a process, from a process to the environment or from a process to a process. A signal may carry values to be passed from the sender to the receiver; on consumption of the signal, these values are assigned to local variables of the receiver. A signal route is a unidirectional connection between the processes of two types, or between the processes of one type and the environment, for conveying signals. A signal route may contain a channel.\(^3\) Signals that must pass through a channel are delayed, but signals always leave a channel in the order in which they have entered it. A signal route is a communication path for sending signals, with or without a delay, from the environment to a process, from one process to another process or from a process to the environment. If a signal is sent to a process via a signal route that does not contain a channel, it can be instantaneously delivered to that process. Otherwise there can be an arbitrary delay. A channel may be contained in more than one signal route.

\(^3\)The original channels have been merged with signal routes, but the term channel is reused in \( \varphi \)SDL (see also Section 2.4).
**Syntax:**

```plaintext
<system definition> ::= system <system nm> ; {<definition>}+ endsystem ;
<definition> ::=
dcl <variable nm> <sort nm> ;
signal <signal nm> [(( <sort nm> {, <sort nm> })*)] ;
channel <channel nm> ;
signalroute <signalroute nm>
  from {<process nm> | env} to {<process nm> | env}
  with <signal nm> {, <signal nm>}*[ delayed by <channel nm> ] ;
process <process nm> ( <natural ground expr> ) ;
  [ fpar <variable nm> {, <variable nm>}* ; ]
start ; <transition> {<state def>}*
endprocess ;
```

A system definition consists of definitions of the types of processes present in the system, the local variables used by the processes for storage of values, the types of signals used by the processes for communication, the signal routes via which the signals are conveyed and the channels contained in signal routes to delay signals.

A variable definition `dcl v T` defines a variable `v` that may be assigned values of sort `T`.

A signal definition `signal s(T_1, \ldots, T_n)` defines a type of signals `s` of which the instances carry values of the sorts found in `T_1, \ldots, T_n`. If `(T_1, \ldots, T_n)` is absent, the signals of type `s` do not carry any value.

A channel definition `channel c` defines a channel that delays signals that pass through it.

A signal route definition `signalroute r from X_1 to X_2 with s_1, \ldots, s_n` defines a signal route `r` that delivers without a delay signals sent by processes of type `X_1` to processes of type `X_2`, for signals of types found in `s_1, \ldots, s_n`. The process types `X_1` and `X_2` are called the sender type of `r` and the receiver type of `r`, respectively. A signal route from the environment can be defined by replacing `from X_1` by `from env`. A signal route to the environment can be defined analogously. A signal route delivering signals with an arbitrary delay can be defined by adding `delayed by c`, where `c` is the channel causing the delay.

A process definition `process X(k) ; fpar v_1, \ldots, v_m ; start ; tr d_1 \ldots d_n endprocess ;` defines a type of processes `X` of which `k` instances will be created during the start-up of the system. On creation of a process of type `X` after the start-up, the creating process passes values to it which are assigned to the local variables found in `v_1, \ldots, v_m`. If `fpar v_1, \ldots, v_m` is absent, no values are passed on creation. The process body `start ; tr d_1, \ldots, d_n` describes the behaviour of the processes of type `X` in terms of states and transitions (see further Section 2.2). Each process will start by making the transition `tr`, called its start transition, to enter one of its states. The state definitions found in `d_1 \ldots d_n` define all the states in which the process may come while it proceeds.
2.2 Process behaviours

First of all, the πSDL view of a process is briefly explained.

To begin with, a process is either in a state or making a transition to another state. Besides, when a signal arrives at a process, it is put into the unique input queue associated with the process until it is consumed by the process. The states of a process are the points in its behaviour where a signal may be consumed. However, a state may have signals that have to be saved, i.e. withhold from being consumed in that state. The signal consumed in a state of a process is the first one in its input queue that has not to be saved for that state. If there is no signal to consume, the process waits until there is a signal to consume. So if a process is in a state, it is either waiting to consume a signal or consuming a signal.

A transition from a state of a process is initiated by the consumption of a signal, unless it is a spontaneous transition. The start transition is not initiated by the consumption of a signal either. A transition is made by performing certain actions: signals may be sent, variables may be assigned new values, new processes may be created and timers may be set and reset. A transition may at some stage also take one of a number of branches, but it will eventually come to an end and bring the process to a next state or to its termination.

A timer can be set which sends at its expiration a signal to the process setting it. A timer is identified with the type and carried values of the signal it sends on expiration. Thus an active timer can be set to a new time or reset; if this is done between the sending of the signal noticing expiration and its consumption, the signal is removed from the input queue concerned. A timer is de-activated when it is reset or the signal it sends on expiration is consumed.

Syntax:

\[
\begin{align*}
\langle \text{state def} \rangle & \ ::= \ \\
& \quad \text{state} \ \langle \text{state nm} \rangle ; \\
& \quad \quad \text{save} \ \langle \text{signal nm} \rangle \{, \langle \text{signal nm} \rangle \}^* ; \{\langle \text{transition alt} \rangle \}^* \\
\langle \text{transition alt} \rangle & \ ::= \\
& \quad \langle \text{input guard} \rangle \ | \ \text{input none} ; \langle \text{transition} \rangle \\
\langle \text{input guard} \rangle & \ ::= \\
& \quad \text{input} \ \langle \text{signal nm} \rangle \{ (\langle \text{variable nm} \rangle \{, \langle \text{variable nm} \rangle \}^*) \} ; \\
\langle \text{transition} \rangle & \ ::= \\
& \quad \{ \langle \text{action} \rangle \}^* \{ \langle \text{nextstate} \ \langle \text{state nm} \rangle \ | \ \text{stop} ; \langle \text{decision} \rangle \} ; \\
\langle \text{action} \rangle & \ ::= \\
& \quad \text{output} \ \langle \text{signal nm} \rangle \{ (\langle \text{expr} \rangle \{, \langle \text{expr} \rangle \}^*) \} \\
& \quad \quad \{ \text{to} \ \langle \text{pid expr} \rangle \ | \ \text{via} \ \langle \text{signalroute nm} \rangle \{, \langle \text{signalroute nm} \rangle \}^* \} ; \\
& \quad \quad \{ \text{set} \ \langle \text{time expr} \rangle , \langle \text{signal nm} \rangle \{ (\langle \text{expr} \rangle \{, \langle \text{expr} \rangle \}^*) \} \} ; \\
& \quad \quad \{ \text{reset} \ \langle \text{signal nm} \rangle \{ (\langle \text{expr} \rangle \{, \langle \text{expr} \rangle \}^*) \} \} ; \\
& \quad \quad \{ \text{task} \ \langle \text{variable nm} \rangle \ ::= \ \langle \text{expr} \rangle ; \\
& \quad \quad \{ \text{create} \ \langle \text{process nm} \rangle \{ (\langle \text{expr} \rangle \{, \langle \text{expr} \rangle \}^*) \} \} ; 
\end{align*}
\]
A state definition \texttt{state st; save s_1, \ldots, s_m; alt_1 \ldots alt_n} defines a state \texttt{st} in which certain signals may be consumed and subsequently certain transitions must be made. The signals of the types found in \texttt{s_1, \ldots, s_m} are saved for the state. Each input guard occurring in \texttt{alt_1 \ldots alt_n} gives a type of signals that may be consumed in the state; the corresponding transition is the one that is initiated on consumption of a signal of that type. The transitions with \texttt{input none}; instead of an input guard are the spontaneous transitions that may be made from the state. No signals are saved for the state if \texttt{save s_1, \ldots, s_m} is absent.

An input guard \texttt{input s(v_1, \ldots, v_n)}; may consume a signal of type \texttt{s} and, on consumption, it assigns the carried values to the variables found in \texttt{v_1, \ldots, v_n}. If the signals of type \texttt{s} carry no value, \texttt{(v_1, \ldots, v_n)} is left out.

A transition \texttt{a_1 \ldots a_n nextstate st;} performs the actions found in \texttt{a_1 \ldots a_n} in sequential order and ends with entering the state \texttt{st}. Replacing \texttt{nextstate st} by the keyword \texttt{stop} yields a transition ending with process termination. Replacing it by the decision \texttt{dec} leads instead to transfer of control to one of two or more transition branches.

An output action \texttt{output s(e_1, \ldots, e_n) to e via r_1, \ldots, r_m;} sends a signal of type \texttt{s} carrying the current values of the expressions in \texttt{e_1, \ldots, e_n} to the process with the current \texttt{(pid)} value of the expression \texttt{e} as its address, via one of the usable signal routes found in \texttt{via r_1, \ldots, r_m}. If the signals of type \texttt{s} carry no value, \texttt{(e_1, \ldots, e_n)} is left out. If \texttt{to e} is absent, the signal is sent via one of the signal routes found in \texttt{via r_1, \ldots, r_m} to an arbitrary process of its receiver type. The output action is called an output action with explicit addressing if \texttt{to e} is present. Otherwise, it is called an output action with implicit addressing.

A set action \texttt{set (e, s(e_1, \ldots, e_n));} sets a timer that expires, unless it is set again or reset, at the current (time) value of the expression \texttt{e} with sending a signal of type \texttt{s} that carries the current values of the expressions in \texttt{e_1, \ldots, e_n}.

A reset action \texttt{reset (s(e_1, \ldots, e_n));} de-activates the timer identified with the signal type \texttt{s} and the current values of the expressions in \texttt{e_1, \ldots, e_n}.

An assignment task action \texttt{task v := e;} assigns the current value of the expression \texttt{e} to the local variable \texttt{v}.

A create action \texttt{create X(e_1, \ldots, e_n);} creates a process of type \texttt{X} and passes the current values of the expressions in \texttt{e_1, \ldots, e_n} to the newly created process. If no values are passed on creation of processes of type \texttt{X}, \texttt{(e_1, \ldots, e_n)} is left out.

A decision \texttt{decision e_1; tr_1 \ldots (e_n; tr_n) enddecision} transfers control to the transition branch \texttt{tr_i} \texttt{(1 \leq i \leq n)} for which the value of the expression \texttt{e_i} equals the current value of the expression \texttt{e}. Non-existence and non-uniqueness of such a branch result in an error. A non-deterministic choice can be obtained by replacing the expression \texttt{e} by the keyword \texttt{any} and removing all the expressions \texttt{e_i}.
2.3 Values

The value of expressions in pSDL may vary according to the last values assigned to variables, including local variables of other processes. It may also depend on the system state, e.g. on timers being active or the system time.

Syntax:

\[
\text{<expr> ::=}
\]

\[
\text{<operator \text{ nm}> } [ ( \text{<expr> } \{ \text{<expr>} \}^* ) ]
\]

\[
\text{if <boolean \text{ expr}> then <expr> else <expr> fi}
\]

\[
\text{<variable \text{ nm}>}
\]

\[
\text{view( <variable \text{ nm}> , <pid \text{ expr}> )}
\]

\[
\text{active( <signal \text{ nm}> [ ( <\text{expr}> \{ , <\text{expr}> \}^* ) ] )}
\]

\[
\text{now| self| parent| offspring| sender}
\]

An operator application \text{op(\text{e}_1, \ldots, \text{e}_n)} evaluates to the value yielded by applying the operation \text{op} to the current values of the expressions in \text{e}_1, \ldots, \text{e}_n.

A conditional expression \text{if \text{e}_1\text{ then }\text{e}_2\text{ else }\text{e}_3\text{ fi} evaluates to the current value of the expression \text{e}_2 if the current (Boolean) value of the expression \text{e}_1 is true, and the current value of the expression \text{e}_3 otherwise.

A variable access \text{v} evaluates to the current value of the local variable \text{v} of the process evaluating the expression.

A view expression \text{view(\text{v}, \text{e}) evaluates to the current value of the local variable \text{v} of the process with the current (pid) value of the expression \text{e as its address.}}

An active expression \text{active(\text{s(\text{e}_1, \ldots, \text{e}_n)}) evaluates to the Boolean value true if the timer identified with the signal type \text{s and the current values of the expressions in \text{e}_1, \ldots, \text{e}_n is currently active, and false otherwise.}

The expression \text{now evaluates to the current system time.}

The expressions \text{self, parent, offspring and sender evaluate to the pid values of the process evaluating the expression, the process by which it was created, the last process created by it, and the sender of the last signal consumed by it.}

2.4 Differences with SDL

Syntactically, pSDL is not exactly a subset of SDL. The syntactic differences are as follows:

- variable definitions occur at the system level instead of inside process definitions;
- signal route definitions and process definitions occur at the system level instead of inside block definitions;
- channel paths in channel definitions are absent;
- the option \text{delayed by \text{c} in signal route definitions is new;}
- formal parameters in process definitions are variable names instead of pairs of variable names and sort names;
- signal names are used as timer names.
These differences are all due to the simplifications mentioned in Section 1.
Recall that channels and signal routes have been merged. Because the resulting communication paths connect processes with one another or with the environment, like the original signal routes, we chose to call them signal routes as well. However, the new signal routes may have delaying parts which are reminiscent of the original channels. Therefore, we chose to reuse their name for these delaying parts.

3 Process algebra preliminaries

This section gives a brief summary of the ingredients of process algebra which make up the basis for the semantics of $\varphi$SDL presented in Section 5. We will suppose that the reader is familiar with them. Appropriate references to the literature are included.

We will make use of the Algebra of Communicating Processes (ACP),\textsuperscript{4} introduced in [8], extended with the silent step $\tau$ and the abstraction operator $\tau_1$ for abstraction. Semantically, we adopt the approach to abstraction, originally proposed for ACP in [9], which is based on weak bisimulation due to Milner [15]. For a systematic introduction to ACP, the reader is referred to [5].

Further we will use the following extensions:

state operator We will use the state operator $\lambda_S$, added to ACP in [1]. This operator formalizes execution of a process in a state. Basic is the execution of actions: the action $a'$ that occurs as the result of executing an action $a$ in a state $S$, and the state $S'$ that results when executing $a$ in $S$. This leads to defining equations of the form $\lambda_S(a \cdot P) = a' \cdot \lambda_S(P)$.

process creation We will also use the process creation mechanism, added to ACP in [6]. The process creation operator $E_\phi$ introduced there allows, given a mapping $\phi$ from process names to process expressions, the use of actions of the form $cr(X)$ to create processes $\phi(X)$. The most crucial equation from the defining equations of this operator is $E_\phi(cr(X) \cdot P) = cr(X) \cdot E_\phi(\phi(X) \parallel P)$. Note that the process creation operator leaves a trace of actions of the form $cr(X)$.

conditionals Besides, we will use the one-armed conditional operator $:\rightarrow$ as in [3]. The expression $b :\rightarrow P$, is to be read as if $b$ then $P$; it can only be performed if $b \neq \text{false}$. It is often referred to as a guarded command.

iteration We will also use the binary version of Kleene's star operator $*$, added to ACP in [7], with the defining equation $P * Q = P \cdot (P * Q) + Q$. The behaviour of $P * Q$ is zero or more repetitions of $P$ followed by $Q$.

discrete time We need a relative time version of discrete time process algebra in the form of ACP. We will use the extension of ACP that can be found

\textsuperscript{4}We will actually use ACP without communication, also known as PA$_\alpha$. 

in [2], which is quite similar to ATP [17]. Here we briefly survey discrete
time processes in an informal way.

Time is divided into slices indexed by natural numbers. These time slices
represent time intervals of a length which corresponds to the time unit
used. If the current time is \( t \), \( t \in \mathbb{R}_{\geq 0} \), the current time slice is the time
interval \( \left[ \lfloor t \rfloor, \lfloor t \rfloor + 1 \right) \) (where \( \lfloor t \rfloor \) denotes the floor of \( t \)). We will use the
constants \( \frac{a}{b} \) (one for each action \( a \)) and \( \Delta \) as well as the delay operator \( \sigma_{\text{rel}} \).
\( \frac{a}{b} \) is \( a \) performed within the current time slice and \( \sigma_{\text{rel}}(P) \) is \( P \) delayed till
the next time slice. In a parallel composition \( P_1 \parallel \ldots \parallel P_n \) the transition
to the next time slice is a simultaneous transition of each of the \( P_i \)s. For
example, \( \frac{a}{b} \parallel \sigma_{\text{rel}}(b) \) will never perform \( b \) because \( \frac{a}{b} \) can neither be delayed
nor performed, so \( \frac{a}{b} \parallel \sigma_{\text{rel}}(b) = \frac{a}{b} \). However, \( \frac{a}{b} \parallel \sigma_{\text{rel}}(b) = a \cdot \sigma_{\text{rel}}(b) \).

**summation over data domains** We will in addition use actions parametrized
by data and summation over a data domain as in \( \mu \text{CRL} \) [13, 14]. The
notation \( a(t_1, \ldots, t_n) \), where the \( t_i \)s denote data values, is used for instances
of parametrized actions. In \( \sum_{x:D} P \), the scope of the variable \( x \) is exactly
\( P \). The behaviour of \( \sum_{x:D} P \) is a choice between the instances of \( P \) for the
different values that \( x \) can take, i.e. the values from the data domain \( D \).

The above-mentioned extensions of ACP with a state operator and a process
creation mechanism are also presented in [5]. In ACP with abstraction, the
operators \( \lambda_S \) and \( E_\varphi \) can be defined.

We will also use some abbreviations. Let \( (P_i)_{i \in I} \) be a indexed set of process
expressions where \( I = \{i_1, \ldots, i_n\} \). Then, we write:

\[
\sum_{i \in I} P_i \quad \text{for} \quad P_{i_1} + \ldots + P_{i_n}
\]

\[
\parallel_{i \in I} P_i \quad \text{for} \quad P_{i_1} \parallel \ldots \parallel P_{i_n}
\]

Let \( P \) be a process expression and let \( n \in \mathbb{N} \). Then, we write:

\[
\parallel^n P \quad \text{for} \quad \underbrace{P \parallel \ldots \parallel P}_{n \times}
\]

If conditionals are present, the definition of the state operator needs in addi-
tion an evaluation function \( \text{eval}_S \). The additional equation is \( \lambda_S(b \rightarrow P) = \text{eval}_S(b) \rightarrow \lambda_S(P) \). Thus, execution of \( P \) is disabled in state \( S \) if \( b \) evaluates to
false in \( S \). The state operator used for the semantics of \( \varphi \text{SDL} \) is a slight adap-
tation of the state operator described in [1], due to the highly state dependent
nature of the SDL mechanisms for storage, communication, timing and process
creation. Actions parametrized by domains that are built on expressions denot-
ing values, in contrast with values, are needed as state transforming actions. The
reason for this is that, in general, the values concerned depend on the state in
which the actions are executed. Consequently, the equations become somewhat
more involved than suggested above, as is witnessed by Section 4. The evaluation
function \( \text{eval}_S \) is also needed in these equations.

\[\text{In [4], a revision of [2], a different notation for } \frac{a}{b} \text{ is used, viz. } \text{cts}(a).\]
The process creation operator used for the semantics of $\varphi$SDL is a slight adaptation of the process creation operator described in [6], due to the following details of the process creation mechanism of SDL:

- formal parameters are local variables and parameter passing amounts to assigning initial values to local variables of a newly created process when its execution starts;
- the pid value of the creating process is passed to a newly created process when its execution starts.

Consequently, the process creation action needs, in addition to the name of a process type, parameters to be used by the state operator described in Section 4. So the defining equations have to be reformulated. This is, however, trivial because these additional parameters of the process creation action are ignored by the process creation operator. For example, the most crucial equation becomes

$$E_{\sigma}(cr(X,(v_1,\ldots,v_n),(u_1,\ldots,u_n),i) \cdot P) =$$
$$E_{\sigma}(cr(X,(v_1,\ldots,v_n),(u_1,\ldots,u_n),i) \cdot E_{\phi}(\phi(X) \parallel P)$$

where $v_1,\ldots,v_n$ are the formal parameters and $u_1,\ldots,u_n$ are the corresponding actual parameters.

### 4 Processes with states

The input guards, the SDL actions and the terminator stop constitute the SDL mechanisms for storage, communication, timing and process creation. In the process algebra semantics of $\varphi$SDL, which will be presented in Section 5, the state operator mentioned in Section 3 is used to describe these mechanisms in whole or in part. This means that input guards, SDL actions and stop correspond to ACP actions that interact with a global state. In this section, we will describe the state space, the actions that transform states, and the result of executing processes, built up from these actions, in a state from this state space.

#### 4.1 Preliminaries

We mentioned before that $\varphi$SDL does not deal with the specification of abstract data types. We assume a fixed algebraic specification covering all data types used and an initial algebra semantics, denoted by $\mathcal{A}$, for it. We will write $\text{Sort}_\mathcal{A}$ and $\text{Op}_\mathcal{A}$ for the set of all sort names and the set of all operation names, respectively, in the signature of $\mathcal{A}$. We will write $U$ for $\bigcup_{T \in \text{sort}(\mathcal{A})} T^\mathcal{A}$, where $T^\mathcal{A}$ is the interpretation of the sort name $T$ in $\mathcal{A}$. We will assume that $\text{nil} \notin U$. In the sequel, we will use for each $op \in \text{Op}_\mathcal{A}$ an extension to $U$, also denoted by $op$, such that $op(t_1,\ldots,t_n) = \text{nil}$ if at least one the $t_i$s is not of the appropriate sort. Thus, we can change over from the many-sorted case to the one-sorted case for the description of the meaning of $\varphi$SDL constructs. We can do so without loss of generality, because it can (and should) be statically checked that only terms of appropriate sorts occur.
Uncustomary notation concerning sets, functions and sequences, used in this section, is explained in Appendix A.

4.2 Basic domains and functions, the state space

The state space, used to describe the meaning of system definitions, depends upon the specific variables, types of signals, channels and types of processes introduced in the system definition concerned. They largely make up the contextual information extracted from the system definition by means of the function $\{\bullet\}$ defined in Appendix B. For convenience, we define these state space parameters for arbitrary contexts $\kappa$ (the notation concerning contexts introduced in Appendix B is used):

$$V_\kappa = \text{vars}(\kappa)$$
$$S_\kappa = \text{sigs}(\kappa)$$
$$C_\kappa = \text{chans}(\kappa)$$
$$P_\kappa = \text{procs}(\kappa)$$

First, we define the set $\text{Sig}_{\kappa}$ of signals and the set $\text{ExtSig}_{\kappa}$ of extended signals, which fit into the picture of the communication mechanism. A signal consist of the name of its type and the sequence of values that it carries. An extended signal contains, in addition to a signal, the pid values of its sender and receiver. The pid value of the sender is needed seeing that the identity of the sender may otherwise get lost; a delivered signal need not be consumed immediately, but may be put into an input queue instead. In case a signal must pass through a channel, the pid value of the receiver is also essential because of the possible loss of identity due to queueing or delaying.

$$\text{Sig}_{\kappa} = S_\kappa \times U^*$$
$$\text{ExtSig}_{\kappa} = \text{Sig}_{\kappa} \times \mathbb{N} \times \mathbb{N}$$

We write $\text{snm}(\text{sig})$ and $\text{vals}(\text{sig})$, where $\text{sig} = (s, vs) \in \text{Sig}_{\kappa}$, for $s$ and $vs$, respectively. We write $\text{sig}(\text{esig})$, where $\text{esig} = (\text{sig}, i, i') \in \text{ExtSig}_{\kappa}$, for $\text{sig}$.

The local state of a process includes a storage which associates local variables with the values assigned to them, an input queue where delivered signals are kept until they are consumed, and a component keeping track of the expiration times of active timers. We define the set $\text{Stg}_{\kappa}$ of storages, the set $\text{InpQ}_{\kappa}$ of input queues and the set $\text{Timers}_{\kappa}$ of timers as follows:

$$\text{Stg}_{\kappa} = \bigcup_{V \subseteq V_\kappa} (V \overset{f_{\kappa}}{\to} U)$$
$$\text{InpQ}_{\kappa} = \text{ExtSig}_{\kappa}^*$$
$$\text{Timers}_{\kappa} = \bigcup_{T \subseteq S_{\kappa}} (T \overset{f_{\kappa}}{\to} \mathbb{N} \cup \{\text{nil}\})$$

We will follow the convention that the domain of a function from $\text{Stg}_{\kappa}$ does not contain variables with which no value is associated because a value has never been assigned to them. Consequently, the absence of a value need not to be represented by nil. We will also follow the convention that the domain of a function from $\text{Timers}_{\kappa}$ contains precisely the active timers. While an expired timer is still
active, its former expiration time will be replaced by nil. The basic operations on \( Stg_\kappa \) and \( Timers_\kappa \) are general operations on functions: function application, overriding (\( \oplus \)) and domain subtraction (\( \setminus \)). Overriding and domain subtraction are defined in Appendix A. In so far as the communication mechanism of SDL is concerned, the basic operations on \( InpQ_\kappa \) are the functions

\[
\begin{align*}
getnxt &: InpQ_\kappa \times P_{\bar{P}}(\S_\kappa) \to ExtSig_\kappa \cup \{\text{nil}\}, \\
rmvfirst &: InpQ_\kappa \times Sig_\kappa \to InpQ_\kappa, \\
merge &: P_{\bar{P}}(InpQ_\kappa) \to P_{\bar{P}}(InpQ_\kappa)
\end{align*}
\]

defined below. The value of \( \text{getnxt}(\sigma, ss) \) is the first (extended) signal in \( \sigma \) that is of a type different from the ones in \( ss \). The value of \( \text{rmvfirst}(\sigma, \text{sig}) \) is the input queue \( \sigma \) from which the first occurrence of the signal \( \text{sig} \) has been removed. Both functions are used to describe the consumption of signals by SDL processes. The function \( \text{getnxt} \) is recursively defined by

\[
\begin{align*}
\text{getnxt}((), ss) &= \text{nil} \\
\text{getnxt}((\text{sig}, i, i') \& \sigma, ss) &= (\text{sig}, i, i') \quad \text{if } \text{snm}(\text{sig}) \not\in ss \\
\text{getnxt}((\text{sig}, i, i') \& \sigma, ss) &= \text{getnxt}(\sigma, ss) \quad \text{if } \text{snm}(\text{sig}) \in ss
\end{align*}
\]

and the function \( \text{rmvfirst} \) is recursively defined by

\[
\begin{align*}
\text{rmvfirst}((), \text{sig}) &= () \\
\text{rmvfirst}((\text{sig}, i, i') \& \sigma, \text{sig}) &= \sigma \\
\text{rmvfirst}((\text{sig}, i, i') \& \sigma, \text{sig}') &= (\text{sig}, i, i') \& \text{rmvfirst}(\sigma, \text{sig}') \quad \text{if } \text{sig} \neq \text{sig}'
\end{align*}
\]

For each process, sequences of signals coming from different channels as well as signals noticing timer expiration have to be merged when time progresses to the next time slice. The function \( \text{merge} \) is used to describe this precisely. It is inductively defined by

\[
\begin{align*}
\sigma \in \text{merge}(\{\sigma\}) \\
() \in \text{merge}(\{(), ()\}) \\
\sigma \in \text{merge}(\{\sigma_1, \sigma_2\}) \Rightarrow (\text{sig}, i, i') \& \sigma \in \text{merge}(\{(\text{sig}, i, i') \& \sigma_1, \sigma_2\}) \\
\sigma \in \text{merge}(\{\sigma_1, \sigma_2\}) \land \sigma_2 \in \text{merge}(\Sigma) \Rightarrow \sigma \in \text{merge}(\{\sigma_1 \cup \Sigma\})
\end{align*}
\]

We define now the set \( L_\kappa \) of local states. The local state of a process contains, in addition to the above-mentioned components, the name of its type. Thus, the type of the process concerned will not get lost. This is important, because a signal may be sent to an arbitrary process of a process type.

\[
L_\kappa = Stg_\kappa \times InpQ_\kappa \times Timers_\kappa \times P_\kappa
\]

We write \( \text{stg}(L), \text{inpq}(L), \text{timers}(L) \) and \( \text{ptype}(L) \), where \( L = (\rho, \sigma, \theta, X) \in L_\kappa \), for \( \rho, \sigma, \theta \) and \( X \), respectively.

The global state of a system contains, besides a local state for each existing process, components keeping track of the system time and the pid value issued last, and also a queue for each channel where signals presented to the channel are kept until it is their turn to pass through it. To keep track of the system time and the pid value issued last, natural numbers suffice. We define the set \( ChQ_\kappa \) of channel queues as follows:
ChQₖ = (ExtSigₖ × N)*

Each element in a channel queue contains, in addition to an (extended) signal, a natural number presenting the duration of the delay that it experiences when it does pass through the channel; the arbitrary choice between all possible durations of this delay is made before the signal is put into the channel queue – by means of alternative composition. Global states can be transformed by actions as well as by progress of time. As mentioned above, there may be signals leaving channels and entering the input queues of processes when time progresses to the next time slice, and there may be timers expiring and corresponding signals entering the input queues as well. In so far as channels are concerned, the functions that are used to describe this precisely are the following ones:

unitdelay: ChQₖ → ChQₖ,
arriving: ChQₖ × N → InpQₖ,
coming: ChQₖ → ChQₖ.

The value of unitdelay(γ) is the channel queue γ in which the delay duration of the first signal is decreased by one time unit. The value of arriving(γ, i) is the longest prefix of γ that consists of signals with delay duration zero, weeded of signals with other receivers than i and stripped of delay durations. The value of coming(γ) is the longest suffix of γ that does not start with a signal with delay duration zero. These functions are used to describe the delivery of signals by channels. The function unitdelay is defined by the following equations:

unitdelay(⟨⟩) = ⟨⟩
unitdelay(⟨(sig, i, i'), 0) & γ) = (⟨sig, i, i'⟩, 0) & γ
unitdelay(⟨(sig, i, i'), d + 1) & γ) = (⟨sig, i, i'⟩, d) & γ

The function arriving and coming are recursively defined by

arriving(⟨⟩, i) = ⟨⟩
arriving(⟨(sig, i, i'), 0) & γ, i') = (⟨sig, i, i'⟩ & arriving(γ, i')
arriving(⟨(sig, i, i'), 0) & γ, j') = arriving(γ, j') if i' ≠ j'
arriving(⟨(sig, i, i'), d + 1) & γ, j') = ⟨⟩

coming(⟨⟩) = ⟨⟩
coming(⟨(sig, i, i'), 0) & γ) = coming(γ)
coming(⟨(sig, i, i'), d + 1) & γ) = (⟨sig, i, i'⟩, d + 1) & γ

We define now a set Mₖ of global states which contains proper as well as improper states. Recall that the global state of a system contains a component keeping track of the pid value issued last, a component keeping track of the system time, a channel queue for each channel and a local state for each existing process. The channel queues are indexed by the fixed set of channel names and the local states are indexed by a variable set of pid values, which contains the pid values of the currently existing processes. The improper states are the ones that does not keep the last issued pid value up to date.
$$\mathcal{M}_\kappa = N \times N \times (C_\kappa \rightarrow ChQ_\kappa) \times \bigcup_{I \subseteq \mathbb{N}_1} (I \rightarrow L_\kappa)$$

We write \(cnt(G), now(G), chs(G)\) and \(lsts(G)\), where \(G = (c, n, \Gamma, \Sigma) \in \mathcal{M}_\kappa\), for \(c, n, \Gamma\) and \(\Sigma\), respectively. Note that the local states are indexed by a subset of \(\mathbb{N}_1\). This means that 0 will never serve as the pid value of a process that exists within the system. But 0 is not excluded from being used as a pid value; it is reserved for the environment.

Last, we define the state space \(\mathcal{G}_\kappa:\)

$$\mathcal{G}_\kappa = \{G \in \mathcal{M}_\kappa \mid \forall i \in \text{dom}(lsts(G)) \cdot i \leq cnt(G)\}$$

We write \(\text{exists}(i, G)\), where \(i \in \mathbb{N}\) and \(G \in \mathcal{G}_\kappa\), for \(i \in \text{dom}(lsts(G))\). The state space \(\mathcal{G}_\kappa\) consists exactly of the proper states in \(\mathcal{M}_\kappa\).

4.3 Actions

In this subsection, we will introduce the actions that are used for the semantics of \(\varphi\text{SDL}\). We will make a distinction between the state transforming actions and the actions that do not transform states. For each action \(a\) from the latter kind, the action that appears as the result of executing \(a\) in a state is always the action \(a\) itself; i.e. \(\lambda_G(a \cdot P) = a \cdot \lambda_G(P)\). These actions are called inert actions.

We mentioned before that we will use actions parametrized by domains that are built on expressions denoting values, in contrast with values, as state transforming actions. These expressions are needed because, in general, the values concerned depend on the state in which the actions are executed. The syntax of these expressions, called value expressions, is as follows:

\[
\text{<vexpr> ::=}
\]

\[
\quad \text{<operator nm> } [\ldots] \\
\quad \text{cond ( <vexpr> {,<vexpr>} * )} \\
\quad \text{eval ( <variable nm> , <vexpr> )} \\
\quad \text{value ( <variable nm> , <pid vexpr> )} \\
\quad \text{active ( <signal nm> [ ( <vexpr> {,<vexpr>} * ] )} \\
\quad \text{now} \\
\quad \text{<value nm> } \\
\quad \text{<vexpr> = <vexpr> } \\
\quad \text{cnt} \\
\quad \text{waiting ( <signal nm> {,<signal nm>} * , <pid vexpr> )} \\
\quad \text{type ( <pid vexpr> )} \\
\quad \text{hasinst ( <process nm> )}
\]

We assume a fixed set of terminal productions of \(<\text{value nm}>\) including the special value name \(\text{self}\). We will write \(VExp_\kappa\) for the set of all terminal productions of \(<\text{vexpr}>\) where the set of terminal productions of \(<\text{operator nm}>\), \(<\text{variable nm}>\), \(<\text{signal nm}>\) and \(<\text{process nm}>\) are \(O_{\mathbb{A}}\), \(V_\kappa\), \(S_\kappa\) and \(P_\kappa\), respectively. We will write \(NExp_\kappa\) for \(\{u \in VExp_\kappa \mid \forall G \in \mathcal{G}_\kappa \cdot \text{eval}_G(u) \in \mathbb{N} \cup \{\text{nil}\}\}\).

The first five cases correspond to operator applications, conditional expressions, view expressions, active expressions and the expression \(\text{now}\), respectively,
in SDL. The SDL expressions **parent**, **offspring** and **sender** are regarded as variables accesses, and variable accesses are treated as a special case of view expressions. The sixth case includes **self**, which corresponds to the SDL expressions **self**.

The remaining five cases are needed to reflect the intended meaning of various other SDL constructs exactly. The expression **cnt** is used to associate a unique pid value with each created process. Expressions of the form **waiting**($s_1, \ldots, s_n, u$) are used to give meaning to SDL’s state definitions. They are needed to model that signal consumption is not delayed till the next time slice when there is a signal to consume. Expressions of the forms **type**($u$) and **hasinst**($X$) are used to give meaning to SDL’s output actions. They are needed to check (dynamically) if a receiver with a given pid value is of the appropriate type for a given signal route and to check if a receiver of the appropriate type for a given signal route exists. Expressions of the form $u_1 = u_2$ are, as a matter of course, used to give meaning to SDL’s decisions. Furthermore, they are used with expressions of the form **cnt** or **type**($u$) as left-hand sides where the latter expressions are used.

The state transforming actions are parametrized by several domains that are built on $VExpr_\kappa$:

\[
\begin{align*}
SigD_\kappa &= S_\kappa \times VExpr_\kappa^* \\
ExtSigD_\kappa &= SigD_\kappa \times NExpr_\kappa \times NExpr_\kappa \\
ExtSigP_\kappa &= (S_\kappa \times V_\kappa^*) \times \{\text{nil}\} \times NExpr_\kappa
\end{align*}
\]

The domains $SigD$ and $ExtSigD$ are like $Sig$ and $ExtSig$, respectively, but with $U$ and $N$ replaced by $VExpr_\kappa$ and $NExpr_\kappa$, respectively. The domain $ExtSigP$ differs slightly from $ExtSigD$, because it represents signal patterns, with variables used for the unknown values and nil for “don’t care”.

The following state transforming actions are used:

<table>
<thead>
<tr>
<th>Action</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>$ExtSigP_\kappa \times P_{\delta_\alpha}(S_\kappa)$</td>
</tr>
<tr>
<td>output</td>
<td>$ExtSigD_\kappa \times (C_\kappa \cup {\text{nil}}) \times NExpr_\kappa$</td>
</tr>
<tr>
<td>set</td>
<td>$NExpr_\kappa \times SigD_\kappa \times NExpr_\kappa$</td>
</tr>
<tr>
<td>reset</td>
<td>$SigD_\kappa \times NExpr_\kappa$</td>
</tr>
<tr>
<td>ass</td>
<td>$V_\kappa \times VExpr_\kappa \times NExpr_\kappa$</td>
</tr>
<tr>
<td>$\overline{\tau}$</td>
<td>$P_\kappa \times V_\kappa \times VExpr_\kappa \times (NExpr_\kappa \cup {\text{nil}})$</td>
</tr>
<tr>
<td>stop</td>
<td>$NExpr_\kappa$</td>
</tr>
<tr>
<td>inispont</td>
<td>$NExpr_\kappa$</td>
</tr>
</tbody>
</table>

These are the ACP actions that correspond to input guards, SDL actions, **stop** and **input none**. The second parameter of an **input** action is the save set being in force. The third parameter of an **output** action denotes the delay that the signal experiences if it must pass through a channel. The last parameter of the remaining actions denotes the pid value of the process from which the action originates. Recall that the second and third parameter of an $\overline{\tau}$ action are the formal parameters and the actual parameters, respectively, of the process to be created. The presence of nil needs some further explanation. The second parameter of an **output** action is a channel if the signal to be sent must pass through a channel, and nil otherwise. The last parameter of a $\overline{\tau}$ action is a
value expression denoting the pid value of the creating process if it exists, and nil otherwise – a creating process does not exist for the processes created during system start-up. Similar remarks also apply to the corresponding actions after execution, and to a cr action (see below).

The following inert actions are used:

\[
\begin{align*}
\text{cr} & : P_\kappa \times V_\kappa^{*} \times VExpr_\kappa^{*} \times (NExpr_\kappa \cup \{\text{nil}\}) \\
\text{input'} & : EvtSig_\kappa \times V_\kappa^{*} \\
\text{output'} & : EvtSig_\kappa \times (C_\kappa \cup \{\text{nil}\}) \\
\text{set'} & : N \times Sig_\kappa \times N \\
\text{reset'} & : Sig_\kappa \times N \\
\text{ass'} & : V_\kappa \times U \times N \\
\text{cr'} & : P_\kappa \times V_\kappa^{*} \times U^{*} \times (N \cup \{\text{nil}\}) \\
\text{stop'} & : N
\end{align*}
\]

They do not transform states. They are the actions that appear as the result of executing a state transforming action, except for cr. The instances of cr are used for process creation, leaving instances of cr as a trace. The action \( \text{t} \) is a special action with no observable effect whatsoever. It appears as the result of executing an instance of inispont as well as during system start-up as explained in Section 3.2.

The second parameter of a create action (cr, \( \text{cr} \) or \( \text{cr}^{'} \)) is the sequence of formal parameters for the relevant process type. This is convenient in two ways. Firstly, the alternative to make the association between process types and their formal parameters itself a parameter of the state operator is very unattractive. Secondly, that association is not fully immutable. Recall that the formal parameters are variables and that parameter passing amounts to assigning initial values to these variables – as part of a process creation action. During the start-up of the system, such values are not available and no parameter passing takes place, which corresponds to a different association between process types and formal parameters. This can simply be accomplished in the approach adopted here by using an empty sequence.

### 4.4 State transformers and observers

In the process algebra semantics of \( \varphi \)SDL, which will be presented in Section 5, ACP actions that transform states from \( G_\kappa \) are used to describe the meaning of input guards, SDL actions and \text{stop}. State transforming actions are also needed to initiate spontaneous transitions (indicated by \text{input none}). In the next subsection, we will define the result of executing a process, built up from these actions, in a state from \( G_\kappa \). That is, we will define the relevant state operator. This will, for the most part, boil down to describing how the actions, and the progress of time (modelled by the delay operator \( \sigma_\text{el} \)), transform states. For the sake of comprehensibility, we will first define matching state transforming operations, and also some state observing operations.
A few of the state observing operations are used directly to define the state operator; the others are used to define the evaluation function for the expressions being used in case the values concerned depend on a state. First of all, these expressions are needed as constituents of the actions because the values concerned depend on the state in which these actions are executed. Besides, they are needed as conditions to describe processes that may proceed conditionally, dependent on the state in which they are executed. Various SDL constructs, as a matter of course including decisions, give rise to such processes. In the next subsection, we will define, in addition to the state operator, the above-mentioned evaluation function.

State transformers:

In general, the state transformers change one or two components of the local state of one process. The notable exception is \textit{rcvsig}, which is defined first. It may change all components except the process type. This is a consequence of the fact that the storage, communication and timing mechanisms are rather intertwined on the consumption of signals in SDL. For each state transformer it holds that everything remains unchanged if an attempt is made to transform the local state of a non-existing process. This will not be explicitly mentioned in the explanations given below.

The function \textit{rcvsig} : \textit{ExtSig} \times V_{\kappa} \times \mathcal{G}_{\kappa} \to \mathcal{G}_{\kappa} is used to describe how ACP actions corresponding to SDL's input guards transform states.

$$
rcvsig((\text{sig}, i, i'), (v_1, \ldots, v_n), G) =
\begin{cases}
(cnt(G), now(G), chs(G), lsts(G) \oplus \{ i' \mapsto (\rho, \sigma, \theta, X) \}) & \text{if exists}(i', G) \\
G & \text{otherwise}
\end{cases}
$$

where $\rho = stg(lsts(G);) \oplus \{ v_1 \mapsto vals(\text{sig})_1, \ldots, v_n \mapsto vals(\text{sig})_n, \text{sender} \mapsto i \}$,
$\sigma = \text{rmufirst}(\text{inpq}(lsts(G);, \text{sig}))$,
$\theta = \{ \text{sig} \} \in \text{timers}(lsts(G);)$,
$X = \text{ptype}(lsts(G);)$

\textit{rcvsig}((\text{sig}, i, i'), (v_1, \ldots, v_n), G) deals with the consumption of signal \textit{sig} sent from $i$ to $i'$. It transforms the local state of the receiver as follows:

- the values carried by \textit{sig} are assigned to the local variables $v_1, \ldots, v_n$ of the receiver and the sender's pid value ($i$) is assigned to \textit{sender};
- the first occurrence of \textit{sig} in the input queue of the receiver is removed;
- if \textit{sig} is a timer signal, it is removed from the active timers.

Everything else is left unchanged.

The function \textit{sndsig} : \textit{ExtSig} \times (C_{\kappa} \cup \{ \text{nil} \}) \times N \times \mathcal{G}_{\kappa} \to \mathcal{G}_{\kappa} is used to describe how ACP actions corresponding to SDL's output actions transform states.

$$
sndsig((\text{sig}, i, i'), c, d, G) =$
\[(\text{cnt}(G), \text{now}(G), \text{chs}(G), \text{lsts}(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\})\]

if \(\exists (i', G) \land (c = \text{nil} \lor (\text{chs}(G)_c = \{\} \land d = 0))\)

\[(\text{cnt}(G), \text{now}(G), \text{chs}(G) \oplus \{c \mapsto \gamma\}, \text{lsts}(G))\]

otherwise

\(G\)

\(\text{sndsig}((\text{sig}, i, i'), c, d, G)\) deals with passing signal \(\text{sig}\) from \(i\) to \(i'\), through channel \(c\) with a delay \(d\) if \(c \neq \text{nil}\). If \(c = \text{nil}\), or the queue of \(c\) is empty and \(d = 0\), it transforms the local state of the receiver as follows:

- \(\text{sig}\) is put into the input queue of the receiver, unless \(i' = 0\) (indicating that the environment is the receiver of the signal).

Otherwise, it transforms the queue of the delaying channel as follows:

- \(\text{sig}\) is put into the queue of the delaying channel.

Everything else is left unchanged.

The function \(\text{settimer} : \mathbb{N} \times \mathbb{Sig}_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa\) is used to describe how ACP actions corresponding to SDL's set actions transform states.

\(\text{settimer}(t, \text{sig}, i, G) =\)

\[(\text{cnt}(G), \text{now}(G), \text{chs}(G), \text{lsts}(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\})\] if \(\exists (i, G)\)

\(G\)

otherwise

\(\text{settimer}(t, \text{sig}, i, G)\) deals with setting a timer, identified with signal \(\text{sig}\), to time \(t\). If \(t\) has not yet passed, it transforms the local state of the process with pid value \(i\), the process to be notified of the timer's expiration, as follows:

- the occurrence of \(\text{sig}\) in the input queue originating from an earlier setting, if any, is removed;
- \(\text{sig}\) is included among the active timers with expiration time \(t\); thus overriding an earlier setting, if any.

Otherwise, it transforms the local state of the process with pid value \(i\) as follows:

- \(\text{sig}\) is put into the input queue after removal of its occurrence originating from an earlier setting, if any;
• *sig* is included among the active timers without expiration time.

Everything else is left unchanged.

The function *resettimer* : $\text{Sig}_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's reset actions transform states.

$$\text{resettimer}(\text{sig}, i, G) =$$
$$\{(\text{cnt}(G), \text{now}(G), \text{chs}(G), \text{lsts}(G) \uplus \{i \mapsto (\rho, \sigma, \theta, X)\})\} \text{ if exists}(i, G)$$
$$G$$
$$\text{otherwise}$$

where

- $\rho = \text{stg}(\text{lsts}(G)_i),$  
- $\sigma = \text{rmvfirst}(\text{inpq}(\text{lsts}(G)_i), \text{sig}),$  
- $\theta = \{\text{sig}\} \uplus \text{timers}(\text{lsts}(G)_i),$  
- $X = \text{ptype}(\text{lsts}(G)_i).$

*resettimer*(\text{sig}, i, G) deals with resetting a timer, identified with signal *sig*. It transforms the local state of the process with pid value $i$, the process that would otherwise have been notified of the timer's expiration, as follows:

- the occurrence of *sig* in the input queue originating from an earlier setting, if any, is removed;
- if *sig* is an active timer, it is removed from the active timers.

Everything else is left unchanged.

Notice that *settimer*(\text{t}, \text{sig}, i, G) and *settimer*(\text{t}, \text{sig}, i, *resettimer*(\text{sig}, i, G)) have the same effect. In other words, *settimer* resets implicitly. In this way, at most one signal from the same timer will ever occur in an input queue. Furthermore, SDL keeps timer signals and other signals apart: not a single signal can originate from both timer setting and customary signal sending. Thus, resetting, either explicitly or implicitly, will solely remove signals from input queues that originate from timer setting.

The function *assignvar* : $\text{V}_\kappa \times \text{U} \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's assignment task actions transform states.

$$\text{assignvar}(v, t, i, G) =$$
$$\{(\text{cnt}(G), \text{now}(G), \text{chs}(G), \text{lsts}(G) \uplus \{i \mapsto (\rho, \sigma, \theta, X)\})\} \text{ if exists}(i, G)$$
$$G$$
$$\text{otherwise}$$

where

- $\rho = \text{stg}(\text{lsts}(G)_i) \uplus \{v \mapsto t\},$  
- $\sigma = \text{inpq}(\text{lsts}(G)_i),$  
- $\theta = \text{timers}(\text{lsts}(G)_i),$  
- $X = \text{ptype}(\text{lsts}(G)_i).$

*assignvar*(\text{v}, \text{t}, \text{i}, \text{G}) deals with assigning value \text{t} to variable \text{v}. It transforms the local state of the process with pid value \text{i}, the process to which the variable is local, as follows:

- \text{t} is assigned to the local variable \text{v}, i.e. \text{v} is included among the variables in the storage with value \text{t}; thus overriding an earlier assignment, if any.
Everything else is left unchanged.

The function \(\text{createproc} : P_\kappa \times V_\kappa^* \times U^* \times (\mathbb{N} \cup \{\text{nil}\}) \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa\) is used to describe how ACP actions corresponding to SDL's create actions transform states.

\[
\text{createproc}(X, (v_1, \ldots, v_n), (t_1, \ldots, t_n), i, G) = \\
(cnt(G) + 1, \text{now}(G), \text{chs}(G), \\
lsts(G) \uplus \{cnt(G) + 1 \mapsto (\rho, \sigma, \theta, X), i \mapsto (\rho', \sigma', \theta', X')\} \text{ if } \text{exists}(i, G) \\
(cnt(G) + 1, \text{now}(G), \text{chs}(G), \\
lsts(G) \uplus \{cnt(G) + 1 \mapsto (\rho, \sigma, \theta, X)\}) \text{ if } i = \text{nil} \\
G \text{ otherwise}
\]

where

\[
\rho = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n, \text{parent} \mapsto i\}, \\
\sigma = \{\\}, \\
\theta = \{\\}, \\
\rho' = \text{stg}(lsts(G), i) \uplus \{\text{offspring} \mapsto \text{cnt}(G) + 1\}, \\
\sigma' = \text{inpq}(lsts(G), i), \\
\theta' = \text{timers}(lsts(G), i), \\
X' = \text{ptype}(lsts(G), i)
\]

\(\text{createproc}(X, (v_1, \ldots, v_n), (t_1, \ldots, t_n), i, G)\) deals with creating a process of type \(X\). It increments the last issued pid value which will be used as the pid value of the created process. In addition, it transforms the local state of the process with pid value \(i\), the parent of the created process, as follows:

- the pid value of the created process is assigned to \text{offspring}.

Besides, it creates a new local state for the created process which is initiated as follows:

- the values \(t_1, \ldots, t_n\) are assigned to the local variables \(v_1, \ldots, v_n\) of the created process and the parent's pid value \(i\) is assigned to \text{parent};
- \(X\) is made the process type.

Everything else is left unchanged.

The function \(\text{stopproc} : \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa\) is used to describe how ACP actions corresponding to SDL's \text{stop} transform states.

\[
\text{stopproc}(i, G) = (\text{cnt}(G), \text{now}(G), \text{chs}(G), \{i\} \uplus \text{lsts}(G))
\]

\(\text{stopproc}(i, G)\) deals with terminating the process with pid value \(i\). It disposes of the local state of the process with pid value \(i\). Everything else is left unchanged.

The function \(\text{inispont} : \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathcal{G}_\kappa\) is used to describe how ACP actions used to initiate spontaneous transitions transform states.

\[
\text{inispont}(i, G) = \\
(cnt(G), \text{now}(G), \text{chs}(G), \text{lsts}(G) \uplus \{i \mapsto (\rho, \sigma, \theta, X)\}) \text{ if } \text{exists}(i, G) \\
G \text{ otherwise}
\]

where

\[
\rho = \text{stg}(\text{lsts}(G), i) \uplus \{\text{sender} \mapsto i\}, \\
\sigma = \text{inpq}(\text{lsts}(G), i), \\
\theta = \text{timers}(\text{lsts}(G), i), \\
X = \text{ptype}(\text{lsts}(G), i)
\]
inispont\((i, G)\) deals with initiating spontaneous transitions. It transforms the local state of the process with pid value \(i\), the process for which a spontaneous transition is initiated, by assigning \(i\) to sender. Everything else is left unchanged.

The function \(\text{unitdelay} : G_n \rightarrow P_{\text{fin}}(G_n)\) is used to describe how progress of time transforms states. In general, these transformations are non-deterministic - how signals from channels and expiring timers enter input queues is not uniquely determined. Therefore, this function yields for each state a set of possible states.

\[
G' \in \text{unitdelay}(G) \iff \\
\text{cnt}(G') = \text{cnt}(G) \land \\
\text{now}(G') = \text{now}(G) + 1 \land \\
\forall c \in \text{dom}(\text{chs}(G)) \cdot \text{chs}(G'_c) = \text{coming}(\text{unitdelay}(\text{chs}(G)_c)) \land \\
\forall i \in \text{dom}(\text{lsts}(G)) \cdot \\
\text{stg}(\text{lsts}(G'_i)) = \text{stg}(\text{lsts}(G)_i) \land \\
\exists \sigma \in \text{InpQ} \cdot \\
\text{inpq}(\text{lsts}(G'_i)) = \text{inpq}(\text{lsts}(G)_i) \setminus \sigma \land \\
\sigma \in \text{merge} \cdot \\
\{(\text{arriving}(\text{unitdelay}(\text{chs}(G)_c), i) \mid c \in \text{dom}(\text{chs}(G))) \cup \\
\{(\text{sig}, i, i) \mid \text{timers}(\text{lsts}(G)_i)(\text{sig}) \leq \text{now}(G))\} \land \\
\text{timers}(\text{lsts}(G'_i)) = \\
\text{timers}(\text{lsts}(G)_i) \oplus \{\text{sig} \mapsto \text{nil} \mid \text{timers}(\text{lsts}(G)_i)(\text{sig}) \leq \text{now}(G)\} \land \\
\text{ptype}(\text{lsts}(G'_i)) = \text{ptype}(\text{lsts}(G)_i) \\
\]

\(\text{unitdelay}(G)\) transforms the global state as follows:

- the last issued pid value is left unchanged;
- the system time is incremented with one unit;
- for each channel, the signals leaving the channel within one time unit are removed from its queue;
- for the local state of each process:
  - its storage is left unchanged;
  - the signals leaving any channel within one time unit and having the process as receiver, as well as the signals that notify expiration of any of its timers within one time unit, are put into its input queue in a merging, order preserving, manner;
  - for each of its timers that expire within one time unit, the expiration time is removed;
  - its process type is left unchanged.

State observers:

In general, the state observers examine one component of the local state of one process. The only exception is \(\text{has-instance}\), which may even examine the process type component of all processes. If an attempt is made to observe the local state of a non-existing process, each non-boolean-valued state observer yields \(\text{nil}\) and each boolean-valued state observer yields \(\text{false}\). This will not be explicitly mentioned in the explanations given below. The functions \(\text{nxtsig} : P_{\text{fin}}(S_n) \times \)
\( \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \text{ExtSig}_\kappa \cup \{\text{nil}\} \) and \( \text{nxtsigm} : \mathcal{P}_{\text{fin}}(S_\kappa) \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow S_\kappa \cup \{\text{nil}\} \) are used to define the result of executing ACP actions corresponding to SDL’s input guards in a state.

\[
\text{nxtsig}(ss, i, G) = \text{getnxt}(\text{inpq}(\text{lsts}(G)_i), ss) \text{ if exists}(i, G) \\
\text{nil otherwise}
\]

\( \text{nxtsig}(ss, i, G) \) yields the first signal in the input queue of the process with pid value \( i \) that is of a type different from the ones in \( ss \).

\[
\text{nxtsigm}(ss, i, G) = \text{snm}(\text{sig}(\text{nxtsig}(ss, i, G))) \text{ if nztsg}(ss, i, G) \neq \text{nil} \\
\text{nil otherwise}
\]

\( \text{nxtsigm}(ss, i, G) \) yields the type of the first signal in the input queue of the process with pid value \( i \) that is of a type different from the ones in \( ss \).

The function \( \text{contents} : V_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathcal{U} \cup \{\text{nil}\} \) is used to describe the value of expressions of the form \( \text{value}(v, u) \) which correspond to SDL’s variable accesses and view expressions.

\[
\text{contents}(v, i, G) = \rho(v) \text{ if exists}(i, G) \wedge v \in \text{dom}(\rho) \\
\text{nil otherwise}
\]

where \( \rho = \text{stg}(\text{lsts}(G)_i) \)

\( \text{contents}(v, i, G) \) yields the current value of the variable \( v \) that is local to the process with pid value \( i \).

The function \( \text{is-active} : \text{Sig}_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathbb{B} \) is used to describe the value of expressions of the form \( \text{active}(\text{sig}, u) \) which correspond to SDL’s active expressions.

\[
\text{is-active}(\text{sig}, i, G) = \text{true if exists}(i, G) \wedge \text{sig} \in \text{dom}(\text{timers}(\text{lsts}(G)_i)) \\
\text{false otherwise}
\]

\( \text{is-active}(\text{sig}, i, G) \) yields true iff \( \text{sig} \) is an active timer signal of the process with pid value \( i \).

The function \( \text{is-waiting} : \mathcal{P}_{\text{fin}}(S_\kappa) \times \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathbb{B} \) is used to describe the value of expressions of the form \( \text{waiting}(s_1, \ldots, s_n, u) \) which are used to give meaning to SDL’s state definitions.

\[
\text{is-waiting}(ss, i, G) = \text{true if exists}(i, G) \wedge \text{nxtsig}(ss, i, G) \neq \text{nil} \\
\text{false otherwise}
\]

\( \text{is-waiting}(ss, i, G) \) yields true iff there is a signal in the input queue of the process with pid value \( i \) that is of a type different from the ones in \( ss \).

The function \( \text{type} : \mathbb{N} \times \mathcal{G}_\kappa \rightarrow \mathbb{P}_\kappa \cup \{\text{env}, \text{nil}\} \) is used to describe the value of expressions of the form \( \text{type}(u) \) which are used to give meaning to SDL’s output actions with explicit addressing.

\[
\text{type}(i, G) = \text{ptype}(\text{lsts}(G)_i) \text{ if exists}(i, G) \\
\text{env if } i = 0 \\
\text{nil otherwise}
\]

332
\(\text{type}(i, G)\) yields the type of the process with pid value \(i\). Different from the other state observers, it yields a result if \(i = 0\) as well, viz. \text{env}.

The function \(\text{has-instance} : (P, \{\text{env}\}) \times G \rightarrow \mathbb{B}\) is used to describe the value of expressions of the form \(\text{hasinst}(X)\), where \(X\) is a process name, which are used to give meaning to SDL’s output actions with implicit addressing.

\[
\text{has-instance}(X, G) = \begin{cases} 
\text{true} & \text{if } \exists i \in \mathbb{N} \cdot (i = 0 \lor \exists \text{exists}(i, G)) \land \text{type}(i, G) = X \\
\text{false} & \text{otherwise}
\end{cases}
\]

\(\text{has-instance}(X, G)\) yields true iff there exists a process of type \(X\).

### 4.5 State operator and evaluation function

In this subsection, we will finally define the state operator that is used to describe, in whole or in part, the SDL mechanisms for storage, communication, timing and process creation. We will not define the \textit{action} and \textit{effect} functions explicitly, as in [1]. Instead we will define, for each state transforming action \(a\), the result of executing a process of the form \(a \cdot P\) in a state from \(G\). Because progress of time transforms states as well, we will also define the result of executing a process of the form \(\sigma_{\text{rel}}(P)\) in a state. In addition, we will define the evaluation function that is used to describe the value of an expression \(u\) in a state \(G\).

**State operator:**

The state transformers defined in Section 4.4 are used below to describe the state \(G'\) resulting from executing a state transforming action \(a\) in a state \(G\). In general, the action \(a'\) that appears as the result of executing a state transforming action \(a\) in a state \(G\) is the action \(a\) with the expressions occurring in it replaced by their values in state \(G\). However, there are exception to this rule for the input actions and the output actions. For output actions, the delay duration is additionally stripped of. Input actions deviate more. The constituents of an input action \(a\) are a pattern of an (extended) signal and a set of signal types, and the constituents of the corresponding action \(a'\) are a signal matching this pattern and the sequence of variables occurring in the pattern. That a signal pattern is replaced by a matching signal is to be expected, the sequence of variables is added because it shows to which variables the values carried by the signal have been assigned, and the set of signal types is removed because there is no use to retain it after execution. There is still another exception for the actions used to initiate spontaneous transitions. As mentioned before, \(\ell\) appears as the result of executing these actions.

We will first define the result of executing a process of the form \(a \cdot P\) in a state from \(G\) for the state transforming ACP actions corresponding to SDL’s input guards, output actions, set actions, reset actions, assignment task actions, create actions and the terminator \text{stop}, and for the state transforming ACP actions of

\[\lambda_G(a \cdot P) = a' \cdot \lambda_G(P)\]
the form $\text{inispont}(u)$ which will be used to set $\text{sender}$ properly when spontaneous transitions take place. All this is rather straightforward with the state transformers defined in Section 4.4; only the case of the ACP actions corresponding to SDL’s input guards needs further explanation. If the value of at least one of the expressions occurring in an ACP action is undefined in the state concerned, the action will fail, i.e. yield deadlock. Different from the other cases, the execution of an action $\text{input}(((s,(v_1,\ldots,v_n)),\text{nil},u),ss)$ may fail in certain states for other reasons as well. It fails if the type of the first signal in the input queue of the process referred to by $u$ with a type not occurring in $ss$ is different from $s$. Otherwise, it succeeds, the values carried by this signal are assigned to the local variables $v_1,\ldots,v_n$ of the process concerned, and the signal is removed from the input queue.

$$\lambda_G(\text{input}(((s,(v_1,\ldots,v_n)),\text{nil},u'),ss)) \cdot P =$$

$$\text{input}(\text{nxt},(v_1,\ldots,v_n)) \cdot \lambda_{\text{retrans}}(\text{nxt},(v_1,\ldots,v_n),G)(P) \quad \text{if } i' \neq \text{nil} \land$$

$$\text{nxtsig}(ss,i',G) = s \quad \text{otherwise}$$

where $\text{nxt} = \text{nxtsig}(ss,i',G)$,

$$i' = \text{eval}_G(u')$$

$$\lambda_G(\text{output}(((s,(u_1,\ldots,u_n)),u,u'),c,u'')) \cdot P =$$

$$\text{output}((\text{sig},i,i'),c) \cdot \lambda_{\text{send}}((\text{sig},i,i'),c,d,G)(P) \quad \text{if } t_1 \neq \text{nil} \land \ldots \land t_n \neq \text{nil} \land$$

$$i \neq \text{nil} \land i' \neq \text{nil} \land d \neq \text{nil} \quad \text{otherwise}$$

where $\text{sig} = (s,(t_1,\ldots,t_n))$,

$$t_j = \text{eval}_G(u_j) \quad (\text{for } 1 \leq j \leq n),$$

$$i = \text{eval}_G(u),$$

$$i' = \text{eval}_G(u')$$

$$\lambda_G(\text{reset}((s,(u_1,\ldots,u_n)),u')) \cdot P =$$

$$\text{reset}((\text{sig},i) \cdot \lambda_{\text{settimer}}((\text{sig},i,i),G)(P) \quad \text{if } t \neq \text{nil} \land t_1 \neq \text{nil} \land \ldots \land t_n \neq \text{nil} \land i \neq \text{nil} \quad \text{otherwise}$$

where $t = \text{eval}_G(u)$,

$$\text{sig} = (s,(t_1,\ldots,t_n)),$$

$$t_j = \text{eval}_G(u_j) \quad (\text{for } 1 \leq j \leq n),$$

$$i = \text{eval}_G(u')$$

$$\lambda_G(\text{assign}(u,v,u') \cdot P =$$

$$\text{assign}(v,t,i) \cdot \lambda_{\text{assignvar}}(v,t,i,G)(P) \quad \text{if } t \neq \text{nil} \land i \neq \text{nil} \quad \text{otherwise}$$

$$\delta$$
where \( t = \text{eval}_G(u) \),
\[ i = \text{eval}_G(u') \)

\[ \lambda_G(\overline{cr}(X, \overline{fpars}, \langle u_1, \ldots, u_n \rangle, u) \cdot P) = \]
\[ \overline{cr}'(X, \overline{fpars}, \overline{apars}, i) \cdot \lambda_{\text{createproc}(X, \overline{fpars}, \overline{apars}, \overline{apars}, i, G)}(P) \] \text{ if } t_1 \neq \text{nil} \land \ldots \land t_n \neq \text{nil}
\[ \overline{cr} \]
\[ \text{otherwise} \]

where \( \overline{apars} = (t_1, \ldots, t_n) \),
\[ t_j = \text{eval}_G(u_j) \] \text{ for } 1 \leq j \leq n,
\[ i = \text{eval}_G(u) \]

\[ \lambda_G(\overline{stop}(u) \cdot P) = \overline{stop}(i) \cdot \lambda_{\text{stopproc}(i, G)}(P) \] \text{ if } i \neq \text{nil}
\[ \overline{stop} \]
\[ \text{otherwise} \]

where \( i = \text{eval}_G(u) \)

\[ \lambda_G(\overline{inispolnt}(u) \cdot P) = \overline{inispolnt}(i, G)(P) \] \text{ if } i \neq \text{nil}
\[ \overline{inispolnt} \]
\[ \text{otherwise} \]

where \( i = \text{eval}_G(u) \)

Here \( \text{eval}_G \) is used to describe the value of expressions, occurring in a state transforming action, in state \( G \). This evaluation function will be defined later on.

Recall that for each inert action \( a \), we simply have
\[ \overline{a} \cdot \overline{P} \]

We will now proceed with defining the result of executing a process of the form \( \sigma_{\text{rel}}(P) \) in a state from \( G \). This case is quite different from the preceding ones. Executing a process that is delayed till the next time slice in some state means that the execution is delayed till the next time slice and, in general, that it takes place in another state due to the progress of time. Usually, it is not uniquely determined how progress of time transforms states. This leads to the following equation:
\[ \lambda_G(\sigma_{\text{rel}}(P)) = \sigma_{\text{rel}}(\sum_{G' \in \text{unidelay}(G)} \lambda_{G'}(P)) \]

Evaluation function:

We will end this section with defining the evaluation function that was already used to describe the value of an expression \( u \) in a state \( G \). Most state observers defined in Section 4.4 are used to define this function. If the value of at least one of the subexpressions occurring in an expression is undefined in the state concerned, the expression will be undefined, i.e. yield \( \text{nil} \).

The SDL expressions are covered by the first six cases, as explained in Section 4.3. These cases do not need any further explanation except the remark that the meta-variable \( x \) ranges over a set of variables in the sense of \( \mu \text{CRL} \) that includes \( \text{self} \), a special variable corresponding to the SDL expression \( \text{self} \).
\[
eval_G(\text{op}(u_1, \ldots, u_n)) =
\begin{cases}
\text{op} & \text{if } \eval_G(u_1) \neq \text{nil} \land \ldots \land \eval_G(u_n) \neq \text{nil} \\
\text{nil} & \text{otherwise}
\end{cases}
\]

\[
eval_G(\text{cond}(u_1, u_2, u_3)) =
\begin{cases}
\eval_G(u_2) & \text{if } \eval_G(u_1) = \text{true} \\
\eval_G(u_3) & \text{if } \eval_G(u_1) = \text{false} \\
\text{nil} & \text{otherwise}
\end{cases}
\]

\[
eval_G(\text{value}(v, u)) =
\begin{cases}
\text{contents}(v, \eval_G(u), G) & \text{if } \eval_G(u) \neq \text{nil} \\
\text{nil} & \text{otherwise}
\end{cases}
\]

\[
eval_G(\text{active}(s, \langle u_1, \ldots, u_n \rangle), u)) =
\begin{cases}
\text{is-active}(\text{sig}, \eval_G(u), G) & \text{if } \eval_G(u_1) \neq \text{nil} \land \ldots \land \eval_G(u_n) \neq \text{nil} \land \\
\eval_G(u) \neq \text{nil} \\
\text{nil} & \text{otherwise}
\end{cases}
\]

where \(\text{sig} = \langle s, \langle \eval_G(u_1), \ldots, \eval_G(u_n) \rangle \rangle\)

\[
eval_G(\text{now}) = \text{now}(G)
\]

\[
eval_G(x) = x
\]

The remaining cases are about expressions which are used in Section 5, as explained in Section 4.3 as well. They are very straightforward.

\[
eval_G(u_1 = u_2) =
\begin{cases}
\text{true} & \text{if } \eval_G(u_1) = \eval_G(u_2) \\
\text{false} & \text{if } \eval_G(u_1) \neq \eval_G(u_2) \\
\text{nil} & \text{otherwise}
\end{cases}
\]

\[
eval_G(\text{cnt}) = \text{cnt}(G)
\]

\[
eval_G(\text{waiting}(s_1, \ldots, s_n, u)) =
\begin{cases}
\text{is-waiting}(\{s_1, \ldots, s_n\}, \eval_G(u), G) & \text{if } \eval_G(u) \neq \text{nil} \\
\text{nil} & \text{otherwise}
\end{cases}
\]

\[
eval_G(\text{type}(u)) = \text{type}(\eval_G(u), G) \text{ if } \eval_G(u) \neq \text{nil}
\]

\[
eval_G(\text{hasinst}(X)) = \text{has-instance}(X, G)
\]

5 Process algebra semantics

In this section, we will present a process algebra semantics of \(\varphi\)SDL. It relies heavily upon the specifics of the state operator defined in Section 4.5. Here, all peculiar details of the semantics, inherited from full SDL, become visible.

The semantics of \(\varphi\)SDL is defined by interpretation functions, one for each syntactic category, which are all written in the form \(\lbrack \rbrack^n\). The superscript \(\kappa\) is used to provide contextual information where required. The exact interpretation function is always clear from the context. We will be lazy about specifying the range of each interpretation function, since this is usually clear from the context as well. Many of the interpretations are expressions, equations, etc. They will simply be written in their display form. We will in addition assume that the interpretation of a name is the same name.
5.1 System definition

The meaning of a system definition is a quadruple \((P, \phi, E, G)\) where:

- \(P\) is a process expression describing the behaviour of the system from its start-up;
- \(\phi\) is the mapping from process names to process expressions that is to be associated with the process creation operator used in \(P\);
- \(E\) is the set of recursive process-equations defining the processes corresponding to the SDL states referred to in the process expressions in the range of \(\phi\);
- \(G\) is the state space that is to be associated with the state operator used in \(P\).

The first component depends on the names introduced by the definitions of channels and process types, and on the given numbers of processes to be created during the start-up of the system for the process types defined. The second and third component depend heavily on the process definitions proper. The last component depends simply on the names introduced by the definitions of variables, signal types, channels and process types — this means that the state space depends solely on purely syntactic aspects of the system.

The meaning of each definition occurring in a system definition is a pair \((\phi, E)\) where:

- \(\phi\) is singleton mapping from process names to process expressions if it is the definition of a process type, and an empty mapping otherwise;
- \(E\) is the set of recursive process-equations defining the processes corresponding to the SDL states referred to in the single process expression in the range of \(\phi\) if it is the definition of a process type, and an empty set otherwise.

In case of a process definition, the first component is expressed in terms of the meaning of its start transition and the second component in terms of the meaning of its state definitions. We write \([D]_{\phi}\) and \([D]_{E}\), where \([D]_{\phi} = (\phi, E)\), for \(\phi\) and \(E\), respectively. Thus, we have \([D]^* = ([D]_{\phi}, [D]_{E})\).

The second and third component of the meaning of a system definition are obtained by taking the union of the first components and second components, respectively, of the meaning of all definitions occurring in it.

\[
\text{[system } S; D_1 \ldots D_n \text{ endsystem]} := \tau_{\{1\}} \circ \lambda G_0 \circ E_\phi (P), [D_1]_{\phi} \cup \ldots \cup [D_n]_{\phi}, [D_1]_E \cup \ldots \cup [D_n]_E, G_\kappa
\]

where \(P = \|X \in \text{process}(\kappa)(\|\text{init}(x,X) \Rightarrow (X, \langle \rangle, \langle \rangle, \text{nil}))\),
\(G_\phi = (0, 0, \{c \mapsto \langle \rangle | c \in \text{vars}(\kappa)\}, \{\})\),
\(\kappa = \{\text{system } S; D_1 \ldots D_n \text{ endsystem}\}\)

\[
\text{[process } X(\lambda); \text{fpar } u_1, \ldots, u_n; \text{start; tr } d_1 \ldots d_n \text{ endprocess]}^* := \langle \{X \mapsto \sum_{s \in \text{set} \cdot \phi} \text{cnt} = \text{self} \mapsto [tr]^*\}, \{[d_1]^*, \ldots, [d_n]^*\}\rangle
\]

where \(\kappa' = \text{uptscoopeunitt}(\kappa, X)\)
\[ [D]^{\kappa} := \{\{\}, \{\}\} \text{ if the definition } D \text{ is not of the form } \]

\[
\text{process } X(k); \ \text{fpar } v_1, \ldots, v_m; \ \text{start}; \ tr \ d_1 \ldots d_n \ \text{endprocess;}
\]

In the case of a system definition, the process expression \( \tau_{\text{in}(t)} \circ \lambda_{G_0} \circ E_s(P) \) expresses that, for each process type defined, the given initial number of processes are created and the result is executed in the state \( G_0 \). Additionally, the internal action \( t \) as well as the actions in \( I \) are hidden. \( I \) is to be regarded as a parameter of the semantics. If one takes the empty set for \( I \), one gets an extreme semantics, viz. a concrete one corresponding to the viewpoint that all internal actions of a system are observable. By taking appropriate non-empty sets, one can get a range of more abstract semantics, including the interesting one that corresponds to the viewpoint that only the communication with the environment is observable. \( G_0 \) is the state in which the last issued pid value and the system time are zero, there is an empty queue for each channel defined, and there are no local states. Recall that the pid value zero is reserved for the environment and that a newly created process gets its pid value and local state only when its execution starts. In the case of a process definition, the process expression in the singleton mapping \( \{X \mapsto \sum_{self: \kappa_0} \text{cnt} = \text{self} \mapsto [tr]^{\kappa} \} \) expresses that, for each process of the type \( X \), its behaviour is the behaviour determined by the given start transition \( tr \) if \( self \) stands for the last issued pid value.

### 5.2 Process behaviours

The meaning of a state definition, occurring in the scope of a process definition, is a process-equation defining, for the process type defined, the common behaviour of its instances from the state being defined (using parametrization by the identifying pid value \( self \)). It is expressed in terms of the meaning of its transition alternatives, which are process expressions describing the behaviour from the state being defined for the individual signal types of which instances may be consumed and, in addition, possibly for some spontaneous transitions. The meaning of each transition alternative is in turn expressed in terms of the meaning of its input guard, if the alternative is not a spontaneous transition, and its transition.

\[
\begin{align*}
\text{[state } st; \ \text{save } s_1, \ldots, s_m; \ \text{alt}_1 \ldots \text{alt}_n]^* := \\
X_{st} &= \neg \text{waiting}(s_1, \ldots, s_m, \text{self}) \mapsto ([\text{alt}_1]^{\kappa'} + \ldots + [\text{alt}_n]^{\kappa'}) + \\
&\quad \text{waiting}(s_1, \ldots, s_m, \text{self}) \mapsto \sigma_{\text{ref}}(X_{st})
\end{align*}
\]

where \( X = \text{scopeunit}(\kappa), \)
\( \kappa' = \text{updsaveset}(\kappa, \{s_1, \ldots, s_m\}) \)

\[
\begin{align*}
\text{[input } s(v_1, \ldots, v_n); tr]^{\kappa} := \\
(it(\text{cnt}, n_0) \mapsto 0)^* (\neg it(\text{cnt}, n_0) \mapsto \text{input}((s, \langle v_1, \ldots, v_n \rangle), \text{nil, self}), ss) \cdot [tr]^{\kappa} \\
\end{align*}
\]

where \( n_0 = \sum_{X \in \text{procs}(\kappa)} \text{init}(\kappa, X).^7 \)
\( ss = \text{saveset}(\kappa) \)

^7Here, we use \( \sum \) for summation of a set of natural numbers.
In the case of a state definition, the process-equation describes that the processes of type $X$ behave from the state $st$ as one of the given transition alternatives, and that this behaviour is possibly delayed till the first future time slice in which there is a signal to consume if there are no more signals to consume in the current time slice. In process-equations, we use names of process types with state name subscripts, such as $X_{st}$ above, as variables; in process expressions elsewhere, we use them to refer to the processes defined thus. Note that, in the absence of spontaneous transitions, a delay becomes inescapable if there are no more signals to consume in the current time slice. In the case of a guarded transition alternative, the process expression $\text{input}((s, (v_1, \ldots, v_n)), \text{nil, self}), ss)\cdot [tr]^x$ expresses that the transition $tr$ is initiated on consumption of a signal of type $s$; iteration is used to guarantee that no communication takes place till the start-up of the system has come to an end. In the case of an unguarded transition alternative, the process expression expresses that the transition $tr$ is initiated spontaneously, i.e. without a preceding signal consumption, with sender set to the value of self.

The meaning of a transition, occurring in the scope of a process definition, is a process expression describing the behaviour of the transition. It is expressed in terms of the meaning of its actions and its transition terminator.

$[a_1 \ldots a_n \text{nextstate } st]^x := [a_1]^x \cdot \ldots \cdot [a_n]^x \cdot X_{st}$

where $X = \text{scopeunit}(\kappa)$

$[a_1 \ldots a_n \text{stop}]^x := [a_1]^x \cdot \ldots \cdot [a_n]^x \cdot \text{stop}(\text{self})$

$[a_1 \ldots a_n \text{dec}]}^x := [a_1]^x \cdot \ldots \cdot [a_n]^x \cdot [\text{dec}]^x$

In the case of a transition terminated by $\text{nextstate } st$, the process expression expresses that the transition performs the actions $a_1, \ldots, a_n$ in sequential order and ends with entering state $st$ - i.e. goes on behaving as defined for state $st$ of the processes of the type defined. In case of termination by $\text{stop}$, it ends with ceasing to exist; and in case of termination by a decision $\text{dec}$. it goes on behaving as described by $\text{dec}$.

Of course, the meaning of a decision is a process expression as well. It is expressed in terms of the meaning of its expressions and transitions.

$[\text{decision } e_1; \text{tr}_1 \ldots (e_n); \text{tr}_n \text{enddecision}]^x :=$

$[e] = [e_1] := [\text{tr}_1]^x + \ldots + [e] = [e_n] := [\text{tr}_n]^x$

$[\text{decision any; } (\ldots \text{tr}_{n-1} ; (e_n); \text{tr}_n \text{enddecision}]^x := [\text{tr}_1]^x + \ldots + [\text{tr}_n]^x$

In the case of a decision with a question expression $e$, the process expression expresses that the decision transfers control to the transition $\text{tr}_1$ for which the value of $e$ equals the value of $e_i$. In the case of a decision with any instead,
the process expression expresses that the decision transfers non-deterministically control to one of the transitions \( tr_1, \ldots, tr_n. \)

The meaning of an SDL action is also a process expression. It is expressed in terms of the meaning of the expressions occurring in it. It also depends on the occurring names (names of variables, signal types, signal routes and process types - dependent on the kind of action).

\[
\text{[output } s(e_1, \ldots, e_n) \text{ to } c \text{ via } r_1, \ldots, r_m]^k := \\
(\text{lt}(cnt, n_0) :\to \{\})^* \\
(\neg \text{lt}(cnt, n_0) :\to (\text{type}(\langle [e] \rangle) = X_1 :\to P_1 + \ldots + \text{type}(\langle [e] \rangle) = X_m :\to P_m + \\
\neg(\text{type}(\langle [e] \rangle) = X_1 \lor \ldots \lor \text{type}(\langle [e] \rangle) = X_m) :\to \{\})
\]

where \( n_0 = \sum_{X \in \text{procs}(\kappa)} \text{init}(\kappa, X), \)

for \( 1 \leq j \leq m: \\
P_j = \frac{\text{output}((s, (\langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self}, \langle [e] \rangle), c_j, 0)}{\sum_{d \in N} \text{output}((s, \langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self}, \langle [e] \rangle), c_j, d)} \quad \text{if } c_j = \text{nil} \\
X_j = \text{recv}(\kappa, r_j), \\
c_j = \text{ch}(\kappa, r_j)
\]

\[
\text{[output } s(e_1, \ldots, e_n) \text{ via } r_1, \ldots, r_m]^k := \\
(\text{lt}(cnt, n_0) :\to \{\})^* \\
(\neg \text{lt}(cnt, n_0) :\to (\sum_{N} \text{type}(i) = X_1 :\to P_1 + \ldots + \text{type}(i) = X_m :\to P_m + \\
\neg(\text{hasinst}(X_1) \land \ldots \land \text{hasinst}(X_m)) :\to \{\})
\]

where \( n_0 = \sum_{X \in \text{procs}(\kappa)} \text{init}(\kappa, X), \)

for \( 1 \leq j \leq m: \\
P_j = \frac{\text{output}((s, (\langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self}, \langle [e] \rangle), c_j, 0)}{\sum_{d \in N} \text{output}((s, \langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self}, \langle [e] \rangle), c_j, d)} \quad \text{if } c_j = \text{nil} \\
X_j = \text{recv}(\kappa, r_j), \\
c_j = \text{ch}(\kappa, r_j)
\]

\[
\text{[set } (e, (e_1, \ldots, e_n));]^{k} := \text{set}(\langle [e] \rangle, (s, (\langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self})
\]

\[
\text{[reset } (s(e_1, \ldots, e_n));]^{k} := \text{reset}(s, (\langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self})
\]

\[
\text{[task } v := e_i;]^{k} := \text{ass}(v, [e], \text{self})
\]

\[
\text{[create } X(e_1, \ldots, e_n);]^{k} := \text{cr}(X, \text{fpars}(\kappa, X), (\langle [e_1] \rangle, \ldots, \langle [e_n] \rangle), \text{self})
\]

All cases except the ones for output actions are straightforward. The cases of output actions needs further explanation. The receiver of a signal sent via a certain signal route must be of the receiver type associated with that signal route. Therefore, the conditions of the form \( \text{type}(u) = X_j \) are used. In the case of an output action with a receiver expression \( e \), if none of the signal routes \( r_1, \ldots, r_m \) has the type of the process with pid value \( e \) as its receiver type, or a process with that pid value does not exist, the signal is simply discarded and no error occurs. This is expressed by the summand \( \neg(\text{type}(\langle [e] \rangle) = X_1 \lor \ldots \lor \text{type}(\langle [e] \rangle) = X_m) :\to \{\}. \)
In the case of an output action without a receiver expression, first an arbitrary choice from the signal routes $r_1, \ldots, r_m$ is made and thereafter an arbitrary choice from the existing processes of the receiver type for the chosen signal route is made. However, there may be no existing process of the receiver type for that signal route. Should this occasion arise, the signal is simply discarded. This is expressed by the summand $\neg (\text{hasinst}(X_1) \land \ldots \land \text{hasinst}(X_m)) \rightarrow \emptyset$. Note that this occasion may already arise if there is one signal route for which there exists no process of its receiver type. Note further that a process expression of the form $\sum_{d \in \mathbb{N}} \text{output}(\text{sig}, c, d)$ is used for each signal route containing a delaying channel $c$. Thus, the arbitrary delay is modelled by an arbitrary choice between all possible delay durations $d$ as already mentioned in Section 4.2. As for input guards, iteration is used to guarantee that no communication takes place till the start-up of the system has come to an end.

5.3 Values

The meaning of an SDL expression is given by a translation to a value expression of the same kind. There is a close correspondence between the SDL expressions and their translations. Essential of the translation is that self is added where the local states of different processes need to be distinguished. Consequently, a variable access $v$ is just treated as a view expression $\text{view}(v, \text{self})$. For convenience, the expressions parent, offspring and sender are also regarded as variable accesses.

\[
\begin{align*}
\text{[op}(e_1, \ldots, e_n)] & := \text{op}([e_1], \ldots, [e_n]) \\
\text{[if e}_1\text{ then e}_2\text{ else e}_3\text{ fi]} & := \text{cond}([e_1], [e_2], [e_3]) \\
\text{[v]} & := \text{value}(v, \text{self}) \\
\text{[view}(v, e)] & := \text{value}(v, [e]) \\
\text{[active}(s(e_1, \ldots, e_n)))] & := \text{active}((s, ([e_1], \ldots, [e_n])), \text{self}) \\
\text{[now]} & := \text{now} \\
\text{[self]} & := \text{self} \\
\text{[parent]} & := \text{value}(\text{parent}, \text{self}) \\
\text{[offspring]} & := \text{value}(\text{offspring}, \text{self}) \\
\text{[sender]} & := \text{value}(\text{sender}, \text{self})
\end{align*}
\]

All cases are very straightforward and need no further explanation. This is due to the choice of value expressions and the evaluation function defined on them in Section 4.5.
6 Closing remarks

Models of highly reactive and distributed systems, in particular telecommunication systems, are frequently made using SDL. This is done with, among other things, the intention to allow for the analysis of their behaviour. Largely due to their intrinsic reactive and distributed nature, giving considerations to time is inherent to the analysis of the behaviour of such systems. The semantics of SDL according to the ITU/TS recommendation is at some points insufficiently precise, and at other points too complex, to allow for interesting analysis; in particular the time related features of SDL, such as timers and channels with delay, miss an adequate semantics. Besides, the existing tools for analysis of models described in SDL are very limited; at best a limited kind of model checking, closely related to simulation of the described behaviour, is provided, and no time related features are supported. In a joint project of KPN Research – the research institute of the telecommunications operator PTT Telecom and the industrial affiliation of the second author – and Utrecht University, a state-of-the-art model checker is adapted to the common needs for analysis of systems modelled using $\varphi$SDL. The intention of that work is to do some first steps in the improvement of the possibilities for analysis of models described in SDL. The work on $\varphi$SDL reported in this paper was initiated by that project.

In [10] a foundation for the semantics of SDL, based on streams and stream processing functions, has been proposed. This proposal indicates that the SDL view of systems gives an interesting type of dynamic dataflow networks, but the treatment of time in the proposal is however too sketchy to be used as a starting point for the semantics of the time related features of SDL. In [11] and [12] attempts have been made to give a structured operational semantics of SDL, the latter including the time related features. However, not all relevant details were worked out, and the results will probably have to be turned inside out in order to deal with full SDL. At the outset, we also tried shortly to give a structured operational semantics of SDL, but we found that it is very difficult, especially if time aspects have to be taken into account. Of course, a structured operational semantics can be derived from the process algebra semantics, and most probably, we will have to do so for the above-mentioned project.

References


A Notational conventions

Meta-language for syntax:
The syntax of SDL is described by means of production rules in the form of an extended BNF grammar. The curly brackets "{" and "}" are used for grouping. The asterisk "*" and the plus sign "+" are used for zero or more repetitions and one or more repetitions, respectively, of curly bracketed groups. The square brackets "[" and "]" are also used for grouping, but indicate that the group is optional. An underlined part included in a nonterminal symbol does not belong to the context free syntax; it describes a semantic condition.

Special set, function and sequence notation:
We write $\mathcal{P}(A)$ for the set of all subsets of $A$, and we write $\mathcal{P}_{\text{fin}}(A)$ for the set of all finite subsets of $A$.

We write $f : A \rightarrow B$ to indicate that $f$ is a total function from $A$ to $B$, that is $f \subseteq A \times B \land \forall x \in A \cdot \exists y \in B \cdot (x, y) \in f$. If $A$ is finite, we emphasize this by writing $f : A \rightarrow B$ instead. We write $\text{dom}(f)$, where $f : A \rightarrow B$, for $A$. For an (ordered) pair $(x, y)$, where $x$ and $y$ are intended for argument and value of some function, we use the notation $x \mapsto y$ to emphasize this intention. The binary operators $\setminus$ (domain subtraction) and $\oplus$ (overriding) on functions are defined by

$$A \triangleleft f = \{ x \mapsto y \mid x \in \text{dom}(f) \land x \notin A \land f(x) = y \}$$

$$f \oplus g = (\text{dom}(g) \triangleleft f) \cup g$$

For a function $f : A \rightarrow B$, presenting a family $B$ indexed by $A$, we use the notation $f_i$ (for $i \in A$) instead of $f(i)$.

Functions are also used to present sequences; as usual we write $(x_1, \ldots, x_n)$ for the sequence presented by the function $\{ 1 \mapsto x_1, \ldots, n \mapsto x_n \}$. The binary operator $\_\_\_$ stands for concatenation of sequences. We write $x \_\_\_ t$ for $(x) \_\_\_ t$. 

344
B Contextual information

The meaning of a \( \varphi \text{SDL} \) construct generally depends on the definitions in the scope in which it occurs. Contexts are primarily intended for modeling the scope. The context that is ascribed to a complete system definition is also used to define the state space used to describe its meaning. The context of a construct contains all names introduced by the definitions of variables, signal types, channels, signal routes and process types occurring in the system definition on hand and additionally:

- if the construct occurs in the scope of a process definition, the name introduced by that process definition, called the scope unit;
- if the construct occurs in the scope of a state definition, the set of names occurring in the save part of that state definition, called the save set.

In case of a signal route, the name is in addition connected with the names of its receiver type and its delaying channel, if present; and in case of a process type, the name is connected with the names of the variables that are its formal parameters and the number of processes of this type that have to be created during the start-up of the system.

\[
\text{Context} = \ \mathcal{P}_{\tilde{\text{fa}}} (\text{VarId}) \times \mathcal{P}_{\tilde{\text{fa}}} (\text{SigId}) \times \mathcal{P}_{\tilde{\text{fa}}} (\text{ChanId}) \times \mathcal{P}_{\tilde{\text{fa}}} (\text{RouteDes}) \times \mathcal{P}_{\tilde{\text{fa}}} (\text{ProcDes}) \times \ (\text{ProcId} \cup \{\text{nil}\}) \times \mathcal{P}_{\tilde{\text{fa}}} (\text{SigId})
\]

where \( \text{RouteDes} = \text{RouteId} \times (\text{ProcId} \cup \{\text{env}\}) \times (\text{ChanId} \cup \{\text{nil}\}) \)

\[
\text{ProcDes} = \text{ProcId} \times \text{VarId}^* \times \text{N}
\]

We write \( \text{vars}(\kappa) \), \( \text{sigs}(\kappa) \), \( \text{chans}(\kappa) \), \( \text{routed}(\kappa) \), \( \text{procds}(\kappa) \), \( \text{scopeunit}(\kappa) \) and \( \text{saveset}(\kappa) \), where \( \kappa = (V, S, C, Rd, Pd, X, ss) \in \text{Context} \). for \( V, S, Ch, Rd, Pd, X \) and \( ss \), respectively. We write \( \text{procs}(\kappa) \) for \( \{ X \mid \exists vs. k \cdot (X, vs, k) \in \text{procds}(\kappa) \} \).

For constructs that do not occur in a process definition, the absence of a scope unit will be represented by \( \text{nil} \) and, for constructs that do not occur in a state definition, the absence of a save set will be represented by \( \{ \} \).

Useful operations on \( \text{Context} \) are the functions

\[
\text{rev} : \text{Context} \times \text{RouteId} \rightarrow \text{ProcId} \cup \{\text{env}\}.
\]

\[
\text{ch} : \text{Context} \times \text{RouteId} \rightarrow \text{ChanId} \cup \{\text{nil}\}.
\]

\[
\text{fpars} : \text{Context} \times \text{ProcId} \rightarrow \text{VarId}^*,
\]

\[
\text{init} : \text{Context} \times \text{ProcId} \rightarrow \text{N},
\]

\[
\text{updscopeunit} : \text{Context} \times \text{ProcId} \rightarrow \text{Context},
\]

\[
\text{updsaveset} : \text{Context} \times \mathcal{P}_{\tilde{\text{fa}}} (\text{SigId}) \rightarrow \text{Context}
\]

defined below. The functions \( \text{rev} \) and \( \text{ch} \) are used to extract the receiver type and the delaying channel, respectively, of a given signal route from the context. These functions are inductively defined by

\[
(\tau, X, c) \in \text{routed}(\kappa) \Rightarrow \text{rev}(\kappa, \tau) = X,
\]

\[
(\tau, X, c) \in \text{routed}(\kappa) \Rightarrow \text{ch}(\kappa, \tau) = c
\]
The functions \( jpars \) and \( init \) are used to extract the formal parameters and the initial number of processes, respectively, of a given process type from the context. These functions are inductively defined by

\[
(X, \alpha, k) \in \text{procds}(\kappa) \Rightarrow jpars(\kappa, X) = \alpha,
\]

\[
(X, \alpha, k) \in \text{procds}(\kappa) \Rightarrow init(\kappa, X) = k.
\]

The functions \( updscopeunit \) and \( updsaveset \) are used to update the scope unit and the save set, respectively, of the context. These functions are inductively defined by

\[
\kappa = (V, S, C, Rd, Pd, X, ss) \Rightarrow updscopeunit(\kappa, X') = (V, S, C, Rd, Pd, X', ss),
\]

\[
\kappa = (V, S, C, Rd, Pd, X, ss) \Rightarrow updsaveset(\kappa, ss') = (V, S, C, Rd, Pd, X, ss').
\]

The context ascribed to a system definition is a minimal context in the sense that the contextual information available in it is common to all contexts on which constructs occurring in it depend. The additional information that may be available applies to the scope unit for constructs occurring in a process definition and the save set for constructs occurring in a state definition. The context ascribed to a system definition is obtained by taking the union of the corresponding components of the (partial) contexts contributed by all definitions occurring in it, except for the scope unit and the saveset which are permanently the same – nil and \{ \}, respectively.

\[
\begin{align*}
\{\text{system } S; D_1 \ldots D_n \text{endsystem;}\} & := \\
& (\text{vars}(\{D_1\}) \cup \ldots \cup \text{vars}(\{D_n\}), \\
& \text{sigs}(\{D_1\}) \cup \ldots \cup \text{sigs}(\{D_n\}), \\
& \text{chans}(\{D_1\}) \cup \ldots \cup \text{chans}(\{D_n\}), \\
& \text{routeds}(\{D_1\}) \cup \ldots \cup \text{routeds}(\{D_n\}), \\
& \text{procds}(\{D_1\}) \cup \ldots \cup \text{procds}(\{D_n\}), \\
& \text{nil, \{ \}}) \\
\{\text{dcl } v_1 \} & := (\{v_1\}, \{\}, \{\}, \{\}, \text{nil, \{ } ) \\
\{\text{signal } s(T_1, \ldots, T_n);\} & := (\{\}, \{s\}, \{\}, \{\}, \text{nil, \{ } ) \\
\{\text{channel } c_1\} & := (\{\}, \{\}, \{c\}, \{\}, \text{nil, \{ } ) \\
\{\text{signalroute } r \text{ from } X_1 \text{ to } X_2 \text{ with } s_1, \ldots, s_n \text{ delayed by } c;\} & := \\
(\{\}, \{\}, \{\}, \{(r, X_2, c)\}, \{\}, \text{nil, \{ } ) \\
\{\text{process } X(\lambda); \text{fpar } v_1, \ldots, v_m; \text{ start; tr } d_1 \ldots d_n \text{endprocess;}\} & := \\
(\{\}, \{\}, \{\}, \{\}, \{(X, \langle c_1, \ldots, v_m \rangle, k)\}, \text{nil, \{ } )
\end{align*}
\]
TSF, a Test Specification Formalism

Loe Feijs, Mark Huizer
Eindhoven University of Technology

Abstract

TTCN is the ISO language for defining tests of protocol services and protocol entities. We report from a project studying the semantics of TTCN using process algebra, which is feasible, in principle. TTCN is a complex language with many features, some of which are essential for testing, others being typically in the style of traditional imperative languages. Using the insights gained from this study we synthesized TSF, a new language, which contains two essential ingredients for testing, but resembles PSF in all other aspects. Amongst other things, TSF contains a novel operator $\rightarrow$. It is argued that a test itself is a formal object, the correctness of which with respect to a given specification is subject to formal analysis. This is illustrated by a simple example.

Key Words: Protocols, Testing, TTCN, formal semantics, PSF.

1 Introduction and motivation

TTCN is the ISO language for defining tests of protocol entities and protocol services [1]. In general, the subject of software testing is highly relevant because in the industrial practice of software development it often takes a significant fraction of the total development effort. For testing protocols, ISO has established a framework containing terminology and concepts. Important issues in protocol testing are the distinction between conformance testing and interoperability testing (see e.g [2]), the fact that telecommunication is a multi-vendor business where protocol entities from distinct suppliers must cooperate. Not only software is to be tested, but combined hardware-software systems. And of course protocol testing demands languages capable of expressing communication behaviour. Typical studies are [3] (relating tests and sequence charts) and [4] (studying test methodologies in a formal setting). Often communication systems are specified and realized using SDL, or in a combination of a high-level language (such as LOTOS, ESTELLE and PSF) and a programming language (C). TTCN is meant as a means of describing tests at an abstract level whereas interworkings and message sequence charts sometimes play a role when deriving tests from SDL. Much
research has already been done on automated test generation (see [5] for a survey). There is still some ongoing debate whether one should use 'normal languages' for describing tests, or special languages.

We undertook to study TTCN and its semantics. For this purpose we translated an essential part of the language to ACP and did various exercises in manual calculations with this semantics. From this study we learned amongst other things that TTCN is a complex language with an un-orthodox syntax, and that TTCN contains certain concepts which are specific for testing and which are interesting for further study and application. We do not express any particular opinion about the design of the language TTCN, but we found that it is quite remote from the kind of language constructs encountered in the ACP and PSF worlds.

This paper contains an attempt to bridge the gap by proposing a language called TSF. It is our goal to keep TSF sufficiently simple to understand its syntax and semantics in terms of initial algebra specifications and process algebra, just like PSF. This enables the direct application of theories and tool technology already developed for ACP and PSF. PSF has already been successfully applied to a large variety of protocols. [7] [8]

One could envisage translators from TTCN to TSF as well as a combined PSF/TSF simulation environment. If protocols are specified in PSF, the tests can thus be simulated themselves before putting them into action on the real systems under test.

It is important to note that a test itself is a formal object, the correctness of which with respect to a given specification is subject to formal analysis. This will be illustrated by a simple example.

In Section 2 a survey of TTCN is given. In Section 3 a simple example is given in TTCN. The sections on TTCN can be skipped by readers wanting to focus on TSF (sections 4 to 6 do not rely on definitions from the sections on TTCN). In Section 4 the new language TSF is introduced. In Section 5 a formal semantics for TSF is proposed, using ACP. In Section 6 the same example is given in TSF and the work of Section 5 is applied to this example. In Section 7 some conclusions are drawn.

2 TTCN

TTCN has a great variety of features, some of which are essential for its main purpose of testing, whereas other features are similar to those encountered in other imperative languages for describing communicating processes.

In our view, the following two language features of TTCN are essential ingredients for testing:

- verdicts: the purpose of a test is to give a kind of 'yes' or 'no' answer with respect to the question whether the implementation under test is in order or not. This is called a 'verdict.' TTCN has three
verdicts: pass, inconclusive and fail. Furthermore there is a notion of ‘preliminary verdict’.

- ordering on alternative statements: typical tests check on ‘expected’ behaviour first and fall back to other options only if the expected behaviour is not offered by the implementation under test.

Many other features in TTCN do not seem essential for testing purposes, but are orthogonal to the test features. These are needed to give the language sufficient expressive power. In the TSF proposal (Section 4) we do not use these but we adopted PSF-like constructs. These TTCN features include:

- sequential composition by means of increasing level of indentation;
- alternative composition (ordered) by means of remaining at the same level of indentation;
- subroutine mechanism (called tree attachment), denoted by +T when calling a subtree T;
- assignments to local and global variables;
- a CSP-like syntax for input and output (‘?’ and ‘!’);
- constraints on communications, where a constraint on a send is used to give a package a value, and a constraint on a receive is used as a guard;
- guards.

We gave a formal semantics for a part of TTCN by using ACP. Due to space limitations and the complexity of TTCN, we cannot give the details here; it suffices to say that we worked along the same lines described in Section 5 (but we did some work on TTCN first, turning to TSF later).

During execution, a TTCN ‘program’ maintains a kind of global variable called R (for result) containing the verdict obtained so far. Verdicts are updated in a particular way, according to the table given below. The following verdict values are considered: N, or None is an initialization value; P, or Pass is a final verdict telling that the implementation under test (IUT) is in order (for some aspect tested); F, or Fail states that the test stops with a negative outcome; I for Inconclusive is an undetermined outcome (e.g. the network failed so that the test could not complete); R or Result means that the test should stop and that the verdict obtained so far becomes the final verdict. E denotes a test error (telling that the test itself is in error, rather than the IUT). Moreover there are preliminary verdicts, also called temporary verdicts, which are written as (P), (I) and (F). These are useful for giving preliminary conclusions, which can be updated later. The following table gives the result of a stated verdict on the variable R:
Some further details of TTCN will be explained using a very small example in the next section.

3 Example in TTCN

3.1 The one-place buffer

Let us assume that we want to test a one-place buffer, called BUF, which will be our model of an IUT. It is supposed to have two points of control and observation (PCO), which is test terminology for 'port'. The PCOs are U (upper) and L (lower). There are two service primitives: DU (data upper) and DL (data lower), which carry data values which are non-negative integers.

The supposed behaviour is as follows: initially the buffer is empty. It accepts one input via U and then it is ready to output its value via L. After that it is empty again and it can accept new input. When non-empty, BUF should not engage in any input communication.

3.2 TTCN test of the buffer

TTCN provides mechanisms for ordering tests into test suites, for declaring variables and constraints, and so on. Here we focus on the dynamic behaviour, which takes the shape of one or more tables in which the essential behaviour is given as a so-called tree. Assume constraints CU0 and CL0 have been declared somehow and that they demand that the data values contained in DU and DL are zero, respectively. Then the following table is a TTCN test case.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U?DU</td>
<td>CU0</td>
<td>Pass</td>
<td></td>
</tr>
<tr>
<td>L?DL</td>
<td>CL0</td>
<td>Fail</td>
<td></td>
</tr>
<tr>
<td>L?Otherwise</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We ought to explain the tree notation. In TTCN, indentation has meaning: going to a deeper level of indentation denotes sequential composition. Alternatives are expressed by statements at the same level of indentation. Alternatives must be tried in the stated order, so \( L?D \) with constraint \( C\Theta \) must be tried first and only if there is no DL message containing a zero, the next alternative is tried (\( L?\text{Otherwise} \)), which will result in Fail.

For a correct implementation of a one-place-buffer, the test above leads to the expected answer (a verdict trace with final verdict 'Pass').

4 TSF: a test specification formalism

In order to represent the essentials of TTCN in an alternative way, a new language is constructed, TSF (Test Specification Formalism). This language will contain essential ingredients of TTCN, and will resemble PSF.

Where possible, we refer to PSF (see [7]) for the production rules. This is represented by \(<\text{as-in-psf}>\) in the syntax-rules. The specification of TSF in Extended Backus Naur Form is given below. The following notation is used: \{ item sep \}+ means one or more occurrences of item, separated by sep.

\[
\begin{align*}
\langle \text{specification} \rangle &::= \langle \text{module} \rangle^+ \\
\langle \text{module} \rangle &::= \langle \text{data-module} \rangle \mid \langle \text{test-module} \rangle \\
\langle \text{data-module} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{var-ident-list} \rangle &::= \{ \langle \text{var-ident} \rangle \}^+ \\
\langle \text{test-module} \rangle &::= \"test\" \"module\" \langle \text{module-ident} \rangle \\
&\quad \text{"begin"} \\
&\quad \left[ \langle \text{t-exports} \rangle \right] \\
&\quad \left[ \langle \text{imports} \rangle \right] \\
&\quad \left[ \langle \text{tests} \rangle \right] \\
&\quad \left[ \langle \text{sets} \rangle \right] \\
&\quad \left[ \langle \text{communications} \rangle \right] \\
&\quad \left[ \langle \text{t-variables} \rangle \right] \\
&\quad \left[ \langle \text{definitions} \rangle \right] \\
&\quad \text{"end"} \langle \text{module-ident} \rangle \\
\langle \text{t-exports} \rangle &::= \"exports\" \\
&\quad \text{"begin"} \\
&\quad \left[ \langle \text{atoms} \rangle \right] \\
&\quad \left[ \langle \text{tests} \rangle \right] \\
&\quad \left[ \langle \text{sets} \rangle \right] \\
&\quad \text{"end"} \\
\langle \text{imports} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{atoms} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{tests} \rangle &::= \"tests\" \langle \text{test-decl-list} \rangle^+ \\
\langle \text{test-decl-list} \rangle &::= \{ \langle \text{test-ident} \rangle \}^+ [ \".\" \langle \text{input-type} \rangle ] \\
\langle \text{input-type} \rangle &::= \{ \langle \text{sort-ident} \rangle \}^+ [ \"#\" ]^+ \\
\langle \text{sets} \rangle &::= \langle \text{as-in-psf} \rangle \\
\langle \text{t-variables} \rangle &::= \"variables\" \langle \text{t-variable-list} \rangle
\end{align*}
\]
The principles of TSF are as follows. There are two kinds of modules: data modules, as usual, and test modules, similar to process modules. Tests can communicate with processes (supposed to be specified elsewhere, say in PSF), but there is no merge operator. Two constructs need explanation:

- **verdict( t ).** The term \( t \) can be used for indicating 'pass' or 'fail' values etc. It is assumed that the values themselves are defined in a special data module (e.g. called Verdict) which exports a sort VERDICT, a constant \texttt{none} and a binary function \texttt{update} from \texttt{VERDICT # VERDICT} to \texttt{VERDICT}. There is a built-in assignable variable called \texttt{result}. Its initial value is \texttt{none}. When using \texttt{verdict( t )}, the type of \( t \) must be \texttt{VERDICT}. The meaning of \texttt{verdict( t )} is that \texttt{result} is assigned the value \texttt{update( t, r')}, where \( r' \) denotes the value of \texttt{result} in the previous state.

- **\( x \leftrightarrow y \).** This means that \( x \) is tried first, and only if \( x \) can make no step (e.g. because no matching communication is offered), then \( y \) is done. The operator \( \leftrightarrow \) is called \textit{preferential alternative composition} because it behaves like \( + \), but if possible it chooses the first option amongst its alternatives.

At the syntactic level, there is no \( + \) operator. In order to keep things simple, we forbid tests communicating with each other. The top level merge
is never written down. If one envisages a combined PSF/TSF simulation environment, then the merge takes place.

The prescribed data module could be as follows, where it is understood that the sort VERDICT and the functions none and update are obligatory, whereas pass, ..., error etc. can be added by the user. The rules for processing a given verdict imply that a verdict can be issued only once (later we shall present alternative data modules for VERDICT).

```
data module FinalVerdict
begin
  exports
  begin
    sorts
      VERDICT
    functions
      none : -> VERDICT
      update : VERDICT # VERDICT -> VERDICT
      pass : -> VERDICT
      fail : -> VERDICT
      inconclusive : -> VERDICT
      error : -> VERDICT
    end
  variables
    r : -> VERDICT
  equations
    -- getting started
    [00] update(r,none) = r
    [01] update(none,r) = r
    -- propagating errors
    [02] update(r,error) = error
    [03] update(error,r) = error
    -- processing multiple final verdicts
    [04] update(pass,pass) = error
    [05] update(pass,fail) = error
    [06] update(pass,inconclusive) = error
    [07] update(fail,pass) = error
    [08] update(fail,fail) = error
    [09] update(fail,inconclusive) = error
    [10] update(inconclusive,pass) = error
    [12] update(inconclusive,inconclusive) = error
  end FinalVerdict
```
5 TSF semantics

5.1 Outline

We claim that ACP offers the features to interpret the new constructs.

- $\Lambda_a$ to describe the state-based concept of current 'result' value.
- $\theta$ and $+$ for interpreting $x \leftrightarrow y$.

Some care is needed with respect to the use of $\theta$. Consider the test

$r(a) \leftrightarrow r(b) \leftrightarrow r(c)$ having three alternatives. Classically one would define

$\gamma(s(n), r(n)) = c(n)$ etc. Now we must label the occurrences of receive actions, yielding $r^0(a)$, $r^1(b)$ and $r^2(c)$ according to their syntactic position

as an argument of $\leftrightarrow$. Let us call this labeling function $\psi$. We label

verdict statements too. Labelling starts at zero. We must adopt a $\gamma$ which preserves the labeling: $\gamma(s(n), r^i(n)) = c^i(n)$. We forbid things like $\gamma(s^i(n), r^j(n))$.

Now we can order the $c^i(n)$ such that $c^0(n) > c^1(n) > c^2(n)$. In this way we find that the ACP interpretation of the test is $r^0(a) + r^1(b) + r^2(c)$.

The translation is non-compositional in the sense that an implementation $I$ and a test $T$, each being an ACP term, must be combined together in the scope of the $\theta$ operator.

$$\theta(\partial_H(I \parallel \Lambda_\sigma(\psi(0, T))))$$

Here $\sigma$ is the initial state which maps $\text{result}$ to $\text{none}$ and $H$ is the set which hides the $r$ and $s$ actions but lets the $c$ actions pass (as usual). So $\theta$ selects amongst the $c$ actions.

5.2 Details

In this section we shall formalize the essential steps outlined in Section 5.1. The ACP terms are as usual, but some of the atomic actions can be labeled with numbers. Furthermore we allow for guarded action of the form $[e] \rightarrow x$, where the guard $e$ is an equation between data terms. These guarded actions are precisely as in PSF (version 1.1 and up). For finite data domains, the following two axioms allow for elimination of guards:

$$[t_1 = t_2] \rightarrow x = x \quad \text{if } t_1 = t_2 \text{ holds}$$

$$[t_1 = t_2] \rightarrow x = \delta \quad \text{if } t_1 \neq t_2 \text{ holds}$$

When dealing with infinite data domains, the terms $t_1$ and $t_2$ may contain bound variables, as for example $d$ in $\sum_{d \in D} [d = 0] \rightarrow x(d)$. This means that the applicability of the above two axioms is restricted to terms in which all such variables have become bound.

First we must describe the function $\psi$ which maps instances of the non-terminal $<\text{test}>$ to ACP terms. In this definition, the set of atomic actions

of TSF, with typical element $a$, is the set containing $\text{skip}$, $\text{verdict}(v)$ for
all \( v \) of type \textsc{verdict} and all instances of the non-terminal <atom>. Furthermore, \( e \) is an arbitrary equation, that is, an instance of the non-terminal <equation>.

\[
\begin{align*}
\psi(n, a) &= a^n & \text{(for atomic action } a) \\
\psi(n, [e] \rightarrow x) &= [e] \rightarrow \psi(n, x) \\
\psi(n, \sum(d \in D, x(d))) &= \sum_{d \in D} \psi(n, x(d)) \\
\psi(n, x.y) &= \psi(n, x) \cdot \psi(0, y) \\
\psi(n, x + y) &= \psi(n, x) + \psi(n + \varphi(x), y)
\end{align*}
\]

where we use an auxiliary function \( \varphi \) which returns as \( \varphi(x) \) the number of fresh labels issued when labeling \( x \), that is, when mapping \( x \) to \( \psi(n, x) \).

\[
\begin{align*}
\varphi(a) &= 1 & \text{(for atomic action } a) \\
\varphi([e] \rightarrow x) &= \varphi(x) \\
\varphi(\sum(d \in D, x(d))) &= 0 & (\text{if } D = \emptyset) \\
&= \varphi(x(d)) & (\text{if } D \neq \emptyset) \\
\varphi(x.y) &= \varphi(x) \\
\varphi(x + y) &= \varphi(x) + \varphi(y)
\end{align*}
\]

The <definitions> clause of a TSF specifications contains a number of defining equations for named tests. It is understood that the function \( \psi(0, \ldots) \) is applied to the right-hand side of each instance of <definition>. We must restrict ourselves to defining equations which are guarded in the sense that each recursive call is preceded by at least one atomic action. Therefore \( \psi \) need not be applied to recursive calls, that is, \( \psi(n, T) = T \) if \( T \) is an instance of <simple-test>.

Next we must give the definition of the communication function \( \gamma \), which will be derived from the <communications> clause of the TSF text under consideration. Syntactically, the <communications> clause is the same as in PSF, but whereas in PSF it can be directly interpreted as a definition of \( \gamma \), for TSF we should now explain precisely the way in which the labeling passes through \( \gamma \). The <communications> clause consists of a number of rules and we consider the following typical form for such rule:

\[
r_0(n) \mid r_1(n) = c(n) \text{ for } n \in D
\]

For example, \( r_0(n) \) could be a sending atom and \( r_1(n) \) a matching receive action. We must adopt a restriction for the arguments of \( \mid \) in each rule: one argument must be an atom declared in the TSF text, whereas the other one must be declared in an imported PSF process module. Each rule satisfying this restriction gives rise to the following equations for \( \gamma \):

\[
\begin{align*}
\gamma(r_0(n)^m, r_1(n)) &= c(n)^m & \text{(all } m \in \mathbb{N}, n \in D) \\
\gamma(r_0(n), r_1(n)^m) &= c(n)^m & \text{(all } m \in \mathbb{N}, n \in D)
\end{align*}
\]

Then it is understood that \( \gamma \) is defined by the conjunction of these equations, combining the results of all rules in the <communications> clause, together with the following two closure clauses:
\[ \gamma(a, b) = \gamma(b, a) \quad \text{if } \gamma(b, a) \text{ defined} \]
\[ \gamma(a, b) = \text{undefined} \quad \text{otherwise} \]

We work in ACP, and then the process algebra axioms describe how \( \gamma \) is extended to the usual \( I \) between arbitrary terms.

Next we must give the definition of the encapsulation operator \( \partial_H \), which requires the derivation of a set \( H \) from the \texttt{<communications>} clause of the TSF text under consideration. We define \( H \) as the union of sets
\[
\{ r(n) \mid n \in D \} \cup \{ r(n)^m \mid n \in D, m \in \mathbb{N} \}
\]
where \( r \) ranges over all atom identifiers, assuming that \( D \) is the type of \( r \), which occur in the left hand side of one or more of the rules of the \texttt{<communications>} clause.

Next we must give the definition of the priority ordering \( < \) on the set of atomic actions, which is required for the proper application of the priority operator \( \theta \).

\[
a^m < b^n \quad \text{iff } m > n \quad \text{(all } m, n \in \mathbb{N})
\]
for all atomic actions \( a, b \in \{ c(n) \mid n \in D \} \cup \{ \text{skip} \} \cup \{ \text{verdict}(v) \mid v \in \text{VERDICT} \} \). In this way an action which appears first in a list of alternatives with respect to \( + > \), and which thus has the lowest label, gets the highest priority.

Finally we give the definition of the functions \texttt{`action'} and \texttt{`effect'}, which are required for the proper application of the generalized state operator \( \Lambda \). We describe them for all communications actions \( r \) occurring in the left hand side of one or more of the rules of the \texttt{<communications>} clause and also for \texttt{skip} and \texttt{verdict}(v). The set of states, with typical element \( \sigma \), is defined as the set of functions which maps our only variable, \texttt{'result'}, to elements of the set \texttt{VERDICT}.

\[
\begin{align*}
\text{action}(r(n)^m, \sigma) &= r(n)^m \\
\text{effect}(r(n)^m, \sigma, r(n)^m) &= \sigma \\
\text{action}(\text{skip}^m, \sigma) &= \text{skip}^m \\
\text{effect}(\text{skip}^m, \sigma, \text{skip}^m) &= \sigma \\
\text{action}(\text{verdict}(v)^m, \sigma) &= \text{verdict}(v, \text{update}(v, \sigma(\text{result})))^m \\
\text{effect}(\text{verdict}(v)^m, \sigma, \text{verdict}(v, v')^m) &= \sigma[v'/\text{result}]
\end{align*}
\]

In this way the \( \Lambda \) operator lets most actions pass unchanged, except for the verdicts, which are processed in two ways: first, whenever a verdict is executed, it leaves a term \texttt{verdict}(v, v') in the resulting trace, where \( v \) is the stated verdict, and \( v' \) is the accumulated effect of this and all previous verdicts, as accumulated in \( \sigma \). Secondly, the state \( \sigma \) is updated. Please note that we use some overloading: the binary \texttt{verdict} occurs in the resulting traces, whereas the single-valued \texttt{verdicts}, roughly speaking, are the \texttt{verdict} statements written in the PSF text.

If the test developer or executer is only interested in the final outcome of the test, he can take the second argument \( (v') \) of the last \texttt{verdict} term in the execution trace.

356
5.3 Algebraic properties

The following laws should hold on the basis of the intended application and the given intuition for the preferential alternative composition.

\[
\begin{align*}
x \leftrightarrow \delta &= x \\
\delta \leftrightarrow x &= x \\
x \leftrightarrow (y \leftrightarrow z) &= (x \leftrightarrow y) \leftrightarrow z \\
(x \leftrightarrow y) \cdot z &= x \cdot z \leftrightarrow y \cdot z
\end{align*}
\]

For the model given in Section 5.2 where TSF tests are mapped by \( \psi \) to ACP terms, these laws hold (see the theorem below). But it would be interesting to have other models for \( \leftrightarrow \) which abstract away from the particularities of our labeling function \( \psi \).

**Theorem.** For \( x, y \) and \( z \) instances of the nonterminal <test> and identifying \( \sum_{\delta \in \Theta} \) with \( \delta \) for all \( \delta \), the following hold:

1. \( ACP_c \vdash \psi(n, x \leftrightarrow \delta) = \psi(n, x) \)
2. \( ACP_c \vdash \psi(n, \delta \leftrightarrow x) = \psi(n, x) \)
3. \( ACP_c \vdash \psi(n, x \leftrightarrow (y \leftrightarrow z)) = \psi(n, (x \leftrightarrow y) \leftrightarrow z) \)
4. \( ACP_c \vdash \psi(n, (x \leftrightarrow y) \cdot z) = \psi(n, x \cdot z \leftrightarrow y \cdot z) \)

**Proof.** We use the definitions of \( \psi \) and \( \varphi \) together with some of the process algebra laws (A1,A2,A4,A6).

1. \( \psi(n, x \leftrightarrow \delta) \)
   \[
   = \psi(n, x) + \psi(n + \varphi(x), \delta)
   = \psi(n, x) + \delta =_{A6} \psi(n, x).
   \]

2. \( \psi(n, \delta \leftrightarrow x) \)
   \[
   = \psi(n, \delta) + \psi(n + \varphi(\delta), x)
   = \delta + \psi(n + 0, x)
   = \delta + \psi(n, x) =_{A1,A6} \psi(n, x).
   \]

3. \( \psi(n, x \leftrightarrow (y \leftrightarrow z)) \)
   \[
   = \psi(n, x) + \psi(n + \varphi(x), y \leftrightarrow z)
   = \psi(n, x) + \psi(n + \varphi(x), y) + \psi(n + \varphi(x) + \varphi(y), z)
   =_{A2} (\psi(n, x) + \psi(n + \varphi(x), y)) + \psi(n + \varphi(x) + \varphi(y), z)
   = \psi(n, x \leftrightarrow y) + \psi(n + \varphi(x \leftrightarrow y), z) = \psi(n, (x \leftrightarrow y) \leftrightarrow z).
   \]

4. \( \psi(n, (x \leftrightarrow y) \cdot z) \)
   \[
   = \psi(n, x \leftrightarrow y) \cdot \psi(0, z)
   = (\psi(n, x) + \psi(n + \varphi(x), y)) \cdot \psi(0, z)
   =_{A4} \psi(n, x) \cdot \psi(0, z) + \psi(n + \varphi(x), y) \cdot \psi(0, z)
   = \psi(n, x \cdot z) + \psi(n + \varphi(x \cdot z), y \cdot z) = \psi(n, x \cdot z \leftrightarrow y \cdot z).
   \]

\( \square \)
6 Example in TSF

6.1 TSF test of the buffer

The test example already discussed in Section 3 can be cast into the TSF format. We use the data module Naturals from the PSF standard library. The data module Verdict is given in the appendix (FinalVerdict as given before would do well too). The process module OnePlaceBuffer models the IUT. It could be specified in PSF (see Section 6.2), or it could be some real IUT.

```plaintext
test module TestOnePlaceBuffer
begin
  exports
  begin
    atoms
      s-DU : NATURAL
      r-DL : NATURAL
    tests
      TST1
  end

  imports
    Naturals, Verdict, OnePlaceBuffer

  tests
    TST1

  communications
    s-DU(n) | r-DU(n) = DU(n) for n in NATURAL
    s-DL(n) | r-DL(n) = DL(n) for n in NATURAL

  definitions
    TST1 = s-DU(zero) +
      sum( n in NATURAL , [n=zero] -> r-DL(n) . verdict(pass)
          +) r-DL(n) . verdict(fail)
  end TestOnePlaceBuffer
```

6.2 Formal analysis of the tests

Using a correct implementation of a One-place-buffer, we show that the test above leads to the expected answer (pass). We start from the following PSF specification of the IUT, which is called BUF.
process module OnePlaceBuffer
begin
exports
begin
atoms
  r-DU : NATURAL
  s-DL : NATURAL
processes
  BUF
end
imports
Naturals
processes
BUF
definitions
BUF = sum( n in NATURAL
  , r-DU(n) . s-DL(n) ) . BUF
end OnePlaceBuffer

First of all we present BUF as an equation in process algebra. The data set D equals \( \mathbb{N} \), as can be concluded from an analysis of Naturals.

\[
BUF = \sum_{n \in D}(r-DU(n) \cdot s-DL(n)) \cdot BUF
\]

Next we interpret the test module TestOnePlaceBuffer as an equation in process algebra. We apply the labeling function \( \psi \) immediately, finding the following translation for the definition of TST1.

\[
\psi(0, TST1) = s-DU(0)^0 \cdot \sum_{n \in D} \left( [n = 0] \rightarrow r-DL(n)^0 \cdot \text{verdict(pass)}^0 + r-DL(n)^1 \cdot \text{verdict(fail)}^0 \right)
\]

The initial state \( \sigma \) is given by \( \sigma = (\text{result} \mapsto \text{none}) \). Next we can derive our communication function \( \gamma \).

\[
\begin{align*}
\gamma(s-DU(n)^m, r-DU(n)) &= DU(n)^m & (\text{all } m \in \mathbb{N}, n \in D) \\
\gamma(s-DU(n), r-DU(n)^m) &= DU(n)^m & (\text{all } m \in \mathbb{N}, n \in D) \\
\gamma(s-DL(n)^m, r-DL(n)) &= DL(n)^m & (\text{all } m \in \mathbb{N}, n \in D) \\
\gamma(s-DL(n), r-DL(n)^m) &= DL(n)^m & (\text{all } m \in \mathbb{N}, n \in D)
\end{align*}
\]

And of course we have the corresponding equations with swapped arguments of \( \gamma \). The set \( H \) contains all actions of the form \( s-DU(n), r-DU(n), s-DL(n), r-DL(n) \) together with all labeled versions of these actions. After these preparations we are ready to perform a calculation and check whether BUF will pass the test TST1. We write simply \text{none} for the state \( \sigma \) such that
\( \sigma(\text{result}) = \text{none} \) and similarly for \text{pass} and \text{fail}. We have to use some shorthands: \( v \) abbreviates \text{verdict}, \( p \) abbreviates \text{pass} and \( f \) abbreviates \text{fail}.

\[
\begin{align*}
\theta(\partial_H(\text{BUF} \parallel \Lambda_{\text{none}}(\psi(0, \text{TST1})))) & = \\
\theta(\partial_H(\text{BUF} \parallel s-DU(0)^0 \cdot (\Lambda_{\text{none}}(\sum_n ([n = 0] \to \ldots + \ldots)))) & = \\
\theta(DU(0)^0 \cdot \partial_H((s-DL(0) \cdot \text{BUF}) \parallel \Lambda_{\text{none}}(\sum_n ([n = 0] \to \ldots + \ldots))) & = \\
DU(0)^0 \cdot \theta(\partial_H((s-DL(0) \cdot \text{BUF}) \parallel (\sum_n \Lambda_{\text{none}}([n = 0] \to r-DL(n)^0 \cdot v(p)^0) + \sum_n \Lambda_{\text{none}}(r-DL(n)^1 \cdot v(f)^0)))) & = \\
DU(0)^0 \cdot \theta(\partial_H((s-DL(0) \cdot \text{BUF}) \parallel (\Lambda_{\text{none}}(r-DL(0)^0 \cdot v(p)^0) + \Lambda_{\text{none}}(r-DL(0)^1 \cdot v(f)^0)))) & = \\
DU(0)^0 \cdot \theta(\text{DL}(0)^0 \cdot \partial_H(\text{BUF} \parallel \Lambda_{\text{none}}(v(p)^0)) + \text{DL}(0)^1 \cdot \partial_H(\text{BUF} \parallel \Lambda_{\text{none}}(v(f)^0))) & = \\
DU(0)^0 \cdot \text{DL}(0)^0 \cdot \theta(\text{BUF} \parallel (v(p, \text{update}(p, \text{none})) \cdot \Lambda_{\text{pass}}(e)))) & = \\
DU(0)^0 \cdot \text{DL}(0)^0 \cdot v(p, p) & = \\
DU(0)^0 \cdot \text{DL}(0)^0 \cdot \text{verdict}(\text{pass}, \text{pass})
\end{align*}
\]

To get the final outcome of the test, we take the second argument of the last \text{verdict} term in the execution trace, which is \text{pass}.

When aiming at a thorough analysis of the test, this is only half the job. We should also compose the test with one or more incorrect IUTs and check if the test yields \text{fail} for at least one of them.

## 7 Conclusions

It is nice that the entire mechanism of verdict updating can be captured by an algebraic data module, as demonstrated in Section 4 (final verdicts only), and Appendices A (preliminary verdicts only) and B (the full TTCN mechanism).

Please note that tests need not be deterministic, nor should the combinations of a test and an implementation under test (IUT) be deterministic. Yet we expect that typical tests are written in such a style that the TSF text exorcizes much of the non-determinism in the IUT. In many cases (as in Section 6.2), the combination of a test and a correct IUT will lead to a unique verdict (which must be \text{pass}). Even if the outcome is not unique, a
correct test when combined with a correct IUT may lead only to values pass or inconclusive.

In this paper we have focused on semantical issues, interpreting things in process algebra terms. This is because we believe that it is of prime importance to have the semantics of a new language as clear as possible. Of course we considered adding more features to the language or making it more TTCN-like, but we restricted ourselves to two new language constructs only (\(\rightarrow\) and verdict) in order to avoid additional semantic complications.

In practice, one might consider simulating tests together with specifications of correct and incorrect IUTs rather than performing a manual analysis in process algebra terms. The following tools could be useful: a simulator (in combination with PSF), and translators from and to TTCN. Typically one would develop tests in TTCN, translate them to TSF and check them against PSF specifications of the IUT. Of course one can also work the other way around: start from TSF and derive the TTCN from that, or avoid TTCN altogether. In this way TSF has the same relation to PSF as the relation between TTCN and SDL. For TTCN some tools available, notably the ITEX compiler of Telelogic, who make also an SDL design tool SDT.

Arie van Deursen has already constructed a syntax-directed editor and checker using the ASF+SDF toolset ([10], [11]). This revealed one priority conflict, which was easily resolved. No other difficulties in the concrete syntax have been encountered and the generated tool has already been used to check small examples, including the example of Section 6. The various data modules describing verdicts have also been typechecked using the PSF toolset.

The newly introduced operator \(\rightarrow\) deserves further study, because we only gave one model, and it is interesting to have other and more abstract models.

Acknowledgements: The authors wish to thank Jos Baeten, Jan Bergstra, Sjouke Mauw, Hans Mulder, Yat Man Lau, Jennifer Witham and Arie van Deursen for the discussions and the help that contributed to the work presented in this paper.

References


A Preliminary verdicts

These are verdicts like TTCN's preliminary verdicts. They are useful when employing an incremental approach to establishing the test result. No final verdicts or errors verdicts are needed. There is an ordering principle: a verdict can never get better than it already was, e.g. a preliminary fail verdict can never be changed in a preliminary pass verdict.

```plaintext
data module PreVerdict
begin
  exports
    begin
    sorts
      VERDICT
    functions
      none : -> VERDICT
      update : VERDICT # VERDICT -> VERDICT
      pass : -> VERDICT
      fail : -> VERDICT
      inconclusive : -> VERDICT
    end
  variables
```

362
\[
\begin{align*}
&\text{The full TTCN set of verdicts} \\
&\text{These are verdicts like TTCN's final and preliminary verdicts. The idea is} \\
&\text{that a final verdict (pass, fail, inconclusive) can only be given once. Therefore} \\
&\text{updating them leads to an error. The preliminary verdicts (pre-pass, pre-fail,} \\
&\text{pre-inconclusive) can be updated however, provided this updating is done in} \\
&\text{a consistent way.}
\end{align*}
\]
\[ r : \rightarrow \text{VERDICT} \]

**equations**

**-- getting started**

\[ [00] \text{update}(r,\text{none}) = r \]
\[ [01] \text{update}(\text{none},r) = \text{error} \]

**-- propagating errors**

\[ [02] \text{update}(r,\text{error}) = \text{error} \]
\[ [03] \text{update}(\text{error},r) = \text{error} \]

**-- processing preliminary verdicts**

\[ [04] \text{update}(\text{pre-pass},\text{pre-pass}) = \text{pre-pass} \]
\[ [05] \text{update}(\text{pre-pass},\text{pre-fail}) = \text{pre-fail} \]
\[ [06] \text{update}(\text{pre-pass},\text{pre-inconclusive}) = \text{pre-inconclusive} \]
\[ [07] \text{update}(\text{pre-pass},\text{pass}) = \text{error} \]
\[ [08] \text{update}(\text{pre-pass},\text{fail}) = \text{error} \]
\[ [09] \text{update}(\text{pre-pass},\text{inconclusive}) = \text{error} \]

\[ [10] \text{update}(\text{pre-fail},\text{pre-pass}) = \text{pre-fail} \]
\[ [11] \text{update}(\text{pre-fail},\text{pre-fail}) = \text{pre-fail} \]
\[ [12] \text{update}(\text{pre-fail},\text{pre-inconclusive}) = \text{pre-fail} \]
\[ [13] \text{update}(\text{pre-fail},\text{pass}) = \text{error} \]
\[ [14] \text{update}(\text{pre-fail},\text{fail}) = \text{error} \]
\[ [15] \text{update}(\text{pre-fail},\text{inconclusive}) = \text{error} \]

\[ [16] \text{update}(\text{pre-inconclusive},\text{pre-pass}) = \text{pre-inconclusive} \]
\[ [17] \text{update}(\text{pre-inconclusive},\text{pre-fail}) = \text{pre-fail} \]
\[ [18] \text{update}(\text{pre-inconclusive},\text{pre-inconclusive}) = \text{pre-inconclusive} \]
\[ [19] \text{update}(\text{pre-inconclusive},\text{pass}) = \text{error} \]
\[ [20] \text{update}(\text{pre-inconclusive},\text{fail}) = \text{error} \]
\[ [21] \text{update}(\text{pre-inconclusive},\text{inconclusive}) = \text{error} \]

**-- processing final verdicts**

\[ [22] \text{update}(\text{pass},\text{pre-pass}) = \text{pass} \]
\[ [23] \text{update}(\text{pass},\text{pre-fail}) = \text{error} \]
\[ [24] \text{update}(\text{pass},\text{pre-inconclusive}) = \text{error} \]
\[ [25] \text{update}(\text{pass},\text{pass}) = \text{error} \]
\[ [26] \text{update}(\text{pass},\text{fail}) = \text{error} \]
\[ [27] \text{update}(\text{pass},\text{inconclusive}) = \text{error} \]

\[ [28] \text{update}(\text{fail},\text{pre-pass}) = \text{fail} \]
\[ [29] \text{update}(\text{fail},\text{pre-fail}) = \text{fail} \]
\[ [30] \text{update}(\text{fail},\text{pre-inconclusive}) = \text{fail} \]
\[ [31] \text{update}(\text{fail},\text{pass}) = \text{error} \]
\[32\] update(fail,fail) = error
\[33\] update(fail,inconclusive) = error

\[34\] update(inconclusive,pre-pass) = inconclusive
\[35\] update(inconclusive,pre-fail) = error
\[36\] update(inconclusive,pre-inconclusive) = inconclusive
\[37\] update(inconclusive,pass) = error
\[38\] update(inconclusive,fail) = error
\[39\] update(inconclusive,inconclusive) = error

end Verdict
Empty Interworkings and Refinement
Semantics of Interworkings Revised

S. Mauw, M.A. Reniers

Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.
sjouke@win.tue.nl, michelr@win.tue.nl

Abstract

The semantics for Interworkings from [MvWW93] does not give a proper meaning to empty entities and empty Interworkings. Furthermore, the process algebra considered has to be extended in order to define refinement of Interworkings. For these purposes we give a revision and extension of the semantics.

1 Introduction

Interworkings are used for the graphical presentation of system traces. An Interworking describes the communication behaviour of system components. Interworkings are similar to Message Sequence Charts ([IT94]), which are standardized by the International Telecommunication Union (ITU). The main difference is that Interworkings describe synchronous communication, whereas Message Sequence Charts describe asynchronous communication.

A first proposal for the syntax and semantics of Interworkings is given in [MvWW93] and [MvW93] contains a description of a tool set for Interworkings. The semantics are given via a translation into process algebra ([BK84a, BK84b, BV95, BW90]). Communications are translated into atomic actions and two composition operators are defined: the interworking sequencing \((\prec_w)\) for vertical composition and the interworking merge \((|_w)\) for horizontal composition. In [vdBG95] Van den Brink and Griffioen describe an extension of Interworkings with discrete absolute time features by labelling actions with a time stamp and by labelling actions with a discrete time interval.

An Interworking consists of entities, represented by vertical axes, and messages, represented by horizontal arrows. The intuition behind the semantics is as follows. If two messages share an entity, the highest drawn message is executed first. Two messages which are not ordered in this way directly (i.e. via a shared entity) or indirectly (via a number of communications) may be executed in any order (see e.g. Figure 2D in which only one execution order is allowed: \(m_1, m_2, m_3\)). This is expressed formally in the definition of the interworking sequencing operator. The interworking merge operator is explained below.

Although the semantics in [MvWW93] are consistent, we are not completely satisfied with it, especially with respect to empty entities and empty Interworkings. These notions imply introduction of the so-called empty process into the process algebra. This extended process algebra is also needed if we define refinement as introduced in [MvWW92]. The problem encountered in the semantics of [MvWW93] with respect to empty entities is the following. Consider the Interworkings from Figure 1. The difference between the Interworkings \(A\) and \(B\) is that \(B\) contains an entity \(s\) while \(A\) does not. In fact Interworking \(B\) specifies that there can be no communication with entity \(s\), whereas Interworking \(A\) does not say anything about the interaction with \(s\).

*Part of this research is funded by Philips Research Laboratories Eindhoven.
If we merge Interworkings $A$ and $C$ (from Figure 1), we first calculate the set of common entities of $A$ and $C$, which is $\{q, r\}$. The result is such that all communications between entities within this set are present in both $A$ and $C$. The result is Interworking $D$ shown in Figure 2.

In Interworking $B$ entity $s$ is present, although not participating in any communication action. If we merge Interworkings $B$ and $C$, the common entities from $B$ and $C$ are $\{q, r, s\}$. Thus $B$ and $C$ have to comply also with respect to the communication of $m_3$ from $r$ to $s$. Since $B$ does not have this communication, a deadlock results as depicted by two horizontal bars in the Interworking $E$ from Figure 2.

We conclude that intuitively the merge of $A$ and $C$ is different from the merge of $B$ and $C$. In the semantics of [MvWW93], however, $A$ and $B$ have exactly the same interpretation. The empty entity $s$ is simply neglected, so $A \|_{\text{int}} C$ and $B \|_{\text{int}} C$ are equal. We solve this difference in intuition and semantics by enriching the process which is the interpretation of a given Interworking with explicit information about the entities which are present.

The second reason for revising the semantics of Interworkings is that we give a formal definition of the notion of refinement. An Interworking is a refinement of another Interworking if they have the same behaviour after aggregation of a number of entities into one single entity. This implementation relation is very useful for expressing levels of abstraction and thus allows for a top-down specification style.

This paper is organized as follows. In Section 2 we give a formal definition of the interworking operators. Section 3 contains several properties of the interworking sequencing and interworking merge operators. The Interworking refinement is defined in Section 4.

**Acknowledgements** We would like to thank Thijs Winter (Philips Hilversum) and Mark van Wijk for cooperating on a preliminary, although never published, version of Interworkings with refinement. Furthermore, we thank Jos Baeten, Twan Basten and Hans Mulder (all Eindhoven University of Technology) for their criticism. Their comments have been very helpful in obtaining the results we did. The anonymous reviewers and Jan Gerben Wijnstra (Philips Research Laboratories Eindhoven) are acknowledged for their comments on a preliminary version of this paper.
We are grateful to Loe Feijs (Philips Research Laboratories Eindhoven) for his comments on the refinement of Interworkings.

2 Process Algebra for Interworkings

In this section we will extend the process algebra $BPA_{iw}(A, EID, E)$ from [MvWW93] with the empty process ($\varepsilon$). The parameters $A$, $EID$ and $E$ are the set of atomic actions, a universe of entity identifiers and a mapping from atomic actions to entity identifiers, respectively. We start by giving the process algebra $BPA_{\delta, \varepsilon}(A)$ from [BW90, BV95]. This process algebra is extended to the process algebra $IWD_{\varepsilon}(A, EID, E)$, i.e., Process Algebra for Interworking Diagrams, with the operator interworking sequencing ($@w$) and some auxiliary operators ($L_{tw}$, $R_{tw}$, and $\sqrt{\varepsilon}$). Then we extend $IWD_{\varepsilon}(A, EID, E)$ with the $E$-interworking merge ($\|Ew$) and some auxiliary operators ($|Ew$, $\|E\varepsilon$). The resulting process algebra is called $IWE_{\varepsilon}(A, EID, E)$. Finally, we extend the processes representing Interworkings with an additional label representing the entities from which the Interworking is built. This process algebra will be called $IWE_{\varepsilon}(A, EID, E)$, i.e. Process Algebra for Interworkings with (empty) entities. We give for each of the process algebras a structured operational semantics in the style of Plotkin [PloS1, PloS3].

In the case of Interworkings, the three parameters of the algebraic theories are instantiated as follows: $A = \{c(p, q, m) \mid p, q \in EID, m \in MID\}$, where MID is some set of message identifiers, EID is some set of entity identifiers and $E$ is a function which associates to each atomic action from $A$ a set of entity identifiers; $E(c(p, q, m)) = \{p, q\}$. In fact, with Interworkings, there are two parameters: the set of entity identifiers $EID$ and the set of message identifiers $MID$, and a constructor function for the atomic actions $c : EID \times EID \times MID \to A$. We have chosen for the approach with three parameters to cover applications where an entity function must be defined explicitly, because it can not be obtained from the atomic actions.

2.1 Basic Process Algebra with Deadlock and Empty Process

We will give a brief introduction to the process algebra $BPA_{\delta, \varepsilon}(A)$ [BV95, BW90]. This process algebra will be our starting point towards the more complex algebras which are introduced in the following sections. The parameter $A$ of the process algebra represents the set of atomic actions. Besides the atomic actions from the set $A$ the process algebra has the additional constants $\delta$ and $\varepsilon$, which represent deadlock and the empty process respectively. The process deadlock is incapable of executing any actions and can moreover not terminate successfully. The empty process can also execute no actions, but it terminates successfully. The set of all constants of the process algebra is denoted by $A_{\delta, \varepsilon}$.

From these constants more complex processes can be built by using the operators $+$ and $\cdot$. The $+$ is called alternative composition and $\cdot$ is called sequential composition. The process $x + y$ can execute either process $x$ or process $y$, but not both. The process $x \cdot y$ starts executing process $x$, and upon termination thereof starts the execution of process $y$. These operators are axiomatized by the axioms from Table 1. In these axioms the variables $x$, $y$ and $z$ denote arbitrary processes.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y = y + x$</td>
<td>A1</td>
</tr>
<tr>
<td>$A6$</td>
<td></td>
</tr>
<tr>
<td>$(x + y) + z = x + (y + z)$</td>
<td>A2</td>
</tr>
<tr>
<td>$A7$</td>
<td></td>
</tr>
<tr>
<td>$x + z = x$</td>
<td>A3</td>
</tr>
<tr>
<td>$A8$</td>
<td></td>
</tr>
<tr>
<td>$(x + y) \cdot z = x \cdot z + y \cdot z$</td>
<td>A4</td>
</tr>
<tr>
<td>$A9$</td>
<td></td>
</tr>
<tr>
<td>$(x \cdot y) \cdot z = x \cdot (y \cdot z)$</td>
<td>A5</td>
</tr>
</tbody>
</table>

Table 1: Axioms of $BPA_{\delta, \varepsilon}(A)$
In order to reduce the number of brackets in processes we have the following priorities on operators: \( \cdot \) binds stronger than all other operators and \(+\) binds weaker than all other operators.

To the process algebra \( BPA_{i,e}(A) \) we associate a structured operational semantics in the form of the term deduction system \( T(BPA_{i,e}(A)) \) in Table 2. For the deduction rules in this table we require that \( a \in A \) and that \( x, y, \) and \( z \) are arbitrary processes. A deduction rule is of the form \( \frac{H}{C} \) where \( H \) is a set of hypotheses and \( C \) is the conclusion. The formula \( x \xrightarrow{a} x' \) expresses that the process \( x \) can perform an action \( a \) and thereby evolves into the process \( x' \) and the formula \( x \mid \) expresses that process \( x \) has an option to terminate immediately and successfully. For a formal definition of term deduction systems we refer to [BV93].

<table>
<thead>
<tr>
<th>( x \mid )</th>
<th>( y \mid )</th>
<th>( x \mid, y \mid )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon \mid )</td>
<td>( x + y \mid )</td>
<td>( x + y \mid )</td>
</tr>
<tr>
<td>( x \xrightarrow{a} x' )</td>
<td>( y \xrightarrow{a} y' )</td>
<td>( x \xrightarrow{a} x' \mid y \xrightarrow{a} y' )</td>
</tr>
<tr>
<td>( x \mid, y \mid )</td>
<td>( x \mid, y \mid )</td>
<td>( x \mid, y \mid )</td>
</tr>
</tbody>
</table>

Table 2: Structured Operational Semantics of \( BPA_{i,e}(A) \)

Finally we would like to mention the following well-known result from literature (e.g. [BW90]): The process algebra \( BPA_{i,e}(A) \) is a sound and complete axiomatization of bisimulation equivalence, notation \( \equiv \), on the closed \( BPA_{i,e}(A) \) terms. This result will be used in the following sections when relating the extended process algebras to \( BPA_{i,e}(A) \).

2.2 Axiomatization of Interworking Sequencing

In this section we will extend the process algebra \( BPA_{i,e}(A) \) from the previous section with an entity function on the atomic actions from \( A \) and the interworking sequencing operator \( \mathcal{Q}_w \). The resulting process algebra is denoted by \( IWD_e(A,EID,E) \). The entity function \( E : A \rightarrow P(EID) \) associates to every atomic action from the set \( A \) a set of entity identifiers from \( EID \). Intuitively the entities of the atomic action \( a \) are the functional blocks on which the atomic action is defined/executed; these entities are called active entities. The universe of entity identifiers \( EID \) and the entity function \( E \) are considered parameters of the process algebra.

Based on the entity function \( E \) on atomic actions we can associate to every closed term over the signature of the process algebra a set of entities; these are the entities to which the atomic actions of the process refer. This is done by extending the entity function on atomic actions to an entity function \( E \) on process terms. This extension is, for \( a \in A \) and \( x \) and \( y \) arbitrary processes, defined in Table 3. For the atomic actions from \( A \), viewed as a process, the entities are given already by the entity function on atomic actions. For the atomic actions \( \varepsilon \) and \( \delta \) we take \( E(\varepsilon) = \emptyset \) and \( E(\delta) = \emptyset \). For more complex processes the entities are obtained from the atomic actions the process is built from.

<table>
<thead>
<tr>
<th>( E(\varepsilon) )</th>
<th>( E(\delta) )</th>
<th>( E(a \cdot x) )</th>
<th>( E(x + y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( = \emptyset )</td>
<td>( = \emptyset )</td>
<td>( = E(a) \cup E(x) )</td>
<td>( = E(x) \cup E(y) )</td>
</tr>
</tbody>
</table>

Table 3: Active Entities of an Interworking
The interworking sequencing of two processes \( x \) and \( y \) is their parallel execution with the restriction that the right-hand side process may execute an action only if the entities of that action are disjoint from the entities of the left-hand side process. The interworking sequencing operator is similar to the weak sequential composition operator from [RW94]. As a starting point for the axiomatization we take the axioms for the interworking sequencing as they are given in [MvWW93]. To obtain an axiomatization of the interworking sequencing operator in the context of empty Interworkings an approach is followed which compares easily with the step from the free merge in a setting without the empty process to a free merge in a setting with the empty process [BW90].

The axiomatization of \( q_w \) as presented in [MvWW93] uses the two auxiliary operators \( L_{lw} \) and \( R_{lw} \). The process \( x L_{lw} y \) behaves like the process \( x q_w y \) with the restriction that the first action to be executed must originate from process \( x \). The process \( x R_{lw} y \) also behaves like the process \( x q_w y \) but this time with the restriction that the first action to be executed must be from process \( y \).

Intuitively, we want \( \varepsilon \) to be a unit for the interworking sequencing, i.e. \( \varepsilon q_w \varepsilon = x q_w \varepsilon = x \). In particular we also want to have that \( \varepsilon q_w \varepsilon = \varepsilon \). The interpretation of \( x L_{lw} y \) is that the process \( x \) is forced to do the first step. Since \( \varepsilon \) is unable to do any step, it seems plausible to define \( \varepsilon L_{lw} y = \delta \). Consequently, we also define \( x R_{lw} \varepsilon = \delta \). If we apply this in the definition of the sequencing operator as given in [MvWW93] we get \( \varepsilon q_w \varepsilon = \varepsilon L_{lw} \varepsilon + \varepsilon R_{lw} \varepsilon = \delta + \delta = \delta \). This is not what we want and therefore we need the additional operator \( \sqrt{\varepsilon} \) as given in Table 4. This operator has also been used by Baeten and Weijland [BW90] in axiomatizing the free merge in a process algebra containing the empty process. The definition of the interworking sequencing operator is given in Table 4.

\[
\begin{align*}
  x q_w y &= x L_{lw} y + x R_{lw} y + \sqrt{x} \cdot \sqrt{y} & (S1) \\
  \varepsilon L_{lw} x &= \delta & (LS1) \\
  \delta L_{lw} x &= \delta & (LS2) \\
  a \cdot x L_{lw} y &= a \cdot (x q_w y) & (LS3) \\
  (x + y) L_{lw} z &= x L_{lw} (z + y) & (LS4) \\
  x R_{lw} \varepsilon &= \delta & (RS1) \\
  x R_{lw} \delta &= \delta & (RS2) \\
  x R_{lw} a \cdot y &= a \cdot (x q_w y) & \text{if } E(a) \cap E(x) = \emptyset & (RS3) \\
  x R_{lw} a \cdot y &= \delta & \text{if } E(a) \cap E(x) \neq \emptyset & (RS4) \\
  x R_{lw} (y + z) &= x R_{lw} y + x R_{lw} z & (RS5) \\
  \sqrt{\varepsilon} &= \varepsilon & (T1) \\
  \sqrt{\delta} &= \delta & (T2) \\
  \sqrt{a \cdot x} &= \delta & (T3) \\
  \sqrt{x + y} &= \sqrt{x} + \sqrt{y} & (T4)
\end{align*}
\]

Table 4: Axioms for interworking sequencing

The structured operational semantics of the interworking sequencing and of the auxiliary operators is given in Table 5. The term deduction system \( T(IWD,(A, EID, E)) \) consists of the deduction rules of \( T(BPA,(A, E)) \) and the deduction rules of Table 5.

Next, we will formulate some interesting theorems concerning this process algebra. These theorems relate the process algebra \( IWD,(A, EID, E) \) to the process algebra \( BPA,(A) \) and to the structured operational semantics as given by the term deduction systems.

**Theorem 2.2.1 (Congruence)** Bisimulation equivalence is a congruence for the function symbols in the signature of \( IWD,(A, EID, E) \).

**Proof** It is straightforward to verify that the deduction rules of the term deduction system which
Table 5: Structured Operational Semantics of interworking sequencing and auxiliary operators

<table>
<thead>
<tr>
<th>Rule</th>
<th>Left-hand Side</th>
<th>Right-hand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{x \downarrow, y \downarrow}{x \sqcup y \downarrow}$</td>
<td>$x \sqcup y \rightarrow x' \sqcup y$</td>
<td>$y \rightarrow y', E(a) \cap E(x) = \emptyset$</td>
</tr>
<tr>
<td>$\frac{x \sqcup y \downarrow}{\sqrt(x) \downarrow}$</td>
<td>$x \sqcup y \rightarrow x' \sqcup y$</td>
<td>$x \sqcup y \rightarrow x \sqcup y'$</td>
</tr>
</tbody>
</table>

consists of the deduction rules from Tables 2 and 5 are in path format. By the congruence theorem from Baeten and Verhoef [BV93] it follows immediately that bisimulation is a congruence on the closed $IWD_e(A, EID, E)$ terms.

**Theorem 2.2.2 (Soundness)** The process algebra $IWD_e(A, EID, E)$ is a sound axiomatization of bisimulation equivalence on closed $IWD_e(A, EID, E)$ terms.

**Proof** Due to the fact that bisimulation is a congruence, we only have to verify the soundness of each axiom. For the axioms SI, LS3,4, and RS3,5 we relate the left-hand side to the right-hand side and we add the diagonal (i.e. we relate each term to itself). For the other axioms we only relate the left-hand side to the right-hand side.

**Theorem 2.2.3 (Conservativity)** The process algebra $IWD_e(A, EID, E)$ is a conservative extension of the process algebra $BPA_{\delta,e}(A)$.

**Proof** The proof of this theorem uses the approach of Verhoef [Ver94]. The theorem follows from the following observations:

1) Bisimulation is definable in terms of predicate and relation symbols only,

2) the process algebra $BPA_{\delta,e}(A)$ is a complete axiomatization of bisimulation equivalence on closed $BPA_{\delta,e}(A)$ terms (see [BV95, BW90]),

3) the process algebra $IWD_e(A, EID, E)$ is a sound axiomatization of bisimulation equivalence on closed $IWD_e(A, EID, E)$ terms (see Theorem 2.2.2),

4) the term deduction system $T(BPA_{\delta,e}(A))$ is pure$^1$, well-founded$^2$ and in path format, and

5) the term deduction system $T(IWD_e(A, EID, E))$ is in path format$^3$.

**Theorem 2.2.4 (Elimination)** The process algebra $IWD_e(A, EID, E)$ has the elimination property for the process algebra $BPA_{\delta,e}(A)$.

**Proof** The term rewrite system associated with the axioms A3-A7 from Table 1 and the axioms from Tables 3 and 4 and the additional rewriting rules from Table 6 is strongly normalizing. This can be proven with the method of the lexicographical path ordering [KLS0, Klo92]. Note that the additional rewriting rules are, for closed terms, derivable from the axioms of $IWD_e(A, EID, E)$.

If we additionally show that every normal form of the closed terms is in fact a closed $BPA_{\delta,e}(A)$ term then the theorem follows easily. Thereto, suppose that $s$ is a normal form with respect to the term rewrite system and suppose that $s$ is not a closed $BPA_{\delta,e}(A)$ term. Then $s$ must contain at

---

$^1$For a definition of pure term deduction systems see [BV93].

$^2$For a definition of well-founded term deduction systems see [BV93].

$^3$For a definition of the path format see [BV93].
least one occurrence of the operators $q_w$, $L_{q_w}$, $R_{q_w}$, or $\sqrt{\cdot}$. Take a smallest subterm of $s$ which is headed by one of these operators. In any case it follows that the operands of this operator are closed $BPA_{\text{A},\text{E}}(A)$ terms. From that it is easily seen that a rewrite rule must be applicable, which contradicts the assumptions. Therefore, we conclude that every normal form of a closed $IWD_t(A, EID, E)$ term is a closed $BPA_{\text{A},\text{E}}(A)$ term.

**Theorem 2.2.5 (Completeness)** The process algebra $IWD_t(A, EID, E)$ is a complete axiomatization of bisimulation equivalence on closed $IWD_t(A, EID, E)$ terms.

**Proof** By the General Completeness Theorem of Verhoef [Ver94], the completeness of the process algebra $IWD_t(A, EID, E)$ follows immediately from the propositions which are used in the proof of Theorem 2.2.3 and the fact that $IWD_t(A, EID, E)$ has the elimination property for $BPA_{\text{A},\text{E}}(A)$ (see Theorem 2.2.4).

### 2.3 Axiomatization of Interworking Merge

In this section we will extend the process algebra $IWD_t(A, EID, E)$ from the previous section with the interworking merge operator ($\parallel_{\text{mw}}$). The resulting process algebra is called $IWE_t(A, EID, E)$. Thereto, we first describe $IWE_t(A, EID, E)$, the extension of $IWD_t(A, EID, E)$ with the $E$-interworking merge operator ($\parallel_{\text{E}}$). Technically speaking, we can axiomatize the interworking merge without using the $E$-interworking merge. But, to stay as close as possible to the existing axiomatization of the interworking merge, we use the $E$-interworking merge. After that we introduce tuples of process terms and entity sets. On this new structure we define the interworking merge operator.

As we have shown in the introduction there is a problem with axiomatizing the interworking merge operator. From an expression representing an Interworking it is not possible to determine the empty entities, since the Interworking can have empty entities which are not represented in the atomic actions describing the Interworking. In this section we will solve this problem by associating to every closed term of the process algebra a label denoting the entities which are present. With that additional information it is straightforward to give an axiomatization of the interworking merge. As was done in [MvWW93] the interworking merge is expressed in terms of the $E$-interworking merge operator and the common entities of the operands.

The axiomatization of the $S$-interworking merge as presented in [MvWW93] uses the auxiliary operators left $S$-interworking merge $\parallel_{\text{lw}}^S$ and synchronization interworking merge $\parallel_{\text{lw}}^S$ with $S$ a set of atomic actions. We will use similar auxiliary operators only now labelled with a set of entities in stead of a set of atomic actions. This set represents the entities on which communication actions must synchronize. The process $x \parallel_{\text{lw}}^E y$ is the parallel execution of the processes $x$ and $y$ with the restriction that the processes must synchronize on all atomic actions which are defined on entities from the set $E$. The process $x \parallel_{\text{lw}}^E y$ behaves like the process $x \parallel_{\text{lw}}^E y$ with the restriction that the first action must come from process $x$ and that action does not have to synchronize with an action from $y$. The process $x \parallel_{\text{lw}}^E y$ behaves as the process $x \parallel_{\text{lw}}^E y$ with the restriction that the first action to be executed must be a synchronization.
Since we have defined the entities of the empty process to be the empty set, it seems plausible that we define $x \|_{Iw}^E \varepsilon = \varepsilon$, and in particular $\varepsilon \|_{Iw}^E x = x \|_{Iw}^E \varepsilon = \varepsilon$. With respect to the main axiom for the interworking merge from [MvWW93]: $x \|_{Iw}^S y = x \|_{Iw}^S y + y \|_{Iw}^S x + x \|_{Iw}^S y$, we derive $\varepsilon \|_{Iw}^S \varepsilon = \varepsilon \|_{Iw}^E \varepsilon + \varepsilon \|_{Iw}^E \varepsilon + \varepsilon \|_{Iw}^E \varepsilon = \varepsilon + \varepsilon + \varepsilon = \varepsilon$. In terms of Interworkings this means that two empty Interworkings are not consistent which is not what we want. Therefore, we need to redefine the interworking merge operator in a way similar to the definition of the sequencing operator. The new axioms for the $E$-interworking merge operator are given in Table 7. Recall that the axioms for the termination operator are given in Table 4.

\[
\begin{align*}
x \|_{Iw}^E y & = x \|_{Iw}^E y + y \|_{Iw}^E x + x \|_{Iw}^E y + \sqrt{x} \cdot \sqrt{y} & \text{EM1} \\
\varepsilon \|_{Iw}^E x & = \delta & \text{LEM1} \\
\delta \|_{Iw}^E x & = \delta & \text{LEM2} \\
a \cdot x \|_{Iw}^E y & = a \cdot (x \|_{Iw}^E y) & \text{LEM3} \\
a \cdot x \|_{Iw}^E y & = \delta & \text{LEM4} \\
(x + y) \|_{Iw}^E z & = x \|_{Iw}^E z + y \|_{Iw}^E z & \text{LEM5} \\
\varepsilon \|_{Iw}^E x & = \delta & \text{SEM1} \\
\delta \|_{Iw}^E x & = \delta & \text{SEM2} \\
x \|_{Iw}^E \varepsilon & = \delta & \text{SEM3} \\
x \|_{Iw}^E \delta & = \delta & \text{SEM4} \\
a \cdot x \|_{Iw}^E b \cdot y & = a \cdot (x \|_{Iw}^E y) & \text{SEM5} \\
a \cdot x \|_{Iw}^E b \cdot y & = \delta & \text{SEM6} \\
(x + y) \|_{Iw}^E z & = x \|_{Iw}^E z + y \|_{Iw}^E z & \text{SEM7} \\
x \|_{Iw}^E (y + z) & = x \|_{Iw}^E y + x \|_{Iw}^E z & \text{SEM8}
\end{align*}
\]

Table 7: Axioms of $E$-interworking merge

Next, we present a structured operational semantics for the operators which are introduced in this section. The term deduction system $T(IW_{\varepsilon}(A, EID, E))$ consists of the deduction rules of $T(IWD_{\varepsilon}(A, EID, E))$ and the deduction rules of Table 8.

\[
\begin{align*}
x \perp, y \perp \\
& \quad x \|_{Iw}^E y \perp \\
x \overset{a}{\rightarrow} x', E(a) \not\subseteq E \\
& \quad x \|_{Iw}^E y \overset{a}{\rightarrow} x' \|_{Iw}^E y' \\
x \overset{a}{\rightarrow} x', E(a) \subseteq E \\
& \quad x \|_{Iw}^E y \overset{a}{\rightarrow} x' \|_{Iw}^E y' \\
& \quad y \overset{a}{\rightarrow} y', E(a) \not\subseteq E \\
& \quad x \|_{Iw}^E y \overset{a}{\rightarrow} x \|_{Iw}^E y' \\
x \overset{a}{\rightarrow} x', E(a) \subseteq E \\
& \quad x \|_{Iw}^E y \overset{a}{\rightarrow} x' \|_{Iw}^E y' \\
& \quad y \overset{a}{\rightarrow} y', E(a) \subseteq E \\
& \quad x \|_{Iw}^E y \overset{a}{\rightarrow} x \|_{Iw}^E y' \\
x \overset{a}{\rightarrow} x', y \overset{a}{\rightarrow} y', E(a) \subseteq E \\
& \quad x \|_{Iw}^E y \overset{a}{\rightarrow} x' \|_{Iw}^E y'
\end{align*}
\]

Table 8: Structured Operational Semantics of $E$-interworking merge and auxiliary operators

Before we turn to the interworking merge operator we list results similar to those of the previous section for the process algebra $IW_{\varepsilon}(A, EID, E)$. The proofs are omitted.

**Theorem 2.3.1 (Congruence)** Bisimulation equivalence is a congruence for the function symbols in the signature of $IW_{\varepsilon}(A, EID, E)$.
Theorem 2.3.2 (Soundness) The process algebra $\mathcal{W}(A, E, E)$ is a sound axiomatization of bisimulation equivalence on closed $\mathcal{W}(A, E, E)$ terms.

Theorem 2.3.3 (Conservativity) The process algebra $\mathcal{W}(A, E, E)$ is a conservative extension of the process algebra $\mathcal{W}(A, E, E)$.

Theorem 2.3.4 (Elimination) The process algebra $\mathcal{W}(A, E, E)$ has the elimination property for the process algebra $\mathcal{W}(A, E, E)$.

Theorem 2.3.5 (Completeness) The process algebra $\mathcal{W}(A, E, E)$ is a complete axiomatization of bisimulation equivalence on closed $\mathcal{W}(A, E, E)$ terms.

Now that we have given the axioms and structured operational semantics of the $E$-interworking merge we will discuss the interworking merge operator. The interworking merge of two processes is their parallel execution with the restriction that the processes must synchronize on all atomic actions which are defined on the common entities of the processes. For the interworking merge operator it is necessary to determine the common entities of the operands. The entities of an operand can not be obtained from the process term representing it, since empty entities are not represented in the process term. Therefore, we label every process term by a set of entity names over $E$. For an Interworking $x$, this set together with the active entities of $x$ (i.e. $E(x)$) represents the entities of the interworking (including the empty entities). An interworking with a dynamical behaviour denoted by $x$ over the entities from $E$ is denoted by $(x, E)$. We do not require that the set $E$ consists of all entities of the interworking; instead we require that the entities of the interworking are given by $E \cup E(x)$. The reason for this choice is that in this setting every tuple from which the dynamical behaviour describes an interworking, can be interpreted as an Interworking in the extended semantics. Such a tuple $(x, E)$ will be called a labelled process. On labelled processes we define the operators $+, \cdot$, $\parallel_{lw}$, $\parallel_{lt}$, and $\parallel_{ll}$. The set of all labelled processes is called $LP$. The axioms for labelled processes are, for $x, y$ processes, $E, E_1$, and $E_2 \subseteq E$, given in Table 9.

Table 9: Extension to labelled processes

<table>
<thead>
<tr>
<th>$(x, E)$</th>
<th>$(x, E \cup E(x))$</th>
<th>if $E(x) \subseteq E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x, E_1) \oplus (y, E_2)$</td>
<td>$(x \oplus y, E_1 \cup E_2)$</td>
<td>if $E(x) \subseteq E_1 \land E(y) \subseteq E_2$</td>
</tr>
<tr>
<td>$(x, E_1) \parallel_{lw} (y, E_2)$</td>
<td>$(x \parallel_{lw} y, E_1 \cup E_2)$</td>
<td>if $E(x) \subseteq E_1 \land E(y) \subseteq E_2$</td>
</tr>
</tbody>
</table>

Example 2.3.6 The labelled processes associated to the Interworkings $A$, $B$, and $C$ from Figure 1 are the following.

$$A = \langle c(p, q, m1) qwc(q, r, m2), \{p, q, r\} \rangle$$
$$B = \langle c(p, q, m1) qwc(q, r, m2), \{p, q, r, s\} \rangle$$
$$C = \langle c(q, r, m2) qwc(r, s, m3), \{q, r, s\} \rangle$$

Then we have the following computations for $A \parallel_{lw} C$ and $B \parallel_{lw} C$.

$$A \parallel_{lw} C = \langle c(p, q, m1) qwc(q, r, m2), \{p, q, r\} \parallel_{lw} c(q, r, m2) qwc(r, s, m3), \{q, r, s\} \rangle$$
$$= \langle c(p, q, m1) qwc(q, r, m2) \parallel_{lw} c(q, r, m2) qwc(r, s, m3), \{p, q, r, s\} \rangle$$
$$= \langle c(p, q, m1) qwc(q, r, m2) qwc(r, s, m3), \{p, q, r, s\} \rangle$$
and
\[ B \parallel_{i_{w}} C = (c(p, q, m_1) q w c(q, r, m_2), \{ p, q, r, s \}) \parallel_{i_{w}} (c(q, r, m_2) q w c(r, s, m_3), \{ q, r, s \}) \]
\[ = (c(p, q, m_1) q w c(q, r, m_2) ||_{i_{w}} c(q, r, m_2) q w c(r, s, m_3), \{ p, q, r, s \}) \]
\[ = (c(p, q, m_1) q w c(q, r, m_2) q w \delta, \{ p, q, r, s \}) \]

Clearly, the expressions do not denote the same interworking in this extended setting. In other
words, the labelled processes are not entity bisimilar (see Definition 2.3.7).

Observe that the role of the empty process as a neutral element for the sequential
composition, the interworking sequencing and the interworking merge
is, with respect to closed labelled processes, taken over by the labelled process
\((\varepsilon, 0)\), i.e. \((\varepsilon, 0) \parallel_{i_{w}} x = x \parallel_{i_{w}} x = x \parallel_{i_{w}} (\varepsilon, 0)\). The role of deadlock as the left zero element for the sequential
composition operator is taken over by the labelled process
\((0, \varepsilon I)\), i.e. \((0, \varepsilon I) \parallel_{i_{w}} x = x \parallel_{i_{w}} x = (0, \varepsilon I)\). These properties of the labelled processes can all easily be derived from the axioms from Table 9
and the axioms for the operators in the process algebra.

Next, we define a structured operational semantics for labelled processes. The deduction
rules of the term deduction system \(T(IWE, (A, EID, E))\) are given by the deduction rules of
\(T(IWE, (A, EID, E))\) and the deduction rules from Table 10.

<table>
<thead>
<tr>
<th>( \frac{x \parallel_{i_{w}} y \parallel_{i_{w}}}{(x, E) \parallel_{i_{w}} (y, F)} )</th>
<th>( \frac{z \parallel_{i_{w}} x \parallel_{i_{w}}}{(x, E) \parallel_{i_{w}} (y, F) \parallel_{i_{w}} (z, E \cup F \cup E(x) \cup E(y))} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{(x, E) \parallel_{i_{w}} (y, F) \parallel_{i_{w}} (z, E \cup F \cup E(x) \cup E(y))}{x \parallel_{i_{w}} (y, F) \parallel_{i_{w}} (z, E \cup F \cup E(x) \cup E(y))} )</td>
<td>( \frac{(x, E) \parallel_{i_{w}} (y, F) \parallel_{i_{w}} (z, E \cup F \cup E(x) \cup E(y))}{x \parallel_{i_{w}} (y, F) \parallel_{i_{w}} (z, E \cup F \cup E(x) \cup E(y))} )</td>
</tr>
</tbody>
</table>

Table 10: Structured Operational Semantics of labelled processes

Definition 2.3.7 The closed LP terms \((s, E)\) and \((t, F)\) are entity bisimilar, 
\((s, E) \equiv (t, F)\), if and only if \(s \equiv t\) and \(E \cup E(x) = F \cup E(y)\).

Theorem 2.3.8 (Congruence) Entity bisimulation equivalence is a congruence for the function
symbols in the signature of \(IWE(A, EID, E)\) which are defined on LP terms.

Proof The theorem from [BV93] as used in the previous theorems on congruence is not applicable
in this case. This theorem is only formulated for strong bisimulation equivalence. Nevertheless
we will see that Theorem 2.3.8 is not too hard to prove. Suppose that \((x_1, E_1) \equiv (x_2, E_2)\)
and \((y_1, F_1) \equiv (y_2, F_2)\). Now we have to prove, for \(\oplus \in \{ +, \cdot, q_w, ||_{i_{w}}, \parallel_{i_{w}} \}\), that \((x_1, E_1) \oplus (y_1, F_1) \equiv (x_2, E_2) \oplus (y_2, F_2)\). From the assumptions we have the following: 
\(x_1 \equiv x_2, y_1 \equiv y_2, E_1 \cup E(x_1) = E_2 \cup E(x_2)\) and \(F_1 \cup E(y_1) = F_2 \cup E(y_2)\).

We will first verify the termination predicate. Suppose that \((x_1, E_1) \oplus (y_1, F_1) \parallel_{i_{w}} (y_2, F_2)\). According
to the deduction rules this is only the case if \(x_1 \parallel_{i_{w}} y_1 \parallel_{i_{w}} (y_2, F_2)\). From \(x_1 \parallel_{i_{w}} y_2 \parallel_{i_{w}} y_2\) we obtain \(x_2 \parallel_{i_{w}} y_2 \parallel_{i_{w}} (y_2, F_2)\). Using the same deduction rule as before we have \((x_2, E_2) \oplus (y_2, F_2) \parallel_{i_{w}} (y_2, F_2)\). In the other
direction the proof is analogous. Suppose that \((x_1, E_1) \parallel_{i_{w}} (y_1, F_1) \parallel_{i_{w}} (y_1, F_1)\). Then we have \((x_1, E_1) \parallel_{i_{w}} (y_2, F_2)\).
and \((y_1, F_1) \perp\). But then also \(x_1 \perp\) and \(y_1 \perp\). By the assumptions we then have \(x_2 \perp\) and \(y_2 \perp\). Then we can deduce \((x_2, E_2) \perp\) and \((y_2, E_2) \perp\) and from that \((x_2, E_2) \perp_{iw} (y_2, F_2) \perp\). In the other direction the proof is analogous.

Next, we will show that every step from the left-hand side can be mimicked by the right-hand side, and vice versa. Also we will show that the resulting labelled processes are entity bisimilar. Suppose that \((x_1, E_1) \perp (y_1, F_1) \perp (z, G)\). Inspection of the deduction rules gives us that we must have that \(x_1 \oplus y_1 \xrightarrow{a} z\) and \(G = E_1 \cup F_1 \cup E(x_1) \cup E(y_1)\). From this we get with the assumptions \(x_2 \oplus y_2 \xrightarrow{a} z\) and \(G = E_2 \cup F_2 \cup E(x_2) \cup E(y_2)\). Therefore, with the same deduction rule we deduce \((x_2, E_2) \perp (y_2, F_2) \perp (z, G)\). The proof in the other direction is analogous.

Suppose that \((x_1, E_1) \perp (y_1, F_1) \perp (z, G)\). We use the following abbreviations \(E'_1 = E_1 \cup E(x_1), F'_1 = F_1 \cup E(y_1)\) and \(F'_2 = F_2 \cup E(y_2)\). Then we must have \(x_1 \perp_{iw} E'_1 \cap F'_1 \perp y_1 \xrightarrow{a} z\) and \(G = E'_1 \cup F'_1\). By the assumptions we then also have \(x_2 \perp_{iw} F'_2 \perp y_2 \xrightarrow{a} z\) and \(G = E'_2 \cup F'_2\). From this we obtain \((x_2, E_2) \perp_{iw} (y_2, F_2) \perp (z, G)\). The proof in the other direction is analogous.

**Theorem 2.3.9 (Soundness)** The process algebra \(IWE_e(A, EID, E)\) is a sound axiomatization of bisimulation equivalence on closed \(IW_e(A, EID, E)\) terms. The process algebra \(IWE_e(A, EID, E)\) is a sound axiomatization of entity bisimulation on closed \(LP\) terms.

**Proof** For the first proposition see Theorem 2.3.2 and observe that we did not add any axioms relating closed \(IW_e(A, EID, E)\) terms. We will prove the second proposition. Since entity bisimulation is a congruence for the closed terms of \(LP\) we only have to show that the axioms are sound. For the first axiom relate the left-hand side to the right-hand side. For the other axioms we additionally relate each term to itself.

**Theorem 2.3.10 (Conservativity)** The process algebra \(IWE_e(A, EID, E)\) is a conservative extension of the process algebra \(IW_e(A, EID, E)\).

**Proof** The theorem follows from the following observations:

1) Bisimulation is definable in terms of predicate and relation symbols only,

2) \(IW_e(A, EID, E)\) is a complete axiomatization of bisimulation on closed \(IW_e(A, EID, E)\) terms (see Theorem 2.3.5),

3) \(IWE_e(A, EID, E)\) is a sound axiomatization of bisimulation on closed \(IW_e(A, EID, E)\) terms (see Theorem 2.3.9), and

4) \(T(IW_e(A, EID, E))\) is pure, well-founded and in path format, and

5) \(T(IWE_e(A, EID, E))\) is in path format.

**Definition 2.3.11** Basic \(LP\) terms are defined inductively as follows:

1) if \(s\) is a closed \(IW_e(A, EID, E)\) term and \(E \subseteq EID\) such that \(E(s) \subseteq E\), then \(s, E\) is a basic \(LP\) term

2) no other closed \(LP\) terms are basic \(LP\) terms

**Theorem 2.3.12 (Elimination)** For every closed \(LP\) term \(s\) there exists a basic \(LP\) term \(t\) such that \(IWE_e(A, EID, E) \vdash s = t\).
Proof This theorem is proven with induction on the structure of a closed LP term. First consider the case \( s \equiv \langle s', E' \rangle \) (\( s' \) a closed \( IW_e(A, EID, E) \) term and \( E' \subseteq EID \)). If \( E(s') \subseteq E' \) then \( s \) is a basic LP term. If \( E(s') \not\subseteq E' \) then we have \( \langle s', E' \rangle = \langle s', E' \cup E(s') \rangle \) which is a basic LP term. Next, consider the case \( s \equiv s_1 \oplus s_2 \) (\( s_1, s_2 \) closed LP terms) for \( \oplus \in \{ +, \cdot, \oplus, \ominus \} \). Then we have by induction that \( s_1, s_2 \) are basic LP terms, i.e. \( s_1 \equiv \langle t_1, E_1 \rangle \) and \( s_2 \equiv \langle t_2, E_2 \rangle \) for some \( t_1, t_2 \) closed \( IW_e(A, EID, E) \) terms and \( E_1, E_2 \subseteq EID \). Then we can apply the second axiom to obtain \( s = \langle t_1 \oplus t_2, E_1 \cup E_2 \rangle \) which is a basic LP term. Finally, consider the case \( s \equiv \{ s \} \|_{iw} s_2 \). Again by induction we have that \( s_1 \) and \( s_2 \) are basic LP terms. Therefore, we have that \( s_1 \equiv \langle t_1, E_1 \rangle \) with \( E(t_1) \subseteq E_1 \) and that \( s_2 \equiv \langle t_2, E_2 \rangle \) with \( E(t_2) \subseteq E_2 \). Then we can apply the third axiom and obtain \( s \equiv \langle t_1 \|_{iw} t_2, E_1 \cup E_2 \rangle \) which is a basic LP term.

**Theorem 2.3.13 (Completeness)** The process algebra \( IW_e(A, EID, E) \) is a complete axiomatization of entity bisimulation on closed LP terms.

Proof By the elimination theorem we only have to prove this theorem for basic LP terms. Let \( \langle s, E_1 \rangle \) and \( \langle t, E_2 \rangle \) be basic LP terms such that \( \langle s, E_1 \rangle \iff \langle t, E_2 \rangle \). By the definition of entity bisimulation we have \( s \iff t \) and \( E_1 \cup E(s) = E_1 = E_2 = E_2 \cup E(t) \). In Theorem 2.3.5 we proved that \( IW_e(A, EID, E) \) is a complete axiomatization of bisimulation equivalence on closed \( IW_e(A, EID, E) \) terms. So we have \( \langle s, E_1 \rangle = \langle t, E_2 \rangle \).

### 3 Properties

In this section we will show that the interworking sequencing is commutative under the assumption that the active entities of the operands are disjoint and that the interworking sequencing is associative. We will also prove that the interworking merge is both commutative and associative. The interworking merge as defined by [MvWW93] did not have the associativity property. This is a direct consequence of our decision to maintain the entities of an Interworking statically. We can illustrate this with the following example.

**Example 3.1** In the semantics of [MvWW93] the Interworkings as shown in Figure 3 are given by \( A = c(p, q, m) \) and \( B = c(p, q, n) \), whereas in this paper they are represented by \( A = \langle c(p, q, m), \{ p, q \} \rangle \) and \( B = \langle c(p, q, n), \{ p, q \} \rangle \).

![Figure 3: Example Interworkings](image)

We compute the expressions \( A \|_{iw} A \|_{iw} B \) and \( A \|_{iw} (A \|_{iw} B) \) in both settings. In the semantics of [MvWW93] we have the following computations.

\[
(A \|_{iw} A) \|_{iw} B = (c(p, q, m) \|_{iw} c(p, q, m)) \|_{iw} c(p, q, n) \\
= c(p, q, m) \|_{iw} c(p, q, n) \\
= \delta
\]
\[ A \, ||_{iw} \, (A \, ||_{iw} \, B) = c(p, q, m) \, ||_{iw} \, (c(p, q, m) \, ||_{iw} \, c(p, q, n)) \]
\[ = c(p, q, m) \, ||_{iw} \, \delta \]
\[ = c(p, q, m) \cdot \delta. \]

In the semantics of Interworkings as presented in this paper we have the following computations. Denote the set \( \{p, q\} \) by \( K \).

\[(A \, ||_{iw} \, A) \, ||_{iw} \, B = (c(p, q, m), K) \, ||_{iw} \, (c(p, q, m), K) \, ||_{iw} \, (c(p, q, n), K) \]
\[ = (c(p, q, m), K) \, ||_{iw} \, (c(p, q, n), K) \]
\[ = (c(p, q, m), K) \, ||_{iw} \, (c(p, q, n), K) \]
\[ = (\delta, K) \]

\[ A \, ||_{iw} \, (A \, ||_{iw} \, B) = \]
\[ = (c(p, q, m), K) \, ||_{iw} \, (c(p, q, m), K) \, ||_{iw} \, (c(p, q, n), K) \]
\[ = (c(p, q, m), K) \, ||_{iw} \, (c(p, q, n), K) \]
\[ = (c(p, q, m), K) \, ||_{iw} \, (\delta, K) \]
\[ = (c(p, q, m), K) \, ||_{iw} \, (c(p, q, n), K) \]
\[ = (\delta, K) \]

These two computations illustrate the difference fairly well. In the computation of \( A \, ||_{iw} \, B \) in the semantics of \([\text{MvWW93}]\) we lost information on the entities which are present, whereas in the second computation we did not. Observe that in the definition of interworking merge as presented in this paper \( (A \, ||_{iw} \, A) \, ||_{iw} \, B \) and \( A \, ||_{iw} \, (A \, ||_{iw} \, B) \) are entity bisimilar.

**Proposition 3.2 (Commutativity of \( q_{iw} \) and \( ||_{iw} \))** For closed \( IW_z(A, E(D), E) \) terms \( x \) and \( y \) and sets of entities \( E, E_1 \) and \( E_2 \) we have

\[ E(x) \cap E(y) = \emptyset \Rightarrow x \, q_{iw} \, y = y \, q_{iw} \, x \]  \hspace{1cm} (1)
\[ x \, ||_{iw} \, E_1 = y \, ||_{iw} \, E_2 \]  \hspace{1cm} (2)
\[ \langle x, E_1 \rangle \, ||_{iw} \, (y, E_2) = \langle y, E_2 \rangle \, ||_{iw} \, (x, E_1) \]  \hspace{1cm} (3)
\[ x \, ||_{iw} \, E_1 = y \, ||_{iw} \, E_2 \]  \hspace{1cm} (4)
\[ \langle x, E_1 \rangle \, ||_{iw} \, (y, E_2) = \langle y, E_2 \rangle \, ||_{iw} \, (x, E_1) \]  \hspace{1cm} (5)

**Proof** For proofs of the propositions (1) and (3) we refer to \([\text{Ren93}]\). Proposition (2) follows immediately from proposition (1). Proposition (4) follows immediately from proposition (3). Proposition (5) can be proven as follows. Denote \( E_1 \cup E(x) \) and \( E_2 \cup E(y) \) by \( E'_1 \) and \( E'_2 \) respectively.

\[ \langle x, E_1 \rangle \, ||_{iw} \, (y, E_2) = \langle x, E'_1 \rangle \, ||_{iw} \, (y, E'_2) \]
\[ = (x) \, ||_{iw} \, (E'_1 \cap E'_2), y, E'_1 \cup E'_2) \]
\[ = (x) \, ||_{iw} \, (E'_1 \cap E'_2), (y, E'_2) \]
\[ = (y) \, ||_{iw} \, (E'_1 \cap E'_2), (x, E'_1) \]
\[ = (y) \, ||_{iw} \, (E'_1 \cap E'_2), x, E'_1 \cup E'_2) \]
\[ = (y) \, ||_{iw} \, (x, E'_1) \]
\[ = (y, E'_2) \, ||_{iw} \, (x, E'_1) \]

379
Proposition 3.3 (Associativity of $\text{l}w$ and $\llbracket \text{l}w \rrbracket$) For closed $\text{IW}_r(A, EID, E)$ terms $x$, $y$, and $z$ and $E_1, E_2, E_3 \subseteq EID$ we have

\begin{align}
(x \text{l}w y) \text{l}w z &= x \text{l}w (y \text{l}w z) \\
((x, E_1) \text{l}w (y, E_2)) \text{l}w (z, E_3) &= (x, E_1) \text{l}w ((y, E_2) \text{l}w (z, E_3)) \\
(x \llbracket E_1 \cap E_2 \rrbracket \text{l}w (y \llbracket E_1 \cup E_2 \rrbracket)) \llbracket E_3 \rrbracket &= x \llbracket (E_1 \cap E_2) \cup E_3 \rrbracket \llbracket E_3 \rrbracket \\
((x, E_1) \llbracket y \llbracket (y, E_2) \llbracket z, E_3) &= (x, E_1) \llbracket (y, E_2) \llbracket (z, E_3)
\end{align}

Proof For a proof of proposition (1) we refer to [Ren93]. Proposition (2) follows easily from proposition (1):

\begin{align}
(x, E_1) \text{l}w (y, E_2)) \text{l}w (z, E_3) &= (x, E_1) \text{l}w (y, E_2) \text{l}w (z, E_3) \\
&= (x, E_1) \text{l}w (y, E_2) \text{l}w (z, E_3)
\end{align}

Proposition (3) is proven with simultaneous induction on the total number of symbols in $x$, $y$ and $z$ of the following propositions. We have used the following abbreviations: $S = E_1 \cap E_2 \cap E_3$, $A = (E_1 \cap E_2) - S$, $B = (E_2 \cap E_3) - S$, and $C = (E_1 \cap E_3) - S$.

\begin{align}
(x \llbracket A \rrbracket \text{l}w y) \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket &= x \llbracket A \rrbracket \text{l}w \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket \\
(x \llbracket A \rrbracket \text{l}w y) \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket &= x \llbracket A \rrbracket \text{l}w \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket \\
(x \llbracket A \rrbracket \text{l}w y) \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket &= x \llbracket A \rrbracket \text{l}w \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket \\
(x \llbracket A \rrbracket \text{l}w y) \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket &= x \llbracket A \rrbracket \text{l}w \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket \\
\sqrt{(x \llbracket A \rrbracket \text{l}w y) \cdot \sqrt{(z)} &= \sqrt{(x) \cdot \sqrt{(y \llbracket B \rrbracket \text{l}w z)}} \\
(x \llbracket A \rrbracket \text{l}w y) \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket &= x \llbracket A \rrbracket \text{l}w \llbracket B \rrbracket \text{l}w \llbracket C \rrbracket
\end{align}

We will not prove these propositions. Proposition (4) follows from proposition (3) as follows. We use the following abbreviations: $F_1 = E_1 \cup E(x)$, $F_2 = E_2 \cup E(y)$, and $F_3 = E_3 \cup E(z)$.

\begin{align}
((x, E_1) \llbracket y \llbracket (y, E_2) \llbracket z, E_3) &= ((x, F_1) \llbracket y \llbracket (y, F_2) \llbracket (z, F_3) \\
&= (x, F_1) \llbracket y \llbracket (y, F_2) \llbracket (z, F_3) \\
&= (x, F_1) \llbracket y \llbracket (y, F_2) \llbracket (z, F_3) \\
&= (x, F_1) \llbracket y \llbracket (y, F_2) \llbracket (z, F_3) \\
&= (x, F_1) \llbracket y \llbracket (y, F_2) \llbracket (z, F_3)
\end{align}

4 Algebraic Definition of Interworking Refinement

Interworking refinement is the replacement of one entity by a number of entities such that the behaviour of the refining Interworking is identical, in a sense to be made precise shortly, to the original Interworking.

Let $f : EID \rightarrow EID$ be a partial mapping from entities to entities. An Interworking $x$ can be an $f$-refinement of another Interworking $y$, denoted by $x \subseteq f y$. This is the case if, after renaming of a set of entities of $x$ into one entity of $y$ (according to $f$) and after removal of all internal actions on the refined entity within the refined interworking and the removal of all internal actions on the
refining entities within the refining interworking, the behaviour of both interworkings is equal. The mapping \( f \) is partial in order to distinguish between an entity \( p \) which is not refined at all \( (p \notin \text{dom}(f)) \) and an entity \( p \) which is refined by (amongst others) an entity \( q \) \((f(p) = q)\). For an example of interworking refinement see Figure 4. The entities \( q_1 \) and \( q_2 \) refine the entity \( q \).

The intuition is that the external behaviour of a single entity within the Interworking \( y \) can be refined into, or implemented by, a collective behaviour of a number of entities within the Interworking \( x \). So, the emphasis is on inter-entity communication and not so much on intra-entity behaviour. Besides the singular \( f \)-entity refinement discussed above, it is also allowed to consider a number of refinements at the same time: multiple refinement. An example of multiple refinement is given in Figure 5. The entity \( p \) is refined by the entities \( p_1 \) and \( p_2 \), and the entity \( q \) is refined by the entities \( q_1 \) and \( q_2 \).

Before we define this refinement formally we define the renaming function \( \rho_f \). This operator renames all occurrences of \( e \in \text{EID} \) into \( f(e) \). For the axiomatization of this operator it is easier to have a total function instead of a partial one. Thereto, we extend the partial function \( f \) to the total function \( f^* \) by asserting that \( f^*(x) = f(x) \) for all \( x \) for which \( f \) is not defined. Let \( F \) be a set and let \( f : F \to F \) be a partial function. The total function associated with \( f \), notation \( f^* \), is for all \( x \in F \), defined by

\[
f^*(x) = \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \\ x & \text{if } x \notin \text{dom}(f) \end{cases}
\]

Let \( f : \text{EID} \to \text{EID} \) be a partial function, then the renaming operator \( \rho_f \) related to \( f \) is defined by the axioms in Table 11. This renaming operator resembles the renaming operator \( \rho_f \) from [BB88].

---

**Figure 4: Interworking Refinement**

**Figure 5: Multiple Interworking Refinement**
\[
\begin{align*}
\rho_f(\varepsilon) &= \varepsilon \\
\rho_f(\delta) &= \delta \\
\rho_f(c(p, q, m)) &= c(f^*(p), f^*(q), m) \\
\rho_f(a \cdot x) &= \rho_f(a) \cdot \rho_f(x) \\
\rho_f(x + y) &= \rho_f(x) + \rho_f(y)
\end{align*}
\]

Table 11: Entity Renaming function on processes

In Table 12 the entity renaming operator on processes is extended to labelled processes. Note that also the entity component of a labelled process is renamed with respect to the mapping \( f \).

\[
\rho_f((x, E)) = (\rho_f(x), \{f^*(e) | e \in E\})
\]

Table 12: Entity Renaming function on labelled processes

In order to remove all internal actions on entities we are not interested in, we substitute the empty process \( \varepsilon \) for them. The entities for which we remove the internal actions are given by \( \text{rng}(f) \). We define the set of all internal actions on the entities of a set \( E \) as follows:

\[
\text{Int}(E) = \{a \in A | E(a) \subseteq E \land |E(a)| = 1\}
\]

Let \( I \) be a set of atomic actions, in Table 13 we define the operator \( \varepsilon_I \) that renames atomic actions from \( I \) into \( \varepsilon \). This operator is taken from [Vra91].

\[
\begin{align*}
\varepsilon_I(\varepsilon) &= \varepsilon \\
\varepsilon_I(\delta) &= \delta \\
\varepsilon_I(a \cdot x) &= \varepsilon_I(x) & \text{if } a \in I \\
\varepsilon_I(a \cdot x) &= a \cdot \varepsilon_I(x) & \text{if } a \notin I \\
\varepsilon_I(x + y) &= \varepsilon_I(x) + \varepsilon_I(y)
\end{align*}
\]

Table 13: Renaming atomic actions into \( \varepsilon \)

In Table 14 the operator for removing internal communications is extended to labelled processes. By renaming actions into \( \varepsilon \) it can be the case that entity information is removed completely from the process expression. Therefore, we first make sure that all entity information is contained in the entity component of the labelled process.

Let \( f : EID \rightarrow EID \) be a refinement mapping, the \( f \)-refinement relation on labelled processes is then defined by the equation in Table 15.

Next, we extend this notion of refinement with a fixed mapping to a notion of refinement which abstracts from this mapping. This notion of refinement is called entity refinement. Interworking \( x \) is an entity refinement of Interworking \( y \), notation \( x \sqsubseteq_f y \) if and only if there exists a refinement mapping \( f \) such that \( x \subseteq_f y \).

Example 4.1 As an illustration of this algebraic definition of refinement the refinement relation between the Interworkings in Figure 5 is computed. The left-hand side Interworking will be called
Table 15: $\rho$-Refinement

Proposition 4.2 (Reflexivity) For all closed labelled processes $x$ we have $x \sqsubseteq x$.

Proof We have to show that there exists a partial mapping $f : EID \rightarrow EID$ such that $x \sqsubseteq f y$. Take the mapping $f$ with empty domain. Then $x$ is an $f$-refinement of $x$.

Proposition 4.3 (Transitivity) For all closed labelled processes $x$, $y$, and $z$ we have

$x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$.

Table 16: Entity refinement
Proof Let $F$ be some set and let $f : F \to F$ be a partial function. For all $G \subseteq F$ the extension of $f$ with respect to $G$, notation $f^G$, is, for all $x \in F$, defined as follows

$$f^G(x) = \begin{cases} f(x) & \text{if } x \in \text{dom}(f) \\ x & \text{if } x \notin \text{dom}(f) \land x \in G \\ \text{undefined} & \text{if } x \notin \text{dom}(f) \land x \notin G \end{cases}$$

Suppose that there exist $f, g : EID \to EID$ such that $x \sqsubseteq_f y$ and $y \sqsubseteq_g z$. It is our claim that $x \sqsubseteq_{g^*\text{ref}(f)} z$. The proof of this claim is omitted.

We do not have that the relation $\sqsubseteq$ is anti-symmetrical. This is due to the treatment of internal actions. Consider, for example, the Interworkings $x = c(p, p, m)$ and $y = c(p, p, n)$. Then we have $x \sqsubseteq y$ and $y \sqsubseteq x$, but we do not have $x = y$. For Interworkings without internal communications we do have antisymmetry of entity refinement. So for Interworkings without internal communication the entity refinement relation is a partial ordering. For the more general class of Interworkings entity refinement is a pre-order.

5 Conclusions

We have given a semantics for Interworkings in which we solved the problems encountered in a former semantics and which allows a definition of refinement. An empty interworking is simply represented by the empty process. The anomaly with respect to empty entities in the context of an interworking merge has been solved by attributing the process expressions with the set of entities involved. The definition of refinement presented here is a little easier than a former definition. We have derived several properties and obtained sound and complete theories.

It is expected that this new semantics has only limited impact on the already existing tools for Interworkings. Furthermore, due to the formal definition also an implementation of refinement seems plausible.

References


Generating FSMs from Interworkings

L.M.G. Feijs
Philips Research Laboratories Eindhoven, University of Technology Eindhoven.

April 19, 1995

Abstract

A technique for the automated synthesis of FSMs (finite state machines) from sets of interworkings (synchronous sequence charts) is described. This is useful for obtaining feedback from a set of scenarios during a system's definition phase or test phase. It is sound in the sense that the generated FSM only exhibits traces that correspond to one of the interworkings from the given set. It preserves deadlock freedom in the sense that no behaviours are lost. The concrete syntax of SDL is used to represent the resulting FSMs.

1 Introduction and motivation

When analysing requirements for a system, viewed as one or more communicating processes, it often occurs that a set of interworkings (synchronous sequence charts, see [1, 2]) is devised. The system itself, however is often designed as one or more communicating FSMs (finite state machines), for example described in SDL [3, 4]. The automatic generation of a FSM from a set of interworkings can be used for obtaining early feedback from initial and possibly incomplete scenarios during a system's definition phase. In particular if the resulting FSM is presented in an attractive graphical syntax (e.g. by means of SDL tools), it reveals the shortcomings of a given set of scenarios. In the same way it can also be used to get feedback on a set of test scenarios during a system's test phase. On the long term it could even lead to better support or partial automation of the generation of the product's FSMs. It could be equally useful for generating test programs.

The work described in this report starts from the observation that both interworkings and FSMs can be viewed as algebraic terms. Interworkings are interpreted as terms of ACP (algebra of communicating processes), following the approach of Mauw, Van Wijk and Winter [2]. Sets of interworkings are interpreted in a setting based on bisimulation semantics. FSMs are coded in ACP too. In particular, a machine with a finite set of states $S$ can be viewed as a process, say $p$, described by equations. For each state $s \in S$ there is one equation of the form

$$p_s \overset{\text{def}}{=} \sum_{s' \in S} t_{s,s'} \cdot p_{s'}$$

to describe the transitions $t_{s,s'}$ from state $s$ to state $s'$ (this is similar to the well-known UNITY format, see [5]). The fact that both the input (interworkings) and the output (FSMs) are viewed as algebraic terms provides the approach with a formal basis and facilitates the use of term rewriting as a transformation technique. The essential steps of the synthesis are performed by means of rewrite rules, such as the distributive rules, directed as contractions.
A sequence of transformation steps is described to transform a set of interworkings into a FSM description for one selected process. First an ACP [6, 7] term is derived by taking the sum of the terms of the individual interworkings (so in this term the + operator occurs). The subsequent transformation steps guarantee a result which is sound in a bisimulation setting (and which is consistent with Mauw’s delayed choice semantics [8] for a set of interworkings). To this sum term various rewrite rules are applied in order to arrive at a set of equations which can be interpreted as describing transitions. The main transformers are the distributive rules, directed as contractions:

\[
\begin{align*}
\frac{z \cdot x + z \cdot y}{\Rightarrow} & \quad z \cdot (x + y) \quad \text{TR} \\
\frac{x \cdot z + y \cdot z}{\Rightarrow} & \quad (x + y) \cdot z \quad \text{A4}
\end{align*}
\]

By factoring out common subparts from distinct traces (using the rule \(x + x \rightarrow x\) from process algebra next to TR and A4, next to the usual rules for +, like commutativity) the number of states of the resulting state machine can be reduced. The rule TR does not belong to ACP, but here it is needed to perform a transformation, the properties of which (soundness, preservation of deadlock freedom and completeness) will be analyzed using ACP.

We give an example to make it plausible at this point already that factoring out common subparts of process terms reduces the number of states when using a state machine interpretation of processes. Let \(\tau(m)\) be the action of receiving message \(m\), then the example process given below, having four summands (traces), can be rewritten as follows:

\[
\begin{align*}
\tau(a) \cdot \tau(c) \cdot \tau(d) + \tau(a) \cdot \tau(c) \cdot \tau(e) + \tau(b) \cdot \tau(c) \cdot \tau(d) + \tau(b) \cdot \tau(c) \cdot \tau(e)
\end{align*}
\]

rewrite

\[
\begin{align*}
(r(a) + r(b)) \cdot \tau(c) \cdot (r(d) + r(e))
\end{align*}
\]

In our state machine interpretation of processes this rewriting corresponds to the transformation shown in Figure 1.

![Figure 1: Reducing the number of states of a process.](image)

Note the effect of rule TR, which postpones choices: whereas the left hand side process in Figure 1 can make the wrong choice upon reception of \(a\) (e.g. taking the first branch, leading to deadlock later when an \(e\) comes instead of \(d\)), the right hand side process
chooses exactly at the appropriate point. Another way of viewing this is given in [8] where a special summation operator called 'delayed choice' (⊕) is used to combine a collection of traces into a single term.

We have built an experimental compiler using Prolog during a feasibility study in which we constructed one particular realization of the transformation of Section 3. It is described in [10], which also shows the approach in action on a very small example (2 interworkings) and on a slightly larger 'case study' (12 interworkings).

Now we survey the paper. Section 2 provides the necessary definitions concerning the elementary communication actions, and traces thereof. The body of the work consists of a description of the step-wise transformation of a set of interworkings to a FSM, which consists of five transformation steps. Section 3 gives for each step its domain and its range, together with the transformation itself. Section 4 is devoted to an analysis of the properties of the FSM thus obtained. Section 5 presents some conclusions and options for further research, also based on considerations gathered during the experiments reported on in [10]. The appendix contains details of the rewriting and implementation notes of the experimental compiler.

2 Assumptions and notations


**Definition 2.1** Let \( EID \) and \( MID \) be finite sets of entity identifiers and message identifiers respectively. Typical elements of \( EID \) are denoted by \( p, p_1, p_2, \ldots \) or \( q, q_1, q_2, \ldots \) and typical elements of \( MID \) are denoted by \( m, m_1, m_2, \ldots \). A pair \((p, q)\) with \( p, q \in EID \) is called a port.

We will consider a specific set of actions \( A \), containing send actions, receive actions and communication actions via ports.

**Definition 2.2** The action set \( A \) is defined by:

\[
A = \{ s_{(p,q)}(m), r_{(p,q)}(m), c_{(p,q)}(m) \mid p, q \in EID, m \in MID \}
\]

We excluded ports \( (p, p) \) (but there is no real problem to add them). If we extend this set of atomic actions with constants such as \( \epsilon \) (the empty process), \( \delta \) (deadlock), or combinations, we get \( A_\epsilon, A_\delta, A_{\delta, \epsilon} \) etc. Atomic actions will be denoted by \( a, b, \ldots \).

We write \( s_{p,q}(m) \) for \( s_{(p,q)}(m) \) and similarly \( r_{p,q}(m) \) and \( c_{p,q}(m) \) for \( r_{(p,q)}(m) \) and \( c_{(p,q)}(m) \), respectively.

**Definition 2.3** The communication function \( \gamma \) is defined in the obvious way:

\[
\begin{align*}
\gamma(s_{p,q}(m), r_{p,q}(m)) &= \\
\gamma(r_{p,q}(m), s_{p,q}(m)) &= c_{p,q}(m) \\
\gamma(a, b) &= \delta \quad \text{for all other cases.}
\end{align*}
\]

Later this \( \gamma \) will be extended to the communication merge \( \mid \) by the rules of ACP (the algebra of communicating processes).
Processes will be denoted by $x, y, \ldots$. Processes are constructed from atomic actions and the operators $+$ for \textit{alternative composition} and $\cdot$ for \textit{sequential composition}. The process $x + y$ executes either $x$ or $y$, but not both (the choice between $x$ and $y$ can depend on communications offered by other processes, but is nondeterministic otherwise). The process $x \cdot y$ first executes $x$, and upon completion of $x$ starts $y$. Because we are explicitly studying transformations of (the terms representing) processes we shall take a syntactic point of view and speak about \textit{terms}. Unless stated otherwise we consider terms from the set defined next:

\textbf{Definition 2.4} Let $\text{Term}(A_{\delta, e})$ denote the set of terms built over $A_{\delta, e}$ using $+$ and $\cdot$.

We shall study the transformation of terms in $\text{Term}(A_{\delta, e})$ which are the analogues of interworkings, sets thereof, and fragments of FSM descriptions. This is the main subject of Section 3. In the analysis of these transformations (formulating properties of synthesized FSMs in Section 4), we shall employ $\|$, $\|$, and $|$ as well.

\textbf{Definition 2.5} A \textit{communication trace} is a finite sequence of communication actions, that is $c_{p_1,q_1}(m_1) \cdots c_{p_N,q_N}(m_N)$ for $N \in \mathbb{N}$. The empty communication trace ($N = 0$) is denoted by $\lambda$. $CT$ denotes the set of communication traces.

We can also represent communication traces as products of communication actions, that is as $c_{p_1,q_1}(m_1) \cdots c_{p_N,q_N}(m_N) \cdot \epsilon$ for $N \in \mathbb{N}$ (of course the $\epsilon$ is redundant but it allows us to write down the definitions of some of our operators in such a way that they can be made operational directly at the syntactic level of terms). For $N = 0$ there is only one communication trace, which is $\epsilon$. Sometimes we shall convert implicitly from the sequence representation to product representation or conversely; in this sense $CT \subset \text{Term}(A_{\epsilon})$.

Later we shall obtain the elements of $CT$ by translating interworkings to process algebra terms.

\textbf{Definition 2.6} If $\vec{i} \in CT^*$ is a finite sequence of communication traces, such that $\vec{i} = (i_0, i_1, \ldots, i_{n-1})$, then $\Sigma \vec{i}$ is the term $i_0 + i_1 + \ldots + i_{n-1}$. If $n = 0$ we find that $\vec{i}$ is empty and we define $\Sigma \vec{i} = \delta$.

This $\Sigma$ is a useful operator that brings a collection of interworkings $\vec{i}$ completely into the realm of process algebra by turning the collection into a single term. Of course it is appropriate to view $\Sigma \vec{i}$ as a multiset and indeed we shall use commutativity rules. But we treat it as a sequence here in order to describe the (syntactic) transformation process as precisely as possible.

\textbf{Definition 2.7} For $p \in EID$, define $\text{project}_p : \text{Term}(A_{\delta, e}) \rightarrow \text{Term}(A_{\delta, e})$ by:

\[
\begin{align*}
\text{project}_p(\epsilon) & = \epsilon \\
\text{project}_p(c_{q_1,q_2}(m) \cdot x) & = \text{project}_p(x) & \text{if } p \neq q_1 \text{ and } p \neq q_2 \\
\text{project}_p(c_{p,q}(m) \cdot x) & = s_{p,q}(m) \cdot \text{project}_p(x) \\
\text{project}_p(c_{q,p}(m) \cdot x) & = r_{q,p}(m) \cdot \text{project}_p(x) \\
\text{project}_p(\delta) & = \delta \\
\text{project}_p(x + y) & = \text{project}_p(x) + \text{project}_p(y)
\end{align*}
\]
The operator $\text{project}_p$ is our main transformation to reconstruct an entity (process) identified by $p$ from a communication trace. And this transformation is recursively applied to all subterms of a sum. So for given $p$ and $i$ we can make $\text{project}_p(\Sigma i)$ which is a term containing information about the behaviour of $p$.

Example 2.8 Consider the following two interworkings shown in the textual syntax of [2]:

\begin{verbatim}
INTERWORKING an_example
  PROCESSES x,p,q ENDPROCESSES
  p SENDS m1 TO x
  q SENDS m TO x
  x SENDS a TO p
ENDINTERWORKING

INTERWORKING another_example
  PROCESSES x,p,q ENDPROCESSES
  p SENDS m2 TO x
  q SENDS m TO x
  x SENDS a TO p
ENDINTERWORKING
\end{verbatim}

or, which is the same, Figure 2.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Interworking diagrams.}
\end{figure}

From these we get $i = \langle (c_{p,x}(m_1) \cdot c_{q,x}(m) \cdot c_{x,p}(a) \cdot \epsilon \cdot c_{p,x}(m_2) \cdot c_{q,x}(m) \cdot c_{x,p}(a) \cdot \epsilon \rangle$. Therefore $\text{project}_x(\Sigma i) = r_{p,x}(m_1) \cdot r_{q,x}(m) \cdot s_{x,p}(a) \cdot \epsilon + r_{p,x}(m_2) \cdot r_{q,x}(m) \cdot s_{x,p}(a) \cdot \epsilon$.

In [2] the semantics of an interworking is defined using a special operator $\circ_{iw}$ for combining the communication actions into a process term. For the present purpose we simply use $\cdot$, which is justified because the first transformation step will be to project the semantics of each interworking onto one particular process and this projection (project$_p$) makes the distinction between $\circ_{iw}$ and $\cdot$ vanish. Let us make this justification more precise by means of the following example:

\begin{verbatim}
INTERWORKING pqrst
  PROCESSES p,q,r,s,t ENDPROCESSES
  p SENDS m TO q
  p SENDS n TO r
  s SENDS l TO t
ENDINTERWORKING
\end{verbatim}

In [2] this is interpreted as $c_{p,q}(m) \circ_{iw} (c_{p,r}(n) \circ_{iw} c_{s,t}(l))$ which by the rules for eliminating the auxiliary operators $L_{iw}$ and $R_{iw}$ given in [2] equals $c_{p,q}(m) \cdot (c_{p,r}(n) \cdot c_{s,t}(l) + c_{s,t}(l) \cdot c_{p,r}(n)) + c_{s,t}(l) \cdot c_{p,q}(m) \cdot c_{p,r}(n)$.\footnote{The intuition in [2] is that since $c_{s,t}(l)$ has no entities in common with the other two transmissions, pqrst only requires that $c_{p,q}(m)$ precedes $c_{p,r}(n)$ and should not require the total ordering of all three transmissions, as possibly suggested by the diagram.} so that application of project$_p$ yields $s_{p,q}(m) \cdot (s_{p,r}(n) + \ldots)$.
$s_{p,r}(n) + s_{p,q}(m) - s_{p,r}(n)$ which in ACP (use A3, i.e. $x + x = x$, twice) equals $s_{p,q}(m) - s_{p,r}(n)$. This is the same as project$_p(c_{p,q}(m) - c_{p,r}(n))$, which is used in our $\cdot$-based approach.

We proceed with some further preparations, formalizing the domain and the range of each step. The reader might find it helpful to have a quick look at the first definition of Section 3 where these preparations are used to define the signature of the transformation steps.

**Definition 2.9 (lifting).** We consider lifted terms, that is terms like those in $\text{Term}(A_{\delta,\epsilon})$ except for the fact that some subterms are marked by indices in $\mathbb{N}$. We write $\text{Term}_{\mathbb{N}}(A_{\delta,\epsilon})$ to denote the set of such terms. Marked subterms may contain marked subterms themselves. For $x \in \text{Term}_{\mathbb{N}}(A_{\delta,\epsilon})$ we write $|x|$ for $x$ with all indices removed.

The indices that mark subterms will be used for finding and identifying subterms corresponding to 'states'. Once the states are identified, we shall introduce variables $P_n$ for folding the subterms $(x)_n$ (see Example 2.12 below). The states and the transitions of a FSM can be defined by a finite family of equations

$$E = \{ p_i \overset{\text{def}}{=} x_i \mid 0 \leq i \leq N - 1 \}$$

where the $p_i$ are distinct variables and the $x_i$ are ACP expressions over $A_{\delta,\epsilon}$ at most containing the variables $\{ p_0, \ldots, p_{N-1} \}$. We further assume that every variable occurrence in the $x_i$ is guarded, that is, appears within the scope of an action prefix. The family $E$ determines a set of states $\{ i \mid 0 \leq i \leq N - 1 \}$ and furthermore $E$ determines the transitions from state $i$ which are described by the expression $x_i$ as given by the equation $p_i \overset{\text{def}}{=} x_i$. In this paper we shall restrict ourselves to non-recursive $E$. In order to represent a FSM in this way we must add an initial state, which is given by one more equation $e$ of the form

$$p \overset{\text{def}}{=} x$$

where $p$, which must be distinct from the $p_i$, is called the 'leading variable' and where $x$ is another ACP expression at most containing the variables $\{ p_0, \ldots, p_{N-1} \}$. Thus an FSM is given by a pair $(e, E)$ where $e$ is the defining equation for the leading variable and $E$ is a finite non-recursive set of guarded equations.

**Definition 2.11** We write FSM for the set of pairs $(e, E)$ where $e$ is the defining equation for the leading variable and $E$ is a finite non-recursive set of guarded equations in $A_{\delta,\epsilon}$.

In other words, now we are able to organize algebraic equations in such a way that they can be viewed as a description of a finite state machine: $p_0$ describes the events which happen when the machine is in state 0 (typically some actions ending with some other $p_n$); $p_1$ describes state 1, and so on. Although in general it is no problem to allow for recursion, this will not be needed for the translation of interworkings where we accept a finite number of finite input traces which are combined into a terminating process.

**Example 2.12** Let the equation $e$ be given as $p \overset{\text{def}}{=} p_0$ and let the set of equations $E$ be $\{ p_0 \overset{\text{def}}{=} r_{p,x}(m_2) \cdot p_1 + r_{p,x}(m_1) \cdot p_1, \ p_1 \overset{\text{def}}{=} r_{q,x}(m) \cdot s_{x,p}(a) \}$. Then $(e, E)$ represents the state-machine named $x$ shown in Figure 3 (as a matter of fact these equations describe an FSM which can behave according to the interworkings of Example 2.8).
In above example there is one begin-point: the oval whose outgoing edges are derived from the equation \( e \) defining \( p \). In this example there is one such edge. It leads to state 0: the box with the rounded sides which has two outgoing edges, being derived from the two summands in the equation for \( p_0 \). There is one termination point: the ‘cross’ for successful termination, which denotes the end of the transition for \( p_1 \). The symbols for input (receive) actions and output (send) actions speak for themselves. As in the example, we shall always take care that each input action is preceded by a state (the state where the process waits for input). Output actions can immediately follow receive actions or other send actions, however.

3 The transformation steps

We shall describe a five-step transformation process \( S \) from interworkings to FSMs: The steps are surveyed first:

1. projection onto selected process (purpose: prepare for algebra),
2. contraction (purpose: postpone choices and reduce state space),
3. labeling subterms (purpose: identify states),
4. unwinding terms by right expansion (purpose: avoid ‘joins’, cf. Figure 5),
5. find transition set by folding (purpose: generate FSM).

We adopt a number of additional constraints on the output format which are motivated by our goal of representing the output in an SDL-like syntax. For example, each input action will be preceded by a state, where the process waits for input. These constraints will be explained below when presenting the steps.

For a given entity identifier \( p \) we consider the transformation \( S_p \) consisting of five steps to take a sequence of interworkings \( \tilde{i} \) and turn it into a pair \((e, E)\) containing equations as in the last example of Section 2. In other words, we consider \( \tilde{i} \) as the specification of one ‘target’ process and assume that the other processes are its environment. For example \( p \) could refer to a system to be designed as an FSM whereas its environment will somehow be...
determined externally. Although one could envisage certain generalisations (synthesizing two or more processes in one transformation) this is not done here. Of course one can apply $S$ to $i$ more than once, each time with a different entity identifier.

**Definition 3.1** The domain and the range of the steps of the transformation are defined as follows:

- $\text{project} \circ \Sigma : EID \times CT^* \rightarrow \text{Term}(A_{\delta,c})$
- $\text{rc} \circ \text{lcf} : \text{Term}(A_{\delta,c}) \rightarrow \text{Term}(A_{\delta,c})$ (right and left contraction)
- $\text{lift} : \text{Term}(A_{\delta,c}) \rightarrow \text{Term}_N(A_{\delta,c})$
- $\text{re} : \text{Term}_N(A_{\delta,c}) \rightarrow \text{Term}_N(A_{\delta,c})$ (right expansion)
- $\text{fold} : \text{Term}_N(A_{\delta,c}) \rightarrow \text{FSM}$

Then these steps can be composed as follows:

$$S_p = \text{fold} \circ \text{re} \circ \text{lcf} \circ \text{rc} \circ \text{project}_p \circ \Sigma$$

**Restriction 3.2** We restrict application of $S_p$ exclusively to those $i$ that satisfy the uniqueness condition $U$ defined by:

$$U(i) : \forall p_1, p_2, p \in EID \land \forall m \in MID \cdot \forall i \cdot c_{p_1,p}(m) \in i \land c_{p_2,p}(m) \in i \Rightarrow p_1 = p_2$$

Here we use ‘$i$’ also to denote: ‘occurs in some $i$ occurring in $i$’. The restriction $U$ is useful when assuming that the FSMs cannot receive selectively depending on the sender’s identity (as is the case in SDL).

In other words: we are not interested in the name of the sender of a message. In practice we can always choose distinct names to identify the communication actions of distinct senders.

The first step is to project all interworkings onto one selected process (e.g. the first mentioned) in order to obtain a set of traces. If $p$ is the selected process, each occurrence of $p \text{ SENDS } m \text{ TO } q$ yields an action $s_{p,q}(m)$ and each occurrence of $q \text{ SENDS } m \text{ TO } p$ yields an action $r_{q,p}(m)$. Each trace is viewed as the product of its actions and the traces are added in order to obtain a process algebra term. The name of the process, say $p$, may be retained for later usage; it is a convenient choice for the leading variable in a system of equations and it can re-appear as the name of the FSM in SDL notation: \text{PROCESS} $p; \ldots$ \text{ENDPROCESS} $p;$. The transformation step is project $\circ \Sigma$, that means, for given $p$ and set of interworkings $\bar{i}$, it results in project$_p(\Sigma \bar{i})$.

The second step is to apply contractions to the process algebra term obtained. Because of associativity:

$$\begin{align*}
(x + y) + z &= x + (y + z) & \text{A2} \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z) & \text{A5}
\end{align*}$$

two special list-representations are used:

$$\begin{align*}
\Sigma [x_1, x_2, \ldots, x_n] & \text{ is } x_1 + (x_2 + \ldots + (x_n + \delta) \ldots) \\
\Pi [x_1, x_2, \ldots, x_n] & \text{ is } x_1 \cdot (x_2 \cdot \ldots \cdot (x_n \cdot \epsilon) \ldots)
\end{align*}$$

Footnote: The symbol $\circ$ denotes function composition: $(f \circ g)(x) = f(g(x))$. For convenience we write project $\circ \Sigma$ as a shorthand for $\lambda(p, \bar{i}) \cdot \text{project}_p(\Sigma \bar{i})$, that is, the function which maps $(p, \bar{i})$ to project$_p(\Sigma \bar{i})$. 

394
together with *flattening* rules which bring all terms into a right-associative form (for more details we refer to the appendix):

\[
\begin{array}{ccc}
\Sigma[x_1] & \rightarrow & x_1 \\
\Pi[x_1] & \rightarrow & x_1 \\
\Sigma[x_1, \ldots, x_n] & \rightarrow & \Sigma[x_1, \ldots, x_m] \\
\Pi[\Pi[x_1, \ldots, x_n]] & \rightarrow & \Pi[\Pi[x_1, \ldots, x_m]]
\end{array}
\]

Semantically the \(\Sigma\)-lists are multisets and therefore it is understood that the option of *permutation* of \(\Sigma\)-lists at the left-hand side of a rewrite is built-in to each rule (cf. axiom A1 of ACP: \(x + y = y + x\)).

If we put \(\epsilon = \Pi[]\) and \(\delta = \Sigma[]\), we have for free:

\[
\begin{align*}
x + \delta &= x & \text{(A6)} \\
\delta + x &= x \\
x \cdot \epsilon &= x & \text{(A8)} \\
\epsilon \cdot x &= x & \text{(A9)}
\end{align*}
\]

The contractions are carried out in two passes. First all left-contractions are done (TR + A3 + A7)^3. Denote the reflexive and transitive closure of these rewrites by \(\rightarrow^*\) (we show the rules in \(+\) and \(\cdot\) notation although in practice they work on \(\Sigma\) and \(\Pi\)-lists).

\[
\begin{align*}
z \cdot x + z \cdot y & \rightarrow z \cdot (x + y) & \text{TR} \\
x + x & \rightarrow x & \text{(A3)} \\
\delta \cdot x & \rightarrow \delta & \text{(A7)}
\end{align*}
\]

As it turns out, TR and A8 have a critical-pair conflict (this is Knuth-Bendix terminology, see e.g. [11]), for consider the term \(z \cdot \epsilon + z \cdot y\) which rewrites to \(z \cdot (\epsilon + y)\) by TR, but which also rewrites to \(z + z \cdot y\) by A8. Therefore we added a completion rule \(z + z \cdot y \rightarrow z \cdot (\epsilon + y)\) which we call TR'. For details of the rewriting we refer to the appendix.

Secondly all right-contractions are done (A4 + A3 + A7)^4. Denote the reflexive and transitive closure of these rewrites by \(\leftarrow^*\).

\[
\begin{align*}
x \cdot z + y \cdot z & \rightarrow (x + y) \cdot z & \text{(A4)} \\
x + x & \rightarrow x & \text{(A3)} \\
\delta \cdot x & \rightarrow \delta & \text{(A7)}
\end{align*}
\]

Combining these, \(\rightarrow^*\) followed by \(\leftarrow^*\) (calculating normal forms twice) yields a 'compact' term.

We ought to explain why the contraction \(\sigma \circ \text{lc}\) cannot be realized in one step. There is a critical-pair conflict between TR and A4 which is illustrated by the example

\[
\begin{align*}
r_{q,p}(m_1) \cdot s_{p,q}(n_1) + r_{q,p}(m_1) \cdot s_{p,q}(n_2) + r_{q,p}(m_2) \cdot s_{p,q}(n_2)
\end{align*}
\]

---

^3 When rewriting terms obtained from \(\Sigma[]\) no A7 contractions will occur but it does no harm to have a rewrite system which applies to general terms.

^4 When rewriting terms obtained from \(\Pi[]\) no A3 and A7 contractions will occur but it does no harm to have a rewrite system which applies to general terms. In general, there exist normal forms with respect to \(\rightarrow^*\) in which an A4 step reveals new A3 redexes, e.g. consider \((x + y) \cdot z + x \cdot z + y \cdot z\).
for which left contraction gives
\[ \text{LC} \equiv r_{q,p}(m_1) \cdot (s_{p,q}(n_1) + s_{p,q}(n_2)) + r_{q,p}(m_2) \cdot s_{p,q}(n_2) \]

whereas right contraction yields
\[ \text{RC} \equiv r_{q,p}(m_1) \cdot s_{p,q}(n_1) + (r_{q,p}(m_1) + r_{q,p}(m_2)) \cdot s_{p,q}(n_2) \]

Once the right-hand-side contraction is done, the left contraction step is blocked, causing possible deadlocks later. For example when the environment offers the behaviour \( \text{ENV} \equiv s_{q,p}(m_1) \cdot r_{p,q}(n_2) \), the combined behaviour \( \partial_H (\text{ENV} \parallel \text{RC}) \) has deadlock because there are two branches starting with \( r_{q,p}(m_1) \) and the first of these turns out the wrong one. This is essentially the same point which led to the concept of 'delayed choice' in [8]. In other words: combining \( \text{\rightarrow} \) and \( \text{\leftarrow} \) into one pass may lead to an undesired FSM where a state has two or more inputs for the same signal (and so the process cannot choose properly). In SDL such states are forbidden.

The third step (lift) is to 'pack' all receive actions into (singleton) \( \Sigma \)-lists and to label all subterms starting with a \( \Sigma \). The packing must be applied to each occurrence of a receive action \( r_{q,p}(m) \), except when it is already a direct member of a \( \Sigma \)-list. This avoids the appearance of undesired FSMS in the fifth step, in the sense that for example an \text{INPUT} is immediately followed by another \text{INPUT} without intermediate state (such FSMS are excluded in SDL), as in Figure 4 (note that in the fifth step one state will be created for each \( \Sigma \)).

\[ \text{wrong} \]

\[ \begin{array}{c}
  0 \\
  \downarrow m_1 \\
  m_2 \\
  \downarrow 1
\end{array} \]

Figure 4: Inputs without intermediate state.

The essential step of the packing is
\[ r_{q,p}(m) \rightarrow \Sigma[r_{q,p}(m)] \]

Formally it is introduced as an operator \( \text{pck} \) defined by:
\[
\begin{align*}
\text{pck}(s_{p,q}(m)) & = s_{p,q}(m) \\
\text{pck}(r_{q,p}(m)) & = \Sigma[r_{q,p}(m)] \\
\text{pck}(\Sigma[x_1, x_2, \ldots, x_n]) & = \Sigma[\text{pck}'(x_1), \text{pck}'(x_2), \ldots, \text{pck}'(x_n)] \\
\text{pck}(\Pi[x_1, x_2, \ldots, x_n]) & = \Pi[\text{pck}(x_1), \text{pck}(x_2), \ldots, \text{pck}(x_n)]
\end{align*}
\]

where the auxiliary operator \( \text{pck}' \) works like \( \text{pck} \) except for the fact that when \( \text{pck}' \) is applied to a product, the first action is not packed recursively:
\[ \text{pck}'(s_{p,q}(m)) = s_{p,q}(m) \]
The packing is followed by a labeling which attaches fresh numbers as labels to each $\Sigma$. We write $\Sigma_n[x \ldots x_k]$ instead of $(\Sigma[x \ldots x_k])_n$, attaching the labels to the sigma, thus avoiding brackets:

$$\Sigma[x_1, \ldots, x_k] \rightarrow \Sigma_n[x_1, \ldots, x_k] \quad (n \text{ fresh})$$

Labeling is performed by an operator $\text{lift} : \text{Term}(A_{\delta, \epsilon}) \rightarrow \text{Term}_N(A_{\delta, \epsilon})$. In order to formalize the idea of ‘fresh’ labels we need an operator which carries the labels along as an argument and as a result. Therefore we employ a two-argument version of it, $\text{lift} : \text{Term}_N(A_{\delta, \epsilon}) \times \mathbb{N} \rightarrow \text{Term}_N(A_{\delta, \epsilon}) \times \mathbb{N}$ defined by:

$$\begin{align*}
\text{lift}(s_{p,q}(m), n) &= (s_{p,q}(m), n) \\
\text{lift}(r_{p,q}(m), n) &= (r_{p,q}(m), n) \\
\text{lift}(\Pi x, n_i) &= (\Pi x', n_o) \quad \text{where } (x', n_o) = \text{lift}'(x, n_i) \\
\text{lift}(\Sigma x, n_i) &= (\Sigma_{n_i} x', n_o) \quad \text{where } (x', n_o) = \text{lift}'(x, n_i + 1)
\end{align*}$$

where the auxiliary operator $\text{lift}' : \text{Term}_N(A_{\delta, \epsilon}) \times \mathbb{N} \rightarrow \text{Term}_N(A_{\delta, \epsilon}) \times \mathbb{N}$ works on lists as follows (please read $n_i$ and $n_o$ as $n$-in and $n$-out, respectively):

$$\begin{align*}
\text{lift}'([], n) &= ([], n) \\
\text{lift}'(x y, n_i) &= (x', y', n_o) \quad \text{where } (x', k) = \text{lift}(x, n_i)(\text{for some } k \in \mathbb{N}) \wedge (y', n_o) = \text{lift}'(y, k)
\end{align*}$$

**Definition 3.3** $\text{lift}(x) = \pi_1(\text{lift}(\text{pck}(x), 0))$ where the projection operator $\pi_1$ takes the first argument of a pair. So we start with label 0 and in the end we discard the resulting current label value. $\square$

**Example 3.4** The following term is obtained by applying the previous transformation steps to the process $x$ of Example 2.8: $\Pi[\Sigma[r_{p,x}(m_1), r_{p,x}(m_2)], \Sigma[r_{q,x}(m)], s_{x,p}(a)]$ Lifting this term yields $\Pi[\Sigma_0[r_{p,x}(m_1), r_{p,x}(m_2)], \Sigma[r_{q,x}(m)], s_{x,p}(a)]$. $\square$

The fourth step (re) is to unwind the term obtained so far, bringing its graph into tree-form by means of the right-expansion rule:

$$\begin{array}{c}
(x + y) \cdot z \\
\text{A4}^{-1}
\end{array} \rightarrow \begin{array}{c}
x \cdot z + y \cdot z
\end{array}$$

applied as a set of rules based on the $\Sigma$-list representation, together with $\Sigma A2$ and $\Pi A5$ (but not $\Sigma A6$) for flattening and $A3$ and $A7$ directed as contractions. We apply the rule in such a way that it works for singleton sums too, rewriring $(\Sigma_n[x]) \cdot z$ into $\Sigma_n[x \cdot z]$.

One purpose of the fourth step is to avoid a ‘strange’ FSM where two transitions share a common tail-part. Although using joins this could be modelled in SDL (see Figure 5), we prefer to avoid the joins as well. It may look useless to perform a right-expansion after having done a right-contraction first, but the point is that the packing and lifting takes place in between, leading to a reduced number of states.

Finally the fifth step (fold) is to find the transition set for each label. The transitions are described by equations, one for each transition. This step yields a pair $(e, E)$, where the first equation $e$ describes the initial transition and the set $E$ contains the equations of the remaining transitions. This step works by means of folding (let $x$ be the result of the fourth step and let $p, p_0, p_1$, etc. be distinct variables):
1. for the term \( x \) generate \( e \) given as \( p \overset{\text{def}}{=} x \),
2. for each subterm \( y \) labeled with \( n \), generate one equation \( p_n \overset{\text{def}}{=} y \),
3. in the right-hand side of these equations, working from left to right, whenever encoun-
tering a proper subterm \( \Sigma_n[\tilde{y}] \), replace it by \( p_n \).

The label-removal (\( \mid \) operator) can be built-in to this folding process easily (see the first rule below, whose premiss contains a labeled \( \Sigma \) whereas the resulting equation has an unlabeled \( \Sigma \)). Note that all subterms of the form \( \Sigma_n[\tilde{y}] \cdot x_1 \ldots x_k \) have been eliminated earlier. The rules combining 2., 3. and label-removal are:

\[
\begin{align*}
\Sigma_n[x_1,x_2,\ldots,x_m] & \rightarrow p_n \overset{\text{def}}{=} \Sigma[\text{strip}(x_1),\text{strip}(x_2),\ldots,\text{strip}(x_m)] \\
\text{strip}(a \cdot x) & \rightarrow a \cdot \text{strip}(x) \\
\text{strip}(\Sigma_n[\tilde{x}]) & \rightarrow p_n
\end{align*}
\]

The resulting set of equations is converted to a FSM description using a SDL-like notation\(^5\). If the pair \((e,E)\) is given by \((p \overset{\text{def}}{=} x,\{p_0 \overset{\text{def}}{=} x_0,\ldots,p_{N-1} \overset{\text{def}}{=} x_{N-1}\})\) then its

conversion is done by the following rule:

\[
\begin{align*}
\text{PROCESS } p; \\
\text{START;} \\
\text{translation of } x; \\
\text{STATE } 0; \\
(e,E) & \rightarrow \text{translation of } x_0; \\
\vdots \\
\text{STATE } N-1; \\
\text{translation of } x_{N-1}; \\
\text{ENDPROCESS } p;
\end{align*}
\]

where it is understood that 'translation of' works as follows:

\[
\begin{align*}
r_{q,p}(m) & \rightarrow \text{INPUT } m; \\
s_{p,q}(m) & \rightarrow \text{OUTPUT } m \text{ TO } q; \\
p_n & \rightarrow \text{NEXTSTATE } n; \\
a \cdot x & \rightarrow a \cdot x \\
'\text{end of product}' & \rightarrow \text{STOP};
\end{align*}
\]

\(^5\)Although we use the syntax of SDL, we must clearly state that the semantics of real CCITT/ITU SDL is based on asynchronous communication. For further discussion see [10].
Example 3.5 The following SDL text is obtained from Example 2.8:

```
PROCESS x;
  START;
  NEXTSTATE 0;

STATE 0;
  INPUT m2;
  NEXTSTATE 1;

INPUT m1;
  NEXTSTATE 1;

STATE 1;
  INPUT m;
  OUTPUT a TO p;
  STOP;
ENDPROCESS x;
```

The corresponding diagram was already shown in Figure 3.

Some of the FSMs obtained in this way do not satisfy a condition usually required for SDL processes (motivated by SDL's queueing approach, see [10]). We define a concept of processes being 'receive-guarded', by which we mean 'free of choice points which are not controlled by receive actions'.

**Definition 3.6** A term \( x \) is **receive-guarded** if in each sum the first actions of the summands are distinct receive actions.

The FSM obtained as \( S_p \) can only be converted to a FSM in SDL syntax if the application of transformation \( (rc \circ lc \circ \text{project}_p \circ \Sigma) \) to \( i \) yields a receive-guarded term. The only relevance of this is that for terms that are not receive-guarded, the output of our transformation cannot be fed into readily available SDL tools, e.g. for further diagram layout editing as we did during our study reported in [10] (we found the diagram syntax of SDL intuitive and useful).

Example 3.7 Consider the following terms, assuming that \( m \neq n \):

1. \( s_{p,q}(m) \cdot r_{q,p}(m) + s_{p,q}(n) \cdot r_{q,p}(n) \),
2. \( r_{q,p}(m) \cdot r_{q,p}(m) + r_{q,p}(m) \cdot r_{q,p}(n) \),
3. \( r_{q,p}(m) \cdot r_{q,p}(m) + r_{q,p}(n) \cdot r_{q,p}(m) \).

Then 1. and 2. are not receive-guarded. For example, 1. is not because its first actions are neither distinct nor receiving. Term 1. can be the result of combining two interworkings for which the target \( p \) has different behaviours and where there is no preceding receive action which can be assumed to be the cause of the difference. Term 2. cannot arise as the result of \( S_p \) because otherwise step lc would have done one more TR contraction. Term 3. is receive-guarded. Its branches are controlled by receive actions.
4 Propositions

In this section the propositions concerning synchronous communication are given: it is assumed that the constructed process $S_p(i)$ communicates synchronously with an environment. To make propositions about the behaviour of $p$ we consider certain 'test environments' (tester processes, modeled by some term $x$) and the traces $(t)$ of the combined system $x || p$. Three main theorems apply:

1. soundness of the transformation: in any environment, the FSM only exhibits traces that correspond to one of the given interworkings; this notion of soundness is motivated by the fact that it excludes the introduction of new unspecified behaviour by the transformation;
2. preservation of deadlock freedom: when testing the FSM in an environment derived from one of the interworkings, it will not deadlock;
3. completeness of the transformation: each of the interworkings can be evoked by the FSM together with a suitably chosen environment.

When considering asynchronously communicating FSMs, the FSMs obtained by the given transformation satisfy only much weaker properties. For a discussion of this we refer to [10]. The remainder of this section is about proofs. Readers wanting to skip these can continue with the conclusions of Section 5.

Definition 4.1 For $p \in EID$ define $A_{\delta,e}^- := A_{\delta,e} \setminus \{s_{p,q}(m), r_{q,p}(m)\}$.

So $A_{\delta,e}^-$ denotes the restricted action alphabet $\{s_{q_1,q_2}(m) \mid q_1 \neq p \} \cup \{r_{q_1,q_2}(m) \mid q_2 \neq p \} \cup \{\delta, e\}$. This restriction is useful because a term $x \in \text{Term}(A_{\delta,e}^-)$ can now be viewed as an arbitrary process, except that it cannot perform the communication actions of $p$ itself. Such $x$ will act as a 'test environment' for $p$ in the propositions to be formulated below.

Definition 4.2 Define a semantic function $[\cdot] : \text{Term}(A_{\delta,e,-}) \rightarrow \mathcal{P}(\mathcal{C}T)$ which assigns to each term $x$ the set of its traces $[x]$:

$$
\begin{align*}
[\epsilon] &= \{\epsilon\} = \{\delta\} = \{\lambda\} \\
[a] &= \{\lambda, a\} \quad (a \in A) \\
[r \cdot x] &= [\epsilon \cdot x] = [x] \\
[a \cdot x] &= \{\lambda\} \cup \{a \cdot t \mid t \in [x]\} \quad (a \in A) \\
[x + y] &= [x] \cup [y]
\end{align*}
$$

This is the same as in [6] (pp. 181) except for the notation $\text{tr}(x)$ (we use $[x]$) and the fact that we have extended the domain to allow $\epsilon$.

Intuitively, for a given term $x$, each of its traces represents a path from the root of the execution graph of $x$, where it is understood that:

1. a path need not be complete,
2. a trace contains no $\delta$: the process of following a path stops at $\delta$.
The definition of \([x]\) can be extended to terms containing the ACP operators \(|,\|\) and \(|\) in view of the fact that \(|,\|\) and \(|\) can be eliminated. Amongst the rules to eliminate them we mention \(\| y \| \rightarrow x \| y \| + x \| y \| + x \) (CM1), \(a\| x \| \rightarrow a \cdot x\) (CM2) and \((a \cdot x)\| y \| \rightarrow a \cdot (x\| y \|)\) (CM3). \(^6\)

We say that a finite system of \(1 + N\) equations consisting of \(p \overset{\text{def}}{=} x\) and \(p_i \overset{\text{def}}{=} y_i\) \((i = 0, \ldots, N - 1)\) is non-recursive if the variable \(p\) occurs neither in the term \(x\) nor in one of the terms \(y_i\) and if moreover for each \(i < N\) the variable \(p_i\) occurs not in \(y_0, y_1, \ldots, y_i\). Of course \(p\) and the \(p_i\) must be distinct variables. We shall only generate such non-recursive finite systems. We write \(y_i\) to indicate that the right-hand side of each of the equations is a term, which in principle is different for each \(i\). The leading variable \(p\) is special in the sense that it is only used at the left-hand side of the top-level equation.

**Lemma 4.3** A term \(p\) defined by non-recursive specifications \(p \overset{\text{def}}{=} x\) and \(p_i \overset{\text{def}}{=} y_i\) \((i = 0, \ldots, N - 1)\) has a normal form that is obtained by elimination of the variables \(p_i\).

**Proof.** Let us write \(x[p_i := y_i]\) for the term obtained from \(x\) by substituting \(y_i\) for all occurrences of \(p_i\). Now note that each \(y_i\) contains only variables from \(\{p_0 \ldots p_{N - 1}\}\). First eliminate \(p_0\), replacing \(p \overset{\text{def}}{=} x\) by \(p \overset{\text{def}}{=} x[p_0 := y_0]\) and replacing each \(p_i \overset{\text{def}}{=} y_i\) by \(p_i \overset{\text{def}}{=} y_i[p_0 := y_0]\). Similarly, eliminate \(p_1 \ldots p_{N - 1}\) substituting \(y_i\) for \(p_i\) etc. Each step may increase the size of the remaining right-hand side terms, but after \(N\) steps we find that \(p\) is given by just one equation of the form \(p \overset{\text{def}}{=} x'\) where no \(p_i\) occurs in \(x'\) (of course there are other reduction orders, such as rewriting \(p \overset{\text{def}}{=} x\) to \(p \overset{\text{def}}{=} ((x[p_{N - 1} := y_{N - 1}])[p_{N - 2} := y_{N - 2}]) \ldots [p_0 := y_0]\), but they all lead to the same outcome).

When comparing a trace \(t\) with a specified trace \(i\) it is convenient to write \(t \leq_p i\), expressing that \(t\) is a prefix of \(i\), except for the fact that \(t\) may contain certain irrelevant steps. The irrelevant steps are steps which are not related to \(p\) and which correspond to internal actions of the test environment \(x\) of \(p\). In other words, the 'p-projection' of \(t\) is a prefix of the 'p-projection' of \(i\). This motivates the following definition.

**Definition 4.4.** We write \(t_1 \leq_p t_2\) if trace \(t_1\) is a prefix of trace \(t_2\). And for \(p \in EID\) we define \(t_1 \leq_p t_2\) to indicate that \(\text{proj}_p(t_1) \leq \text{proj}_p(t_2)\). \(\square\)

If we consider process graphs which are equivalent upto the ordering of the branches at the nodes as being the same, we say that we consider process graphs modulo \(AC\) (for Associativity and Commutativity). A term defined by a non-recursive specification has a unique tree-shaped process graph (modulo \(AC\)) and this graph has finite traces only.

Since for \(p \in EID\) the transformation result \(S_p(i)\) is a pair of the form \((e, E)\), where \(e\) is thus a defining equation for the leading variable \((e\) at its turn containing variables defined in \(E)\) Without loss of generality, we can assume that \(p\) itself is used as the leading variable. For convenience, we adopt this assumption. Now we could write \([p]\) to denote the set of traces of \(p\), but since \(p\) is only defined through the equations in \((e, E)\), a more explicit notation is \([p \mid S_p(i)]\) (the traces of \(p\) given its defining equations). For practical reasons however we shall use \([S_p(i)]\) as a shorthand for \([p \mid S(i)_p]\). Also, if \(x\)

\(^6\)For a discussion of proof techniques showing that these rewrites are strongly normalizing, see remark 4.11 in [11].
is a term without variables (for example modeling some test environment), we ought to write \( [x \| p \mid (e, E)] \) for the set of traces of the combined system \( x \| p \), but again we use \( [x \| S_p(t)] \) as a shorthand. And so on.

After these preparations we are ready to formulate the first main proposition about the behaviour of a process \( p \) obtained from \( i \) by means of the five steps of the transformation process \( S_p \). It says that whenever we offer a test environment \( x \) for process \( p \), finding \( t \) as one of the execution traces, then this \( t \) is in fact one of the behaviours already in \( T \), or at least a prefix thereof.

By way of preparation for this soundness theorem we present a lemma first which characterizes the possible behaviours of the first transformation step \( \text{project}_p \). Let us say that a term \( i \) is linear if it is a product of communication actions only.

**Lemma 4.5** For linear \( i = i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1} \) we have

\[
\forall t \in CT \forall x \in \text{Term}(A_{\tau P}) \Rightarrow t \in [\partial_H(x \| \text{project}_p(i))] \Rightarrow t \leq_p i
\]

where \( \partial_H \) with \( H \) defined as \{ \( s_{p_1, p_2}(m), r_{p_1, p_2}(m) \mid p_1, p_2 \in \text{EID}, m \in \text{MID} \} \) hides all internal send and receive actions.

**Proof.** By rule CTM1 of ACP, \( x \| \text{project}_p(i) \) is

\[
x \| (i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1}) = (i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1}) + (i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1}) + \sqrt{x} \cdot \sqrt{i'}
\]

where \( i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1} = i' = \text{project}_p(i) \) and where thus each \( i_j = \text{project}_p(i_j) \).

The lemma is proved by induction on the structure of \( i \) (or, which is the same in this case: by induction on \( l \)).

**Basis.** To prove the property for all \( t \) and for all \( x \), we consider some \( t \) and some \( x \) (arbitrary). We set \( l = 0 \) whence only \( x \| (i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1}) \), that is \( x \| \epsilon \), contributes to \( t \) and hence \( t \) consists only of actions of \( x \) that are not related to \( \text{project}_p(i) \) since \( \epsilon \). Hence \( t \) consists only of actions of \( x \) that are not related to \( p \) (if \( x \) is empty too, we get \( \sqrt{x} \cdot \sqrt{i'} \) which is empty). Thus \( t \leq_p i \) is true because such actions of \( x \) do not pass the \( \text{project}_p \) operator defining \( \leq_p \) and we only have to check that \( \lambda \leq \text{project}_p(i) \). Let us say that these actions of \( x \) are irrelevant with respect to \( \leq_p \).

**Induction step.** Consider some \( t \) and \( x \) (arbitrary). Now \( l > 0 \) and \( t \) can begin with some irrelevant steps, possibly followed by a trace resulting from

\[
x' \parallel (i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1})
\]

for a subterm \( x' \) of \( x \). Suppose that \( i_0 \) is a send action \( s_{p,q}(m) \) (if it is a receive action we have an analogous case). Then \( x' = r_{p,q}(m) \cdot x'' \) and

\[
x' \parallel (i_0 \cdot i_1 \cdot \ldots \cdot i_{l-1}) = (r_{p,q}(m) \cdot x'') \parallel (s_{p,q}(m) \cdot (i_1 \cdot \ldots \cdot i_{l-1}))
\]

where \( \text{tail}(i') \) denotes \( i_1 \cdot \ldots \cdot i_{l-1} \). Hence \( t \) is a trace of \( x'' \parallel \text{tail}(i') \). The first action of \( \text{project}_p(t) \) is \( s_{p,q}(m) \) which is precisely the first action of \( i' \). Therefore \( t \leq_p i \) if \( t' \leq_p \text{tail}(i) \), which follows by induction hypothesis (applied to \( i' \) and \( x'' \)).  \( \square \)
Theorem 4.6 (soundness). Let $\tilde{t} \in CT^*, p \in EID$, then

$$\forall x \in \text{Term}(A^c_p) \cdot \forall t \in [\partial_H(x \| S_p(\tilde{t}))] \cdot \exists i \in \tilde{t} \cdot t \leq_p i$$

where $\partial_H$ with $H$ defined as $\{ s_{p_1,p_2}(m), r_{p_1,p_2}(m) \mid p_1, p_2 \in EID, m \in MID \}$ hides all internal send and receive actions.

Proof. Instead of proving directly that the realization $S_p(\tilde{t})$ of $p$ is safe with respect to $\tilde{t}$ we first show that the initially constructed process $\text{project}_p(\tilde{t})$ is safe (after that we only have to show that the remaining four steps, $\text{rc} \circ \text{lc}$, lift, re and fold, preserve this soundness property).

Note that $\text{project}_p(\tilde{t})$ is a sum of traces, which we write as $i_0' + i_1' + \ldots + i'_{n-1}$. For each $k$, the trace $i'_k$ equals $\text{project}_p(i_k)$ and consists of send or receive actions of $p$ but contains no other actions.\footnote{Some care is needed here: in the previous lemma, $i'_j$ was a single action, but here we must use a similar notation $i'_k$ for an entire trace.}

The following is shown:

$$\forall x \in \text{Term}(A^c_p) \cdot \forall t \in [\partial_H(x \| \text{project}_p(\tilde{t}))] \cdot \exists i \in \tilde{t} \cdot t \leq_p i$$

Take an arbitrary $x \in \text{Term}(A^c_p)$ and consider an arbitrary trace $t$ of

$$\partial_H \left( \begin{array}{c}
  x \| (i_0' + i_1' + \ldots + i'_{n-1}) \\
  + (i_0'' + i_1'' + \ldots + i''_{n-1}) \| x \\
  + x \| (i_0'' + i_1'' + \ldots + i''_{n-1}) \\
  + \sqrt{x} \cdot \sqrt{(i_0'' + i_1'' + \ldots + i''_{n-1})}
\end{array} \right)$$

then $t$ consists of zero or more actions not related to $p$, followed by a trace $t'$ that does start with a communication action and that results from

$$x' \mid (i_0'' + i_1'' + \ldots + i''_{n-1})$$

for a subterm $x'$ of $x$. Note that the difference between $t$ and $t'$ is irrelevant with respect to $\leq_p$.

The first action of $t'$ results from one of the $i'_k$, e.g. because the first action of $i'_k$ is matching an action $r_{p,q}(m)$ of $x'$ whence the head of $t'$ is $c_{p,q}(m)$ and thus with respect to the first elements of $t'$ and $i'_k$ the condition $t' \leq_p i'_k$ is satisfied because $\text{project}_p(c_{p,q}(m)) = s_{p,q}(m)$. If $x'$ is sending instead of receiving we have an analogous case. Let $x' = r_{p,q}(m) \cdot x''$ and $i'_k = s_{p,q}(m) \cdot i''_k$, then

$$x' \mid i'_k = (r_{p,q}(m) \cdot x'') \mid (s_{p,q}(m) \cdot i''_k)$$

$$= c_{p,q}(m) \cdot (x'' | i''_k)$$

Then we only need to show that for $t'' \in [\partial_H(x''|i''_k)]$ the condition $t'' \leq_p i''_k$ holds. But this $i''_k$ is linear and the proof is by application of the above lemma.

The remaining four steps preserve the soundness property:

2. $\text{rc}$ and $\text{lc}$ consist of applications of the rules TR, A3, A4 and A7; of these, A3, A4 and A7 present no problem for they are laws of ACP, which is sound in the sense that $x =_{ACP} y \Rightarrow [x] = [y]$. TR is not a law of ACP but its use is justified by the fact that $x \xrightarrow{\text{TR}} y \Rightarrow [x] = [y]$.
3. lift only introduces harmless labels to be removed again later;
4. re consists of A4, A3 and A7, which are laws of ACP;
5. fold is harmless because it replaces equals by equals while introducing fresh variables;
   since no recursion is involved, fold preserves all ACP properties of the transformed term.

The next property concerns deadlocks, or better, their absence. We adopt the usual
inductive definition of 'x has a deadlock' for closed term x which is: (i) δ has a deadlock,
(ii) if x has a deadlock and a ∈ A, then every process a · x + y has a deadlock.

It follows that if x has no deadlock and a ∈ A then a · x has no deadlock. And if both
x and y have no deadlock, then neither has x + y.

Roughly speaking, the following proposition says that when p is triggered according
to one of the given i, no deadlock occurs.

**Theorem 4.7** (preservation of deadlock freedom). Let i ∈ CT*. For i ∈ ti define TESTi =
projectq1(i) || ... || projectqk(i) where the merge involves all entities q1, q2... occurring in
ports of i, except for p itself. Then

\[ ∀i ∈ ti ( (p′ || TESTi) has no deadlock) \]

where \( \partial_H \) again hides the internal send and receive actions.

**Proof.** Take an arbitrary i ∈ ti. Let p′ denote the initial process, being a sum of traces,
which we write as \( i_0' + i_1' + ... + i_{n-1}' \). For each k, the trace \( i_k' \) equals projectp(i_k). For
one value of k we have i = i_k; without loss of generality we assume that i = i_0.

In general, \( \partial_H(p′ || TEST_i) \) can have a deadlock, but the first transformation result
\( lc(p′) \) already satisfies the 'preservation of deadlock freedom' property.

Assume that \( i_0' \) has length \( l_0 \), whence we can write \( i_0' = i_0'[0] · i_0'[1] · ... · i_0'[l_0 - 1] \).
Similar for each k write \( i_k' = i_k'[0] · i_k'[1] · ... · i_k'[l_k - 1] \). For example if \( l_k = 2 \) (all k)
then \( lc(p′) \) is a kind of 'head normal form' with the typical form:

\[
\begin{align*}
  i_0'[0] & \cdot (i_0'[1] \\
  & + i_d_1'[1] \\
  & + ... \\
  & + i_d_{k-1}'[1] \\
  & + i_d_k'[1] \\
  & + ... \\
  & + i_d_k+l'[1] \\
  & + ...
\end{align*}
\]

which occurs if \( i_0'[0] = i_1'[0] = ... = i_d_{k-1}'[0] \neq i_d_k'[0] \), reflecting the fact that \( d_1 \) traces
\( i_0', ..., i_d_{k-1}' \) do not differ in their first action; here we assumed without loss of generality
that \( i_k' \) which have the same first action are adjacent in the ordering. The lc transformation

\[ ^{4} \] We need a lot of indexing here, so let us clarify the conventions: \( i_0', i_1', ... \) etc. are traces. Each trace
consists of a sequence of actions, viewed as a product, and we use square brackets to select an element of
such a sequence. Thus \( i_k'[0] \) is the first action of \( i_k' \), \( i_k'[1] \) is the second action of \( i_k' \) and so on.
has factored out their common prefix. And amongst those $d_1$ traces $i_k'$ which have the same first action as $i_0'$, there are $d_2$ traces which have the same second action as $i_0'$. And so on.

Now we have to analyze $\partial_H(lc(p')) || \text{TEST}_i$. To simplify the presentation we assume that (next to $p$) only two processes, $q_1$ and $q_2$ say, occur in $i_0$. Also assume that in the derivation of $\text{project}_{q_1}(i)$, $\text{project}_{q_2}(i)$ and $\text{project}_{q_2}(i)$ no unrelated (to $p$) steps had to be removed and that these processes do not communicate with each other (but only with $p$).

So $\text{TEST}_i$ is $\text{project}_{q_1}(i[0] \cdot i[1] \ldots \cdot i[l - 1]) || \text{project}_{q_2}(i[0] \cdot i[1] \ldots \cdot i[l - 1])$.

We prove that $lc(p') = lc(\text{project}_{p}(\Sigma \vec{f}))$ merged with $\text{TEST}_i$ has no deadlock, which we call property "$L(i)$". After that only the remaining steps must be shown to preserve deadlock freedom.

Although $lc(p') || \text{TEST}_i$ contains sums and merges, $\partial_H(lc(p') || \text{TEST}_i)$ is equal to a single trace (we write ". . ." to abbreviate some of the subterms in the typical form of $lc(p')$ given above):

$$
\partial_H(lc(p') || \text{TEST}_i) =_1 \partial_H \left( \begin{array}{c} i_0'[0] \cdot \ldots \\ + i_1'[0] \cdot \ldots \\ \vdots \end{array} \right) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) \\
= 2 \partial_H \left( \begin{array}{c} i_0'[0] \cdot \ldots \\ + i_1'[0] \cdot \ldots \\ \vdots \end{array} \right) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) + \\
\partial_H \left( \begin{array}{c} i_0'[0] \cdot \ldots \\ \vdots \end{array} \right) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) + \\
\partial_H \left( \begin{array}{c} i_0'[0] \cdot \ldots \\ \vdots \end{array} \right) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) \\
= 3 \delta + \\
\delta + \\
\partial_H \left( \begin{array}{c} i_0'[0] \cdot \ldots \\ + i_1'[0] \cdot \ldots \\ \vdots \end{array} \right) \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) + \\
\partial_H \left( \begin{array}{c} i_0'[0] \cdot \ldots \\ \vdots \end{array} \right) \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) + \\
\delta = 4 \partial_H \left( (i_0'[0] \cdot \ldots) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) \right) + \\
\partial_H \left( (i_0'[0] \cdot \ldots) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) \right) + \\
\cdots + \\
\delta = 5 \partial_H \left( (i_0'[0] \cdot \ldots) || \left( \text{project}_{q_1}(i_0) || \text{project}_{q_2}(i_0) \right) \right) + \\
\delta
where c is either $c_{p,q_1}(m)$ for some m (if p is the sender and q1 the receiver) or $c_{q_1,p}(m)$ (otherwise). The justification for the numbered equalities is:

1. replace $p'$ by its typical form, exhibiting its internal structure, which is that common prefixes have been factored out;
2. use CTM1 to expand the ‘|’ operator;
3. note that the actions of $i_0'[0]$, $i_0''[0]$, project$_{q_1}(i_0)$ and project$_{q_2}(i_0)$ are send or receive actions which are turned into $\delta$ by the $\partial_H$ operator (axiom D2); the $\gamma$'s yield $\delta$ too.
4. use A6 to eliminate the $\delta$s and CMS to distribute the ‘+’ over the ‘|’;
5. note that $i_0''[0]$ differs from $i_0'[0]$ and in view of the way by which both project$_{q_1}(i_0)$ and project$_{q_2}(i_0)$ are derived from $i_0$ we can conclude that $i_0''[0]$ cannot engage in a communication (it only contributes a $\delta$ by axiom CF2);
6. use A6 in order to get rid of the $\delta$ and make the internal structure of project$_{q_1}(i_0)$ visible;
7. note that project$_{p}(i_0)$, project$_{q_1}(i_0)$ and project$_{q_2}(i_0)$ are derived from the same trace $i_0$ and so for precisely one of the $q_j$ (j = 1, 2) we find that project$_{p}(i_0[0])$ and project$_{q_j}(i_0[0])$ form a matching pair, one being a send action, and the other a receive. Then use amongst others, CTM1 to expand the ‘|’”, CM9 to distribute the resulting ‘+’s, CM7 to see that ‘|’ applies, CF1 and CF2 to apply $\gamma$ and D1–D4 to effectuate $\partial_H$. Here we showed the case of $j = 1$. The case of $j = 2$ is analogous.

In this way we can continue the development of $\partial_H(lc(p'))|\text{TEST}_i|$, which thus equals a sequence of $c(m)$ actions (use induction on the length of $i_0$). After each of the equalities $\Rightarrow_5$ and $\Rightarrow_7$ only one term in the sum remains, so there is no choice (except for choices of the form $\ldots + \delta$). Therefore it has no deadlock.

If there are steps removed in the derivation of project$_{q_1}(i)$ and project$_{q_2}(i)$, we find a similar calculation except for the fact that the unique trace includes $\tau$ steps. If $q_1$ and $q_2$ communicate with each other, then the trace also includes actions $c_{q_1,q_2}$ and actions $c_{q_2,q_1}$ and if moreover there is also a $q_3$ and a $q_4$ then the trace is not unique any more (but its restriction to p still is!).

This proves $L(i)$. Next the remaining steps must be shown to preserve deadlock freedom.

2. rc consist of applications of the rules A4, A3 and A7; they are laws of ACP, whose equality is a congruence with respect to deadlock behaviour (note that the deadlock predicate is well-defined);
3. lift only introduces harmless labels to be removed again later;
4. re consists of A4, A3 and A7, which are laws of ACP;
5. fold is harmless because it replaces equals by equals. $\square$

The next theorem states that each behaviour of the constructed p can be obtained by triggering with a suitable $x$. 

406
Theorem 4.8 (completeness). Let $\bar{i} \in CT^*, p \in EID$, then
\[
\forall i \in \bar{i} : \exists x \in \text{Term}(A_1^p) : i \in [\partial_H(p||x) | S_p(\bar{i})]
\]
where $\partial_H$ with $H$ defined as $\{s_{p_1,p_2}(m), r_{p_1,p_2}(m) | p_1,p_2 \in EID, m \in MID\}$ hides all internal send and receive actions.

Proof. Take an arbitrary $i \in \bar{i}$. Then take $x = \text{TEST}_i$ as in the 'preservation of deadlock freedom' theorem.

Note that the successive $c_{p,q_1}(m)$ and $c_{q_1,q_2}(m)$ occurring in the result of the calculation of $\partial_H(|c(p')||\text{TEST}_i)$ are precisely the actions of $i$. ☐

5 Concluding Remarks

From Sections 2–4 we conclude that the algebraic approach forms a good framework for the specification, implementation and analysis of transformations related to interworkings, FSMs or both. In principle, the synthesis of FSMs from interworkings is feasible. Further refinements of the methodology are needed if one wants to respect all complications of the SDL semantics. We did not yet perform a completion of the rewrite rules to make the system confluent (Church Rosser). Further efficiency improvements are needed when scaling-up the approach. The synthesis of a FSM from interworkings is kind of ‘programming by demonstration’: in [13] it is argued that one of the present problems with programming by demonstration is that most often the recorded systems are not represented in a way that users can understand. Our approach to present the recorded system has two strong points with respect to readability: first, the system’s behaviour is not just recorded, but common subsequences are factored out; secondly the result is represented in the syntax of SDL which appeals to intuition and which can be manipulated with existing tools. In [9] related work is reported, but using a richer language and not in an algebraic context. In [14] the generation of interworkings from traces obtained by simulating PSF processes is studied; the SDT toolset for SDL can generate message sequence charts when simulating SDL processes. In a sense these are two translations in the opposite direction of ours.

The synthesis technique can provide feedback to a set of interworkings which are sketched in an early phase of system specification (as demonstrated in [10]). But the approach demands a large or even infinite set of interworkings to fix a process with really useful and non-trivial complexity. One could consider adding more powerful expression means to the interworking language or induction principles in the methodology. Finally we mention the area of test design, in which both sequence charts and FSMs already play important roles; this needs further investigation.

Historic notes: The work on the compiler began in late 1993 and in early 1994 a first experimental compiler was working. It revealed the critical-pair conflict between TR and A4 which was solved by doing the left-contractions and the right-contractions in two subsequent phases. The details were formalized using the theory of [6], as reported in [10] (Feb. 1995); this is also the main approach of this paper. In 1994 Mauw and Baeten independently proposed the delayed choice operator, which is now available as a theory for analysing the proper moment of choice [8]. As pointed out by Mauw, our critical-pair conflict can be viewed as a symptom of using one operator (+) for two types of choice: ≠ and the normal +. This explains why the two-phase approach works: the left-contraction
phase finds all instances of the first delayed choice rule \(a \cdot b \not\equiv a \cdot c = a \cdot (b + c)\). So after
that there are no more terms of the form \(a \cdot b \not\equiv c \cdot d\) for which \(a \neq c\) doesn’t hold and thus
the second delayed choice rule \(a \cdot b \not\equiv c \cdot d = a \cdot b + c \cdot d\) can be applied safely (as our
right-contraction does). Conversely, the compiler and Theorems 4.6, 4.7, 4.8 confirm the
delayed choice axioms for sets of interworkings.

Acknowledgements: The author wishes to thank Jan Bergstra, Jos van Wamel, Henk
Schepers, David Riemens, Sjouke Mauw, Michel Reniers and the two referees of ACP
'95 for their valuable feedback on earlier versions of this paper. Roel de Jong and Leen
Helmink offered useful advice on using Prolog.

References

[1] Rudolph E., Graubmann P., Grabowski J. Towards an SDL design methodology using
sequence chart segments. In: SDL '91, evolving methods, Færøergemand O. and Reed

In: SDL '93 using objects, Færøergemand O. and Sarma A. (Eds.), Elsevier Science

[3] CCITT. Recommendation Z100. Specification and description language (SDL). Docu-
ment AP IX-35-E, International Telegraph and Telephone Consultative Committee,


[9] Ichikawa H., Itoh M., Shibasaki M. Protocol-oriented service specifications and their
transformation into CCITT specification and description language, Transactions of

[10] Feijs L.M.G. Generating FSMs from Interworkings. Philips IST report RWB-510-re-


A Implementation Notes

Below we show the usage of lists as a uniform representation of ACP terms to be rewritten. We show this for the reduction relation $\Rightarrow^*$. $\Sigma$-Lists are used in combination with flattening rules for solving the problem of the commutative law for $\perp$, $\perp_1$, which cannot be interpreted as a rewrite rule because it would make the rewriting system non-terminating. At the same time the flattening rules can take care of associativity laws and therefore a uniform approach can be used, treating $\Sigma$-lists and $\Pi$-lists in a similar way. The approach is exclusively meant for the reduction of closed terms, i.e. terms in which no variables appear. This suffices for the automatic generation of FSMs from interworkings.

A list is of the form $[x_1, \ldots, x_n]$. The empty list $[]$ is considered a list too. We write vector notation $\vec{x}$ instead of $[x_1, \ldots, x_n]$ and we write $[x_1 \vec{y}]$ instead of $[x_1, y_1, \ldots, y_n]$.

The first four rules are a kind of ‘context rules’. Note the rules $\Sigma 2$ and $\Pi 2$ which are needed because the tail of a $\Sigma$-list must be viewed as a sub-term of the $\Sigma$-list itself, and moreover, the tail is treated as a $\Sigma$-list again.

The next four rules are ‘flattening rules’. Then there are two standard process-algebra rules and finally two rules based on the ‘trace’ axiom TR. We write a brace under the left-hand side of a rule to indicate that it stands for a set of rules, one for each permutation of the underbraced list.

$$
\begin{align*}
\Sigma[x_1] &\rightarrow x_1 \quad (\Sigma A 6) \\
\Pi[x_1] &\rightarrow x_1 \quad (\Pi A 8) \\
\Sigma[x_1, \ldots, x_n, x_{n+1}, \ldots, x_m] &\rightarrow \Sigma[x_1, \ldots, x_m] \quad (\Sigma A 2) \\
\Pi[\Pi[x_1, \ldots, x_n, x_{n+1}, \ldots, x_m] &\rightarrow \Pi[x_1, \ldots, x_m] \quad (\Pi A 5) \\
\Pi[\Sigma \vec{x}] &\rightarrow \Sigma \vec{x} \quad (\Pi A 7) \\
\Sigma[x, \Sigma \vec{y}] &\rightarrow \Sigma[x \vec{y}] \quad (\Sigma A 3) \\
\Sigma[\Pi[s\vec{v}], \Pi[s\vec{w}]] &\Rightarrow^* \Sigma[\Pi[s, \Sigma[\Pi \vec{v}, \Pi \vec{w}]], \vec{z}] \quad (\Sigma TR) \\
\Sigma[\vec{s}, \Pi[s\vec{w}]] &\Rightarrow^* \Sigma[\Pi[s, \Sigma[\Pi[]], \Pi \vec{w}]], \vec{z}] \quad (\Sigma TR')
\end{align*}
$$
The relation $\xrightarrow{\sim}$ is the smallest reflexive and transitive relation which is closed under the rules for $\rightarrow$ and for $\xrightarrow{\sim}$ given above.

We have built an experimental compiler using Prolog (which allowed us to program the parsing, rewriting and other transformations conveniently in a declarative style). Of course other high-level programming languages, programming environment generators or compiler-compilers are fine for this purpose too, for example ASF+SDF or Elegant. The experimental compiler is described in [10]. Here only a few interesting details are given. Receive terms $r_{q,p}(m)$ are represented as $r(Q,M)$ because the $p$ is always implicit. Similarly $s_{p,q}(m)$ are represented as $s(Q,M)$. Note that Prolog requires variables to start with an upper case letter.

The scanner is essentially taken from [15], see page 102–104. Parsing is done in the simplest possible way: the concrete syntax is transformed in abstract syntax by formulating the concrete syntax using the usual append operation defined in Prolog, exploiting the unification and backtracking power of Prolog (we had to add some cuts to get an acceptable error-recovery). Here we show part of the parser for an input sequence of interworking descriptions (IWDS) into a list of triples (ipe), each containing an INTERWORKING clause, a PROCESSES clause and an expressionlist.

\[
\text{iwds}([], []) . \\
iwds([ipe(I,P,E)|Rest]) :- \text{append}(IWDS, IWDS), \\
iwd(IWDS, I, P, E), \\
iwds(S, Rest) .
\]

\[
iwd(IWDS, XI, XP, XE) :- \text{append}(I, PE, IWDS), \\
\text{interworkingclause}(I, XI), \\
\text{append}(P, E, PE), \\
\text{processesclause}(P, XP), \\
\text{expressionlistclause}(E, XE) .
\]

The first process of the first interworking description is taken as being the target. The function filteriwds filters a set of interworking descriptions (IWDS) to get a linear action list for one process. In other words, filteriwds($p, i, Out$) binds $Out$ to $\text{project}_p(\Sigma i)$.

\[
generate(IWDS, Target, Out) :- \\
IWDS = [ipe(I, P, E)|Rest], \\
P = [Target|], \\
filteriwds(Target, IWDS, Out) .
\]

The reduction relations of the rewrite rules are mapped onto infix operators declared in Prolog. The notation and the approach of using Prolog for rewriting is similar to the approach described in [16] (pages 67–74). We show some rules for $\xrightarrow{\rightarrow}$ and $\xrightarrow{\sim}$. The binary relation $\sim$ means 'is permutation of'. Finally law is a tracing function writing to the console (it succeeds always).

\[
\text{select}(X, X[X], Xs) . \\
\text{select}(X, [Y|Ys], [YZ|Yzs]) :- \text{select}(X, Ys, Zs), \text{law}("A1") . \\
Xs = [Z|Zs] :- \text{select}(Z, Xs, Ys), Ys = [Zs] . \\
[] = [] . \\
\]

\[
\text{pi}(\{X\}) = X . \\
\text{sigma}(\{X\}) = X . \\
\text{sigma}(\{\text{sigma}(Y)\}|Z) = \text{sigma}(YZ) . \\
\text{pi}(\{\text{pi}(Y)\}|Z) = \text{pi}(YZ) . \\
\]

The reduction relations of the rewrite rules are mapped onto infix operators declared in Prolog. The notation and the approach of using Prolog for rewriting is similar to the approach described in [16] (pages 67–74). We show some rules for $\xrightarrow{\rightarrow}$ and $\xrightarrow{\sim}$. The binary relation $\sim$ means 'is permutation of'. Finally law is a tracing function writing to the console (it succeeds always).

\[
\text{select}(X, X[X], Xs) . \\
\text{select}(X, [Y|Ys], [YZ|Yzs]) :- \text{select}(X, Ys, Zs), \text{law}("A1") . \\
Xs = [Z|Zs] :- \text{select}(Z, Xs, Ys), Ys = [Zs] . \\
[] = [] . \\
\]

\[
\text{pi}(\{X\}) = X . \\
\text{sigma}(\{X\}) = X . \\
\text{sigma}(\{\text{sigma}(Y)\}|Z) = \text{sigma}(YZ) . \\
\text{pi}(\{\text{pi}(Y)\}|Z) = \text{pi}(YZ) . \\
\]
The notation and the approach of using Prolog for rewriting is similar to the approach described in [16] (pages 67–74). In the same style we coded the other transformation steps. Finally we added some pretty-print functions to improve the presentation of the output and a sorting function (putting states in increasing order).
A 3-level algebraic Ethernet specification

Jacob Brunekreef

Programming Research Group, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
e-mail: jacob@fwi.uva.nl

Abstract

In this paper we distinguish three different abstraction levels for protocol specification. The three-level approach is illustrated with a specification of the well-known Ethernet protocol. The specifications are given in the algebraic formalism ACP. At each level a deliberate choice for a particular specification style is made.

1 Introduction

The well-known ISO-OSI Reference Model for computer networks defines a hierarchy of seven network layers, from the physical layer (dealing with wires, electrical voltages, etc.) up to what is usually called the application layer (dealing with high level communication between computers, e.g. file transfer). In each of the seven layers a communication protocol may be defined, directed to fulfill the specific needs of that particular layer. The communication between protocol processes in adjacent layers can be defined in terms of so-called service primitives. The protocol process in the upper layer delivers a service request to the protocol process in the lower layer. This protocol process reports the result of the request by means of a service confirmation. A message is delivered to an upper layer at another peer entity by means of a service indication. In this model the protocol process in the upper layer acts as a client, requiring a service that is provided by the protocol process in the layer underneath.

Many (formal) protocol specifications can be found in the literature. Most of these specifications explain the basic concepts of a protocol in abstract terms such as the transmission/reception of a frame or the firing of a timeout. A specification presents a clear solution to the problem the protocol has been designed for (a lossy channel, multiple access to a channel). It shows the basic principles of the protocol without paying attention to details like how a frame is transmitted/received.

In case a formal verification of the protocol is given, also a specification of the required protocol behaviour is needed. Such a requirements specification contains a clear description of what service may be expected from the protocol. For many point-to-point data link protocols this service specification is fairly simple: a request containing a message is anticipated, sooner or later an indication with the same message is delivered at the other side. However, as we shall see in Section 4, for a Local Area Network protocol like the Ethernet protocol a requirements specification is by far not that simple.

Many protocol specifications are not based on (built from) services that are provided by a protocol in the layer underneath. Therefore, these specifications cannot be integrated in an environment with other protocols, e.g. in a protocol stack based on the OSI layers. In that case a complete and detailed specification of the protocol is required, based on the services that are provided by a protocol in the layer underneath. Very often such specifications are not very intelligible, due to the many details that have to be taken into account. However, we may expect that an implementation of a protocol
2 THE 3-LEVEL APPROACH

In this paper we present a framework for protocol specification within the ISO OSI Reference Model. For a protocol residing in a particular network layer, we will distinguish three different specification levels, based on a client-server model in which a protocol provides a service to a protocol in a layer above and uses a service from a protocol in a layer underneath. The paper is organised as follows. In Section 2 the three different abstraction levels for protocol specification are introduced. The 3-level approach is illustrated by a specification of the well-known Ethernet protocol ([IEE85].) Section 3 contains a short and informal introduction to this protocol. In the Sections 4 to 6 a specification of the Ethernet protocol at three different levels is given. Finally, some concluding remarks are presented in Section 7.

As far as we have noticed, not much work has been reported about a formal approach to the Ethernet protocol. In [SPE84] a formal verification, applying (extended) Temporal Logic, is given of the original XEROX Ethernet protocol, as described in [MB76]. In [Par88] an Ethernet-like protocol is specified (in CCS) and verified. In [MM90] and [Bru95] the Ethernet protocol is specified in the executable formalism PSF.

2 The 3-level approach

In this paper we look at the specification of communication protocols within the context of the ISO OSI Reference Model. This means that we consider a protocol to reside in one of the layers defined in the Reference Model. Furthermore, the communication between protocols in adjacent layers will be defined by a set of service primitives.

We distinguish the following levels of protocol specification:

1. At the service level the service that is required from the protocol is specified, e.g. an order-preserving fault-free communication between two network stations that are connected by an error-prone channel. A specification at this level is often called a requirements specification: the required external behaviour of the protocol is specified. Nothing is stated about the internal protocol actions.

2. At the conceptual level the solution that is offered by the protocol is specified in an abstract way: a frame is transmitted, a collision is detected. The details related to these protocol actions, as well as the interaction with a protocol in the layer underneath, are neglected.

3. At the full-layer level a protocol is specified in full detail. Excluding a protocol in the lowermost (physical) layer, a full-layer protocol will be specified in terms of services that are offered by a protocol that resides in the layer underneath. The protocol to be specified acts as a client with respect to the protocol in the layer underneath.

The three levels reflect the different phases in the design of a protocol (specification). First, the required external behaviour is formalised in a service level specification. This specification defines what services are available for a protocol in the layer above. Moreover, a service level specification plays a role in the verification/validation of a protocol.

A conceptual level specification shows the “solution” offered by the protocol, without the many details that come with a real-life specification. A specification at this level usually is fairly compact and thus more or less easy to understand. It plays a role in explaining the basic concepts of a protocol. And, equally important, a specification at this level is often used in a correctness proof of a protocol: it is formally proven that the specification satisfies the requirements, specified at the service level.

Finally, the abstract protocol design is refined to a concrete and complete protocol specification. A full-layer specification contains all that has to be specified. It is based on well-defined services,
provided by a protocol in the layer underneath. It can be used for the integration of the protocol in a protocol stack or for (a step in the direction of) an implementation of the protocol. Due to its many details, in most cases it will be very hard to prove the correctness of a specification at this level.

At each specification level a deliberate choice has to be made concerning the applied specification style. In Vissers et al. ([VSvSB91]) four different specification styles are introduced. Before discussing which style fits best to which specification level, we shortly list the four styles with their main properties:

1. In the monolithic style only observable actions are presented and ordered as an alternative composition of sequences of actions. The only constructs allowed are alternative composition, sequential composition and tail-recursion.

2. In the constraint-oriented style also only observable actions are presented, but their temporal ordering is defined by a conjunction of different constraints. Parallel composition and encapsulation are now also allowed as constructs. Hiding is forbidden, as all actions (or at least the resulting communications) have to be observable.

3. In the state-oriented style the system is regarded as a single resource, whose state space is explicitly defined. Observable actions are presented in an alternative composition of conditional process terms, in which also changes in the state space are reflected. The same language constructs as in the monolithic style are applied.

4. In the resource-oriented style the external behaviour of a system is defined in terms of the parallel composition of separate resources. Internal actions (between the resources) are hidden. Each resource may be specified using any style. As with the constraint-oriented style, parallel composition and encapsulation are central constructs. Additionally, in the resource oriented style hiding (abstraction) is applied to make internal actions invisible in the external behaviour.

In many ACP/PSF service level (protocol requirements) specifications the monolithic style is applied. Most notably, the external behaviour of point-to-point data link layer protocols like the Alternating Bit Protocol, the PAR protocol, Sliding Window protocols, etc., can be specified very well in a monolithic style. See [Bae90, MV93] for examples. However, as we will see in Section 4 of this paper, for the Ethernet protocol this style is not applicable. In more complex protocols the external behaviour can best be specified in a constraint-oriented style: different requirements concerning the external behaviour are specified in separate constraint definitions. A complete service level specification consist of a conjunction (parallel composition) of the constraint specifications.

Both a conceptual level specification and a full-layer level specification will require an identification of the separate “physical” processes that constitute a complete specification, e.g. a Transmitter process, a Receiver process, a Timer process, a Channel process. This can be captured in a resource-oriented specification of the various processes. Next, each constituting process will have to be specified. Within the ACP/PSF tradition these processes are specified in a monolithic style. Sometimes, when a process itself has a strong ‘state component’, a state-oriented style is chosen. It seems that at this level these two styles are in favour of the others. For verification purposes the state-oriented style may fit best, see further Section 7.

In constraint-oriented specifications a multiway synchronisation of atomic actions plays an important role. In different constraint specifications actions with the same name and argument(s) denote the same ‘physical’ action. An enforced synchronisation of these actions is required. Unlike CSP, ACP does not offer a parallel composition operator with enforced synchronisation. We will apply the ACP communication function together with the renaming operator and the encapsulation operator to model the required synchronisations. This will be further explained in Section 4.

In the remainder of this paper the 3-level approach is illustrated with a specification of the MAC
Introduction to the Ethernet protocol

The Ethernet protocol belongs to a class of communication protocols that is meant to provide a reliable communication on a multiple-access medium: a medium that is accessible to more than one network station. This kind of media (a bus, a ring) can be found in Local Area Networks (LANs): networks with a limited physical size (e.g. a building).

Several LAN protocols have been standardised by the ANSI/IEEE. The protocol that is usually referred to as 'the Ethernet protocol' is an instance of what is called a CSMA/CD (Carrier Sense Multiple Access with Collision Detection) protocol. In [IEEE85] this protocol is standardised, using service primitives for the interfaces between adjacent layers and Pascal-like pseudo-code for a functional definition of the protocol.

The Ethernet protocol includes a 'logical' part and a 'physical' part. The latter resides in the physical layer of the OSI Reference Model. This part of the protocol is completely outside the scope of this paper. The logical part resides in the Data Link layer of the OSI Reference Model. This part does not offer all data link functions as required by the OSI Model. Therefore, it is considered to constitute a sub-layer within the Data Link layer. It is referred to as the 'Media Access Control' (MAC) sub-layer. On top of the MAC sub-layer the 'Logical Link Control' (LLC) sub-layer is defined. The LLC sub-layer offers the data link functions that are not present in the MAC sub-layer.

Within the Ethernet protocol a network station 'senses' the medium to see if it is in use or not. If the medium is idle, a station is permitted to start transmitting. If two or more stations start transmitting at (almost) the same time, a collision occurs and the information on the medium has become worthless. In order to ensure the observation of the collision by all stations a jam signal is transmitted. Afterwards, the network station waits a certain time period before it restarts its transmission procedure. The waiting period is chosen at random from a time interval. The number of retransmissions is limited to a maximum (16 in the ANSI/IEEE document). If after this number of retransmissions still a collision is observed, an excessive collision error is reported to the LLC sub-layer. A receiving network station does not explicitly detect a collision. A collision will lead to the reception of a bad frame, which will be rejected by the receiving station.

The specifications in the coming sections will be based on the service primitives as defined in the ANSI/IEEE document. In the remainder of this section these primitives are shortly described.

The transfer of a message is initiated by a service request from the LLC sub-layer to the MAC sub-layer. This request contains three parameters: the destination address, the service data unit that has to be transported and a service class parameter. The last parameter only guarantees a compatibility with other LAN standard protocols. It is not used in the Ethernet protocol. The destination address may contain the address of a single network station, but also a group address or a broadcast address. In the last case a message is addressed to all network stations. In this paper we will pay no attention to group addresses.

A confirmation from the MAC sub-layer to the LLC sub-layer is used to report the result of the handling of a request. This service primitive has one parameter, indicating the success (transmitOK) or failure (excessiveCollisionError) of a request.

An indication from the MAC sub-layer to the LLC sub-layer is used to report the reception of a message. This service primitive has four parameters: the destination address, the source address, the transported service data unit and a 'receiver status'. The role of the last parameter is not clear from the Ethernet Standard. Several possible values are mentioned, e.g. frameCheckError or...
alignmentError, but otherwise it is explicitly stated that only correctly received frames are delivered to the LLC sub-layer. We will discuss this ambiguity in more detail in Section 7.

The communication between the MAC sub-layer and the physical layer is bit-oriented. A request from the MAC sub-layer to the physical layer contains one parameter. This parameter can take one of three values: 1, 0 or dataComplete. The first two values indicate the value of the next bit that has to be transmitted. The third value signifies that the MAC sub-layer has no more bits to transmit.

A confirmation from the physical layer to the MAC sub-layer also has one parameter. The value outputNext signifies that the physical layer is ready to accept a new bit (request) from the MAC sub-layer. The value outputAbort signifies that a request has failed.

There are several indication service primitives, sent from the physical layer to the MAC sub-layer. First, an indication is used to indicate the reception of a bit. Accordingly, its parameter can have two values: 1 or 0. Second, an indication signals a change in the carrier status. The parameter value carrierOn indicates that a transmission (or possibly more than one transmission) has started. The value carrierOff indicates that the carrier has gone off again. Third, an indication notifies a change in the signal quality. A parameter value signalError signifies the detection of a collision by the physical layer. The value noSignalError signifies the resolution of a collision (as far as the physical layer is concerned).

4 The Ethernet Service level

The external behaviour of the Ethernet protocol will be specified in terms of the service primitives that have been introduced in the previous section. A request for a frame transmission is 'read' from the service user (the local LLC sub-layer). A successful transmission leads to a positive confirmation, sent to the local LLC sub-layer, and to an indication, sent to the LLC sub-layer of the addressed station(s). An unsuccessful transmission (due to excessive collisions) will lead to a negative confirmation to the local LLC sub-layer only.

A specification of the external behaviour of the Ethernet protocol is less simple than one would expect (the reception of a request, followed by the delivery of a local confirmation and the delivery of an indication at the addressed station(s)). This is due to the following aspects of the required behaviour:

1. Autonomous buffering of requests (messages-to-be-sent) and indications (messages-to-be-delivered) at each local network station. We require that the acceptance of a request at network station A is not blocked by a not-yet-delivered indication at network station B. So, a monolithic process specification with a sequential composition of the acceptance of a request, followed by the delivery of one or more indications, will not do.

2. A global ordering on the delivery of indications concerning broadcast messages. We require that in the case of two subsequent broadcast messages, at all stations the message from network station A is delivered before or after the message from network station B. So, a process definition with a parallel composition of N sequential sub-processes ("read request · deliver indication") will not do.

3. The required 'collision behaviour': if at a certain moment a single request has been accepted at one network station only, a normal delivery of an indication and a positive local confirmation may be expected. However, if two or more requests are pending at the same moment, a collision error is possible.
We will consider each of the requirements stated above as a constraint and give an algebraic specification of it, leading to a constraint-oriented specification of the external behaviour of the Ethernet protocol.

Preceding the specifications, some notational remarks are in order. First, the communication between adjacent layers is modelled by read- and send-actions. These actions are denoted by \( r-XY \) and \( s-XY \), where \( X \) is an abbreviation of the source of the information and \( Y \) is an abbreviation of the destination. The following abbreviations are used in the specifications below: \( M \) stands for MAC sub-layer, \( L \) stands for LLC sub-layer. \( T \) denotes a Transmitter process, \( R \) denotes a Receiver process, \( P \) denotes a physical layer process. The set \( NS \) contains the station addresses. A station address is also used for station identification in process names and action names. \( ba \) is short for broadcast address. There is no station with identification \( ba \). The set \( NS' \) is equal to \( NS \cup \{ ba \} \). Finally, the set \( D \) contains the data elements that have to be transmitted. The names of the service primitives (request, confirmation, indication) have been abbreviated to \( req \), \( con \) and \( ind \). An underscore in a parameter list denotes an initial don't care value.

First, the 'transport constraint' (the acceptance of a request at network station \( A \), followed by the delivery of an indication at one or more network stations) is specified in the process \( MT \). This process is defined as being equal to the parallel composition of a number of sub-processes \( MT(s) \), one for each connected network station.

In this sub-process, after the reception of a request, two summands may be chosen. The first summand represents a successful transmission: a local delivery of a positive confirmation and the delivery of an indication, containing a message, at all addressed network stations. As these events may be interleaved in an arbitrary way, they are put in parallel. The second summand represents an unsuccessful transmission. In this case only a local negative confirmation is delivered.

The delivery of an indication is specified in the process \( MD(s, t, d) \), which is defined in a separate equation. Depending on the destination address in the request, an indication is delivered to one or to all network stations.

First, the 'transport constraint' (the acceptance of a request at network station \( A \), followed by the delivery of an indication at one or more network stations) is specified in the process \( MT \). This process is defined as being equal to the parallel composition of a number of sub-processes \( MT(s) \), one for each connected network station.

In this sub-process, after the reception of a request, two summands may be chosen. The first summand represents a successful transmission: a local delivery of a positive confirmation and the delivery of an indication, containing a message, at all addressed network stations. As these events may be interleaved in an arbitrary way, they are put in parallel. The second summand represents an unsuccessful transmission. In this case only a local negative confirmation is delivered.

The delivery of an indication is specified in the process \( MD(s, t, d) \), which is defined in a separate equation. Depending on the destination address in the request, an indication is delivered to one or to all network stations.

\[
MT = \bigoplus_{s \in NS} MT(s)
\]

\[
MT(s) = \sum_{t \in NS', d \in D} r-LM(s, req(t, d, _)) \cdot
(s-ML(s, con(transmitOK)) \| M D(s, t, d) + s-ML(s, con(excColl)) ) \cdot MT(s)
\]

\[
MD(s, t, d) = \bigoplus_{s \in NS} s-ML(v, ind(t, s, d, _)) \cdot M O(s, t, v, d) \cdot t = ba \lor s-ML(t, ind(t, s, d, _))
\]

The 'ordering constraint' is modelled in a simple process \( MO \). In case of a request with a broadcast address, after the delivery of one indication the enforced parallel composition of all other indication deliveries guarantees that no unwanted interleaving takes place at this point.

\[
MO = \sum_{s, t \in NS, v \in NS', d \in D} s-ML(v, ind(t, s, d, _)) \cdot M O(s, t, v, d)
\]

\[
MO(s, t, v, d) = \bigoplus_{v \in NS, v \neq s} s-ML(v, ind(t, v, d, _)) \cdot M O(s, t, v, d) \cdot t = ba \lor M O
\]

Finally, the 'collision constraint' is modelled in the process \( MC(n, c) \). Its first argument, \( n \), denotes the number of pending requests. If this number is greater than one, a collision may occur. However, once the occurrence of a collision has become possible, it remains possible until all pending requests that caused it have been (un)succesfully processed (\( n \) has become zero again). This is reflected in the second argument of the process: the boolean parameter \( c \) is set to \( true \) when \( n \) becomes greater than one, it is reset to \( false \) when \( n \) becomes zero again. This variable \( c \) determines whether or not a collision may occur.

\[
MC(n, c) = \sum_{s \in NS, t \in NS', d \in D, n > 0} r-LM(s, req(t, d, _)) \cdot MC(n + 1, n > 0) + \sum_{s \in NS} s-ML(s, con(transmitOK)) \cdot MC(n - 1, n > 1) \cdot n > 0 \lor \delta
\]
A complete specification of the Ethernet Service level is obtained by the parallel composition of the three constraint specifications. The required multiway synchronisation cannot be specified in ACP in a direct way, it will be obtained by a combination of renamings and encapsulation. First, all actions taking part in a synchronisation are renamed to a unique name. In the specification below this is done by adding a unique natural number as a subscript to the action name. So, an action \( a \) is renamed to \( a_i \). Next, a communication is defined for tuples of renamed actions. In case two actions take part in a synchronisation (as is the case in the example below), the result of the communication is defined as being equal to the original action: \( a_i | a_j = a \). The encapsulation operator is applied to the parallel composition of the constraint processes with the renamed actions. The encapsulation set contains the renamed actions \( a_i, a_j \). So we get

\[
ETH-Service = \beta_H (\rho_{R1}(MT) \parallel \rho_{R2}(MO) \parallel \rho_{R3}(MC(0, false)))
\]

R1 to R3 each denote a set of renamings on actions. For reasons of simplicity the data parameters are skipped.

- \( R1 = \{ rLM \rightarrow rLM_1, sML \rightarrow sML_1 \} \)
- \( R2 = \{ sML \rightarrow sML_2 \} \)
- \( R3 = \{ rLM \rightarrow rLM_3, sML \rightarrow sML_3 \} \)

The following communications are defined. Requests, indications and confirmations are abbreviated to \( rq, in \) and \( cf \).

- \( rLM_1(s,rq) \rightarrow rLM_3(s,rq) = rLM(s,rq) \)
- \( sML_1(s,in) \rightarrow sML_2(s,in) = sML(s,in) \)
- \( sML_1(s,cf) \rightarrow sML_3(s,cf) = sML(s,cf) \)

The encapsulation set \( H \) contains all the ‘subscripted’ actions. This set is not shown.

5 The Ethernet Conceptual level

As explained in Section 2, at the conceptual level an abstract specification of the protocol is given. A specification at this level focusses on the basic concepts of the protocol. The many details that come with a real-life protocol specification are neglected. In general, a specification at this level applies the service primitives that are used in the communication with the local protocol in the layer above. However, due to its abstraction from a number of aspects of the protocol, it does not apply the service primitives used in the communication with the layer underneath.

At the conceptual level and the full-layer level we will give a resource-oriented specification of the Ethernet protocol. The specification consists of a parallel composition of \( N \) network stations. Each network station holds two independent processes: a Transmitter process, taking care of the transmission of frames to the physical medium, and a Receiver process, taking care of the reception of frames from the physical medium. This identification of resources is independent of the level of abstraction of a specification. The process structure is depicted in Figure 1.

In the communication with the layer above (the LLC sub-layer) the same actions as in the previous section will be used, relating to the same service primitives. In the communication with the physical layer underneath a self-defined ‘abstract’ physical layer will be presupposed. This layer, together with the service primitives used in the communication with it, will be defined further on. First we will explain how the Ethernet protocol can be specified at a conceptual level. We will focus on two main aspects of the protocol: carrier sensing and collision detection.
5 THE ETHERNET CONCEPTUAL LEVEL

In a real-life CSMA/CD protocol a frame is transmitted bit by bit. For reasons of simplicity we will not specify a bit-wise transmission at the conceptual level. However, a frame-wise transmission (as applied in many conceptual specifications of Data Link layer protocols) will not work out fine: during the transmission of a frame a collision may come up, which has to be noticed by the Transmitter process. Therefore, a frame is transmitted in two (arbitrarily chosen) parts. The first part contains the destination address and the source address. The second part contains the service data unit.

When a station wants to transmit a data unit (a request is received from the LLC sub-layer), the first part of the frame is transmitted if the medium is idle. It is supposed that, once a first frame part has successfully been transmitted, all stations can see that the medium is in use and thus will refrain from sending. So, a collision (due to transmissions of other stations) will appear before this moment, it will not emerge afterwards. After the transmission of the first frame part, again the carrier is inspected by the transmitting station. If no collision has come up, the second part of the frame is transmitted and the transmission is regarded as successful. If a collision is detected, a jam signal is transmitted instead of the second part of the frame. This signal is meant to garble all information on the medium, so all stations will receive a garbled frame.

After the transmission of a jam signal a station waits some time before a frame is retransmitted. Different stations will have to wait different time intervals, otherwise the same collision is likely to occur again. At this level we will abstract from this real-time aspect: after a collision a station retransmits a frame without any waiting specified. The number of retransmissions is bound to a maximum. If after this number of retransmissions another collision occurs, the transmission fails.

A station continuously waits for an incoming frame. A frame is inspected as soon as it is received. If a correct frame is received, the destination address is inspected. In the case the station is addressed, the relevant frame parts are transferred to the LLC sub-layer. An invalid (garbled) frame is rejected without any notification to the LLC sub-layer.

For the communication with the abstract physical layer the following service primitives are defined:

- A request from the Transmitter process to the physical layer containing a frame part or a jam signal.
- An indication from the physical layer notifying the Transmitter process of a change in the status of the medium. Such an indication will also be considered as a response on a request.
• An indication from the physical layer notifying the Receiver process of the arrival of a frame (garbled or not).

The Transmitter process of a network station starts with waiting for a request from the LLC sublayer. When a request arrives, the first frame part is transmitted if the medium is idle. After the transmission of the first part the new medium status, communicated by an indication service primitive, determines what will happen next. If no collision is observed (status equal to on), a second frame part is transmitted. If a collision is detected (status equal to collision), a jam signal is transmitted instead.

When the transmission of a frame has been successful, the LLC sub-layer is notified by a confirmation service primitive with the parameter set to transmOK.

After a collision a frame is retransmitted. The number of retransmissions is bound to a maximum. In the specification of the Transmitter process the constant atlimit denotes this maximum. If this maximum is reached, the Transmitter process gives up. The LLC sub-layer is notified of this excessive collision behaviour by a confirmation service primitive with the parameter set to excColl.

At any moment in the action sequences described above, the Transmitter process has to accept an indication concerning a change in the medium status. In order to avoid an extra summand in each defining equation, a state-oriented specification style is applied: the whole Transmitter process is defined in a single recursive equation, applying the UNITY Format for ACP specifications (see [Bru94]).

The process CLT holds six data arguments:

• s: the station identity, a member of the set NS.
• ts: the process state with the possible values t1 to t5. Each of the states is shortly characterised in a comment line in the specification to follow.
• ms: the (most recently received) medium status.
• na: the number of attempted-but-failed (re)transmissions.
• t: the destination address of a pending request.
• d: the service data unit of a pending request.

The sets NS' and D have been introduced in the previous section. The set MS holds the medium status values idle, on and collision.

-- initialisation:

\[ CL\text{-TRANSMITTER}(s) = CLT(s, t_1, idle, ..., ...) \]

\[ CLT(s, ts, ms, na, t, d) = \]

-- t1: waiting for a request
\[ \sum_{tt \in NS'} \sum_{dd \in D} \cdot LT(s, req(tt, dd, ...)) \cdot CLT(s, t_2, ms, 0, tt, dd) \land ts = t_1 \land \delta \]

-- t2: transmit first frame part if medium is idle
\[ s \cdot TP(s, req(frame(t, s))) \cdot CLT(s, t_3, ms, na, t, d) \land ts = t_2 \land ms = idle \land \delta \]

-- t3: transmit second frame part if possible, transmit jam signal if required
\[ s \cdot TP(s, req(frame(d))) \cdot CLT(s, t_4, ms, na, t, d) \land ts = t_3 \land ms = on \land \delta \]
\[ s \cdot TP(s, req(jam)) \cdot CLT(s, t_5, ms, na, t, d) \land ts = t_3 \land ms = collision \land \delta \]

-- t4: deliver positive confirmation to LLC layer
\[ s \cdot TL(s, con(transmOK)) \cdot CLT(s, t_1, ms, na, t, d) \land ts = t_4 \land \delta \]

-- t5: retry transmission after collision or report excessive collision
5 THE ETHERNET CONCEPTUAL LEVEL

+ CLT(s,t2,ms,na + 1,t,d) \& ts = t5 \wedge na < atlimit \cdot \delta
+ s-TL(s,con(exCol)) \cdot CLT(s,t1,ms,na,t,d) \& ts = t5 \wedge na \geq atlimit \cdot \delta

- receive medium status indication in any state
+ \sum_{ms1 \in MS} r-PT(s,ind(ms1)) \cdot CLT(s,ts,ms1,na,t,d)

The Receiver process is simple. A complete frame is expected from the physical layer. Once a frame is received, it is checked on several aspects. Only the contents of a well-formed frame that is addressed to the receiving network station (individually or by broadcast address) are delivered to the LLC sub-layer by an indication service primitive. A garbled frame or a frame that is addressed to another station is rejected. The set FR holds all possible frames, including garbled frames.

- wait for incoming frame
  CL-RECEIVER(s) = \sum_{fr \in FR} r-PR(s,ind(fr)) \cdot CL-RC(s,fr)

- check frame
  CL-RC(s,fr) = CL-RECEIVER(s) \& fr = garbled \cdot CL-RCC(s,fr)

- check destination address
  CL-RCC(s,frame(t,v,d)) = s-RL(s,ind(t,v,d,)) \cdot CL-RECEIVER(s) \& t = s \vee t = ba \cdot CL-RECEIVER(s)

A 'complete' conceptual level protocol specification can be obtained by adding a service specification of an abstract physical layer process. Such a specification can be derived from the service level specification of the MAC sub-layer. With respect to the constraints listed in Section 4 we observe that the first constraint (autonomous local buffering of requests and indications) can be simplified: only one single message (frame) has to be buffered by the physical layer. A simultaneous request from another network station will immediately lead to a collision. It should be noticed that a frame is delivered by the Transmitter process in two parts. As only one frame is buffered at a time, the second constraint (order preserving of broadcast messages) is superfluous. At this level the collision constraint is more restrictive: when two or more network stations deliver a request (a frame part) at the same time, a collision will occur. After each frame reception or delivery, the change in medium status is reported to the Transmitter processes in the MAC sub-layer. This leads to the following service level specification of an abstract physical medium.

\[ PT = g_s ENS PT(s) \]

\[ PT(s) = \sum_{t \in ENS} r-TP(s,req(frame(t,s))) \cdot (s-PT(s,ind(on)) \cdot \sum_{d \in D} r-TP(s,req(frame(d))) \cdot (\bowtie s ENS s-PR(v,ind(frame(t,s,d)))) + s-PT(s,ind(collision)) \cdot r-TP(s,req(jam)) \cdot s-PT(s,ind(idle)) \cdot PT(s) + (s-PT(s,ind(on)) + s-PT(s,ind(collision)) + s-PT(s,ind(idle))) \cdot PT(s) \]

\[ PC(n) = \sum_{s \in ENS, t \in ENS} r-TP(s,req(frame(t,s))) \cdot PC(n + 1) + (\bowtie s ENS s-PT(s,ind(on))) \cdot (\bowtie s ENS s-PT(s,ind(idle))) \cdot PC(0) \& n = 1 \oplus \delta + (\bowtie s ENS s-PT(s,ind(collision))) \cdot (\bowtie s ENS s-PR(s,ind(garbled-frame))) \cdot (\bowtie s ENS s-PT(s,ind(idle))) \cdot PC(0) \& n > 1 \oplus \delta \]

As with the Ethernet service level specification of Section 4, a complete physical layer service level specification is obtained by an encapsulated merge of renamed versions of the constraint specifications:

\[ PL-Service = \delta_{HP}(r_{PR1}(PT) \parallel r_{PR2}(PC(0))) \]
A complete conceptual level protocol specification is given in the following equation:

\[ \text{CL-Ethernet} = \partial_{HC}(\exists_{ENs} (\text{CL-TRANSMITTER}(s) \parallel \text{CL-RECEIVER}(s)) \parallel \text{PL-Service}) \]

We skip the definitions of the various sets and communications.

6 The Ethernet Full-layer level

A full-layer level specification is based on the services that are offered by (a protocol in) the layer underneath. In the Ethernet protocol the actual physical layer offers a bit-wise transport service. This means that requests for the transport of a single bit are accepted from a MAC sub-layer. Transport process and indications containing single bits are delivered to a MAC sub-layer Receiver process. Any change in the medium status is reported by other indication service primitives. A request for the transmission of a bit is answered by a positive or negative confirmation. A positive confirmation indicates that the physical layer is ready to accept the next bit. The full-layer level Transport process and Receiver process have to be based on these service primitives.

In the CSMA/CD standard [IEEE85] the physical layer frame format is defined. In this paper we will skip the data specification that comes with the (dis)assembly of a frame. A detailed algebraic specification (in PSF) of the Ethernet frame (dis)assembly is given in [Bru95].

Before giving a specification of the Transmitter process and the Receiver process, we will summarise the functionality required of the MAC sub-layer, as described in the CSMA/CD Standard document.

First, a request has to be accepted from the LLC sub-layer. From the contents of the request a frame has to be assembled. This frame is presented to the physical layer bit-by-bit. The transmission of a frame is deferred until the medium is free. So, the Transmitter process has to perform a carrier sense function.

To ensure a proper observation of both a frame end and frame start by all network stations, a station is not allowed to start the transmission of a frame immediately after the moment it observes the medium to become idle. After this moment, a station has to wait an interframe gap period before a transmission may be started. In the CSMA/CD Standard a dedicated Deference process (re)sets a global boolean variable 'deferring'. Only if this variable holds the value false, the Transmitter process is allowed to start a transmission.

As soon as a transmitting network station detects a collision, the transmission of the remaining bits of the frame is cancelled. In order to be certain that all stations will detect the collision, a transmitting station does not immediately stop bit-transmission at the moment a collision is detected. First a jam signal is transmitted. After a collision no instant retransmission is tried. First, a network station waits a time period. This time period is chosen at random from a time interval. Due to the random choice from the time interval, there is only a limited chance that two colliding stations will also collide in their next attempt.

Retransmission is tried a limited number of times. If after this number of times a transmission is still unsuccessful, then the LLC sub-layer is notified of the failure of the execution of the transmission request, due to excessive collision. It should be noticed that this means that the Ethernet protocol does not provide an error-free transmission service under all circumstances.

At the receiver part of a station a frame has to be assembled from an incoming stream of bits. An incoming frame is inspected. In case of an error (checksum error, length error) the frame is rejected. Only the contents of a well-formed frame that is addressed to the receiving station are delivered to the LLC sub-layer.

In the CSMA/CD Standard document the Transmitter process is specified in a Pascal-like function TransmitFrame. The function operates on a number of global variables. These global variables
represent aspects both of the physical layer status (carrier on/off, collision) and of the internal state of the Transmitter process (deferring, frame waiting). Besides the function TransmitFrame two processes are specified in the Standard document: Deference (determines whether the transmission of a frame has to be deferred or not) and BitTransmitter (transmits a bit to the physical layer if the station is transmitting). These processes are supposed to run in parallel with the function TransmitFrame. They operate on the global variables too. No service primitives are used in the functional specification of the Transmitter process and the Receiver process.

In this paper we strive for specifications that are based on these service primitives. This means that the specifications to come are not a close translation of the various processes, functions and procedures of the CSMA/CD Standard document, but merely an interpretation. However, we claim that, besides some restrictions concerning real-time behaviour and complex arithmetic, the functionality of the specifications in this section covers the functionality of the CSMA/CD Standard.

A set of global variables determines (part of) the state of the Transmitter process. Some are set by indication service primitives from the physical layer. These indications have to be received in any state of the Transmitter process. Therefore, as with the conceptual level Transmitter process in the previous section, we have chosen a state-oriented specification style for the full-layer level Transmitter process with global variables as process parameters. Three global variables are modelled as boolean parameters:

- tsc: transmissionSucceeding. This parameter is set to true as soon as frame transmission is started. It is set to false when a collision is detected or the transmission fails because of a malfunctioning physical layer (indicated by a negative confirmation of a bit-transmission request). So, at the end of a transmission this parameter indicates whether the transmission was successful or a retransmission has to be scheduled.

- df: Deferring. This parameter is set to true as soon as the carrier goes 'on'. It is reset to false a certain time period after the carrier has gone 'off'. The Transmitter process is only allowed to transmit a frame to the physical layer if this parameter has the value false. In this way the 'interframe time-gap' is guaranteed.

- cs: carrierSense. This parameter indicates the carrier status: 'on' (true) or 'off' (false).

Besides these three parameters a state parameter ts with the values t1 to t6 is used to represent the various states of the Transmitting process. Each state is characterised in a comment-line in the specification to come. The Transmitter process has three more parameters: MR holds a request received from the LLC sub-layer, FR holds the not-yet-transmitted bits of a frame, na holds the number of previous frame transmission attempts. The function tr-data-encap constructs a frame-to-be-transmitted from a request. The names of several functions operating on a frame (first, tail, empty) need no further explanation.

Several real-time aspects of the Transmitter process can only be specified in (untimed) ACP in a symbolic way: the specification contains two symbolic atomic actions, wait-backoff-time and wait-deferring-time, modelling a wait-action during a certain time interval.

The process FLL-TRANSMITTER(s) is defined as equal to the initialising process term for the process TR. The first nine summands of TR cover the states t1 to t6. The remaining summands cover the reception of indications from the physical layer, concerning the carrier status and the signal quality. These summands are unconditional: the Transmitter process is open to these messages in any state.

\[ \text{FLL-TRANSMITTER} = \text{TR}(s,t1,\ldots,t6) \]
\[ \text{TR}(s,ts,MR,FR,na,tsc,df,cs) = \]
\[ -- t1: \text{waiting for a request} \]
In the CSMA/CD Standard document the Receiver process is specified by the Pascal-like function \textit{ReceiveFrame}. Like the function for frame transmission, this function operates on a set of global variables. A process \textit{BitReceiver} is supposed to run in parallel with the function. At this point the semantics of a Pascal-like specification conflicts with a specification that is based on service primitives. In Pascal the function \textit{ReceiveFrame} has to be called from elsewhere to become active. In the ANSI/IEEE document it is suggested that this call stems from the LLC sub-layer. After the reception of a frame the function returns a status value indicating the status of a received frame. The relevant elements from a frame (destination address, source address and service data unit) are returned by var-parameters.

However, in a service primitive–based specification the reception of a frame leads to an \textit{indication} from the MAC sub-layer to the LLC sub-layer. So, the initiative lies with the MAC sub-layer. The two different points of view can be integrated under the rather unnatural assumption that a permanent non-blocking function call is issued by the LLC sub-layer.

To make things even more complicated, the ANSI/IEEE document contains an inconsistency between the Pascal-like specification of the reception of a frame and the description of the related indication service primitive. The function \textit{ReceiveFrame} returns a status value with information about the received frame. The indication service primitive is only generated “if the frame is validly formed, received without error, and the destination address designates the local MAC entity”. In our specification of the Receiver process we have chosen to follow the second approach: only a correctly received frame that is addressed to the network station generates an indication to the LLC sub-layer.
SOME CONCLUDING REMARKS

The full-layer level specification of the Receiver process is also given in a UNITY Format. The state parameter rs can have three values: r1 to r3. The state r1 corresponds with the idle state: only a change in the carrier status is awaited. In the state r2 a frame is received bit-by-bit. In the state r3 a received frame is inspected and processed. As in the specification of the Transmitter process, indication messages from the physical layer can be received in any state.

The process FLL-RECEIVER is defined as being equal to the initialising process term for the process RC. The parameter RF holds a received frame (bit-string). The function concat adds a bit to a frame. The function receive-data-decap checks if the frame is well-formed and addressed to the receiving station. The name of this function comes from the ANSI/IEEE document. The functions da, sa and sdu extract the destination address, the source address and the service data unit from a received frame.

\[
FLL-RECEIVER = RC(s, r1, \ldots)
\]

\[
RC(s, rs, RF) =
\]

- carrier sense goes on: start receiving
  \[ r-PR(s, \text{ind}(cs-on)) \cdot RC(s, r2, empty) \]
- carrier sense goes off: stop receiving
  \[ + r-PR(s, \text{ind}(cs-off)) \cdot RC(s, r3, RF) \]
- r2: receive incoming bits
  \[ + \sum_{b \in B} r-PR(s, \text{ind}(b)) \cdot RC(s, rs, concat(b, RF)) \quad r s = r2 \lor \delta \]
- r3: handle received frame: deliver it if it is OK and the station is addressed
  \[ + s-RL(s, \text{ind}(da(RF), sa(RF), sdu(RF), \ldots)) \cdot RC(s, r1, RF) \]
  \[ \text{receive-data-decap}(s, RF) = \text{true} \lor \delta \]
  \[ \text{receive-data-decap}(s, RF) = \text{false} \lor \delta \]

In order to obtain a complete protocol specification at this level, a service-level specification of the physical layer behaviour is needed. Such a specification at least requires the following functionality:

- A bit that is transmitted from the MAC sub-layer to the physical layer is delivered to the MAC sub-layer of all network stations with a delay that corresponds with the network topology (the distance between the transmitting station and a receiving station).
- Medium status functions (carrier sense, collision detect) should be offered to the MAC sub-layer according to the propagation of transmitted bits through the network.

A formal specification of this functionality in untimed ACP is not impossible, but will not yield an intelligible specification. In our view it is clear that the required process has to be specified with a formalism that includes an explicit notion of time. So, at this level no complete protocol specification is presented.

7 Some concluding remarks

In this paper a 3-level approach to protocol specification has been presented. In this approach the external behaviour of a protocol, the basic ideas behind a protocol and, finally, the complete protocol with all its details are specified separately. The Ethernet protocol served as a running example and as a testcase for this approach.

At each level an appropriate specification style has to be chosen. In general we may expect a monolithic style or, in more complex cases like the Ethernet protocol, a constraint-oriented style in service level specifications. In conceptual level specifications and full-layer level specifications we may expect a resource-oriented style with the resource processes specified in various styles. A
constraint-oriented specification requires multiway synchronisation. This 'feature' is not present in ACP. However, it could rather easily be implemented by ordinary communications in combination with renamings and encapsulation.

Both the external behaviour of the Ethernet protocol and an 'abstract' Ethernet protocol could be specified in untimed ACP without any major difficulty. However, in the specification of the complete Ethernet protocol the lack of an explicit notion of time made it impossible to model all the required functionality and a service-level physical layer process. Probably a switch to timed ACP ([BB91]) would work out as an improvement.

It is an interesting question whether or not the conceptual level specification of the Ethernet protocol can be proven correct with respect to the service level specification. (As the full-layer level specification is not complete, at this level verification is out of order.) In our view this will require an approach as applied in several µCRL correctness proofs, see e.g. [BG93, GvdP93]. In this approach processes are specified in a state-oriented style. This may be an argument pro selecting this style at the specification stage.

Last but not least: formalising the CSMA/CD protocol of [IEE85] has brought to light several inconsistencies and obscurities in this standard. The most important inconsistency is the dual view on what should be done with a received erroneous frame (discard it? deliver it with an error code?). An example of an obscurity (also reported in [WZ92]) is the absence of the service primitive value 'data-completed' in the specification of the bit-transmission process and the effect of a physical layer request with this value on the global variable 'transmitting'. These examples show the advantages of the application of formal specification techniques in developing protocol standards.

Acknowledgements Alban Ponse is acknowledged for suggesting the 'ACP-implementation' of multiway synchronisation.

References


REFERENCES


Process Algebra with Language Matching

Extended Abstract *

Jos van Wamelen †
Department of Mathematics and Computer Science
University of Amsterdam
The Netherlands

Abstract
An axiom system $ACP_{LM}$ is presented as a variant of the process algebra $ACP$. The acronym $ACP_{LM}$ stands for $ACP$ with abstraction, extended with operators and axioms for language matching. Language matching is a technique based on trace information for labelling and cutting off process terms that do not match some given language, or set of traces. It is shown that in combination with the axioms for action alphabets interesting results are derivable, the most important of which is the Redundancy Theorem 3.3.6, which roughly states that if no trace labels occur in the expression $\Delta_H(p_1 \parallel q)$, where $p_1$ is a labelled version of some process $p$, then it holds that $\Delta_H(p_1 \parallel q) = \Delta_H(p \parallel q)$. It is shown that under certain natural conditions a similar result holds when abstraction is applied to $p_1$ and $p$.

In order to illustrate the use of language matching the Concurrent Alternating Bit Protocol (CABP) is verified. The CABP is a simple communication protocol, which can be recursively specified over the signature of $ACP'$, and it may be regarded as a slightly more 'sophisticated' variant of the well-known Alternating Bit Protocol. Earlier studies of this protocol have already demonstrated that 'redundancies in a context' make this protocol hard to verify. Our verification is carried out by means of $ACP'$ with language matching, extended with the abstraction rule CFAR*, and the conditional alphabet axioms. The verification is split into two parts so that it is 'modular', and in the verification some basic knowledge about the expected behaviour of the system is used, in order to allow the effective application of language matching.

1 Introduction

In process specification and verification in the process algebra $ACP$ it often happens that a process $p$ has 'redundancies in a context $\Delta_H(\_ \parallel q)$'. This means that $p$ has certain subterms (or from a semantic point of view: behaviour)

* [28] contains a full version of this paper; nearly all proofs that are omitted here are there elaborated in detail. A slightly revised version will appear in [29].
† Partly supported by the Foundation for Computer Science in the Netherlands (SION) and by the Netherlands Organisation for Scientific Research (NWO).
that are (is) eliminated in the context $\partial_H(\cdot \ | \ q)$, due to encapsulation $\partial_H$. Of course, redundancy can simply be a result of 'over-specification', i.e. when subterms are specified that are always encapsulated, and which do not communicate with subterms of $q$. We illustrate redundancy of summands in a given context with the following nice example, borrowed from [23].

**Example 1.0.1.** One of the famous coffee machine specifications from process algebra is given by

$$CM = 30c' \cdot (coffee' + tea') \cdot hum \cdot CM.$$ 

This machine waits for insertion of 30 cents, and after that it only proceeds if either coffee or tea is selected. After selection of a drink it starts humming, and finally enters its initial state again.

A typical, very thirsty Dutch user of such a machine is specified by

$$DU = (coffee + 30c \cdot coffee) \cdot talk \cdot DU.$$ 

Let the communications be defined by $\text{coffee} \ | \ coffee' = coffee^o$, $\text{tea} \ | \ tea' = tea^o$, and $30c \ | \ 30c' = 30c^o$, and $H$ by $H = \{coffee, coffee', tea, tea', 30c, 30c'\}$. The interaction between the Dutch user and the coffee machine is described by

$$\partial_H(DU \ | \ CM) = 30c^o \cdot coffee^o \cdot (talk \ | \ hum) \cdot \partial_H(DU \ | \ CM).$$

It is easy to see that the thrift of the Dutch user makes no sense here, i.e., the first alternative, trying to get the coffee for free, is redundant in the context $\partial_H(\cdot \ | \ CM)$. If we define a normal thirsty user as

$$TU = 30c \cdot coffee \cdot talk \cdot TU$$

we easily find (using RSP) that $\partial_H(DU \ | \ CM) = \partial_H(TU \ | \ CM)$. □

In the above example, redundancy is the result of 'over-specification'. Redundancy may also occur in modular specifications and verifications, where parts of a specification are studied separate from their contexts. In general, redundant terms will not have a simple structure. Therefore, a reason to search for possible redundancies at an early stage of a verification is that further analysis of such terms may not be necessary. This may be useful particularly for the 'expansion' of process expressions to linear equations.

It is usually the case that some information concerning the behaviour of a process in its context is available in the form of trace behaviour. Basic statements of the form "the action $a$ is supposed to happen immediately after $b$" can be regarded as useful trace information. In the example above for instance, $coffee^o$ is not expected to happen before $30c^o$, which seems a reasonable assumption for many coffee machines. It follows immediately from the system equation for $\partial_H(DU \ | \ CM)$ that this assumption is correct.

In Section 3 we develop a method for finding and labelling process terms that are possibly redundant in a given context, given some information about the expected behaviour of the total system in the form of a collection of process traces. \(^1\) We refer to our method as *language matching*. The labelling technique

---

\(^1\)In fact, simulation is also based on this type of knowledge; a simulation of a process is usually nothing else than an automatic generation of certain traces. It is obvious that simulation only makes sense if there are assumptions about the process behaviour.
used in language matching itself consists of a mechanism for replacing process terms that do not match some given \textit{language}, or set of traces, by a special atomic action $r$. For modelling language matching a special process operator is introduced, and its interaction with the operators from ACP is studied.

In Section 4 we illustrate the technique of language matching by means of a large case study: we verify the Concurrent Alternating Bit Protocol as specified and verified earlier in [10, 14, 26]. This protocol is a more complicated variant of the well-known Alternating Bit Protocol (ABP).

The ABP is a simple communication protocol and there are many verifications in the literature, because it is often used as the test case, either for some algebraic formalism or for some tool for the analysis or verification of concurrent systems. What makes this simple protocol so attractive is that it contains many interesting ingredients for concurrency theory. To mention a few: the channels make non-deterministic choices (between successful and unsuccessful delivery of messages), there is communication between components, and in a verification abstraction as well as some fairness assumption is needed in order to hide internal activity of the protocol.

Verifications of the ABP in the formalism ACP can be found in, for instance, [8] and in [22]. Verifications in related formalisms can be found in e.g. [18]. Such a verification can be carried through quite easily by straightforward calculations, although it has recently been demonstrated how a really formal verification leads to a amazing amount of calculations (see [13]). However, the number of states of the ABP is limited, and even without any abstraction from internal actions the process graph is easy to draw.

This is not the case with the CABP, since it suffers from a modest state space explosion, which is an exponential increase of the number of process terms when a parallel composition is being linearised. This state space explosion is a result of the fact that the protocol contains a lot of parallel, internal activity, caused by continuous streams of messages, which are necessary for recovering from error situations. The ABP on the other hand contains no parallelism in the sense that two components can be active, i.e., perform atomic actions at the same time. Our goal now is to verify the CABP equationally, and to control this ‘explosion’ with the help of language matching.

\textbf{Related work.} There are certain parallels between our work and other, previous publications on process operators for ACP, based on trace information. As language matching is particularly interesting in a setting with redundant process behaviour, there are links with [23], where a technique is presented for detecting ‘redundancy in a context’. Language matching is, in fact, also a technique for doing this, although it differs essentially in one respect. Language matching can detect possible redundancies of a process $p$ in a context $\partial_H(- \parallel q)$, using the term $\partial_H(p_1 \parallel q)$, where $p_1$ is some labelled, reduced version of $p$, instead of $\partial_H(p \parallel q)$.

We also refer to [1], where operators for localisation and restriction are studied. Although they do not satisfy the definition for state operators in [1], our operators for language matching are quite similar. Both the operators for language matching and state operators have some global state, a ‘memory’, as a parameter. For an interesting application of state operators see [7].

As for the CABP, one can imagine that channels in a protocol may sometimes contain more than one datum. Contrary to the ABP, this can indeed
happen in the CABP. In [10] (and [12]) a specification and verification of a version of the CABP is given with channels that behave as FIFO queues with unbounded capacity that can lose data. The possibility of communicating checksum errors at the receiver is not included, and an extra operator, the chaining operator, is added to ACP to make a short and elegant verification of this version of the CABP possible ('chaining' is a technique to easily connect two communicating processes via a data transfer link). Another difference is that in [10] no non-determinism in the channels is specified.

A second, totally different approach to the protocol was taken in [13], where an analysis of a specification in a timed variant of ACP is made. For timed versions of ACP the reader is referred to e.g. [2, 3].

The version of the protocol verified in this paper originates from [14]. An executable version in the specification language PSF contains a computer readable version of ACP, see for instance [16, 17] - can be found in [27]. In [14] the correctness is proved in weak bisimulation semantics. There the protocol is split into two modules and certain properties of the process graphs of the modules are studied. We adopt the idea of modularity as presented in that paper and algebraically verify the correctness of the same specification.

Acknowledgements. Thanks to Joris Hillebrand for Figure 4.1, and thanks to Jan Bergstra, Willem Jan Fokkink, Henri Korver, Alban Ponee, Piet Rodenburg and Chris Verhoef for helpful comments.

2 Preliminaries

In this section we give a brief overview of the axioms and rules from process algebra that are needed for understanding this paper.

2.1 ACP with abstraction

The formalism we use throughout this paper is ACP', the Algebra of Communicating Processes with abstraction (see e.g. [5, 6, 8]). We assume that the reader is familiar with this formalism. Moreover we assume some familiarity with the following conventions and concepts:

- A set of atomic actions \( A \) with typical elements \( a, b, c, \ldots \) is assumed. For process variables symbols \( x, y, z, \ldots \) are used, and for closed process terms \( p, q, r, \ldots \)

- All closed terms over the signature \( \Sigma(ACP') \) are provably equal to terms of the form \( a, \delta, r, p + q, a \cdot p, \) or \( r \cdot p \). Terms inductively built from terms with these special forms are known as basic terms (over BPA).

- For all closed terms over \( \Sigma(ACP') \) commutativity and associativity of the parallel operator \( \parallel \) are derivable: \( p \parallel q = q \parallel p \) and \((p \parallel q) \parallel r = p \parallel (q \parallel r)\). For open process terms these identities are known as axioms for standard concurrency (SC).

- With every closed term a (labelled) transition system is associated. The well-known equivalence relation between transition systems, related with
2.2 Recursion

We introduce processes defined by recursion equations.

**Definition 2.2.1.** A recursive specification \( E = \{ x = t_x \mid x \in V_E \} \) over some process algebra signature \( \Sigma \) is a set of equations where \( V_E \) is a set of variables and \( t_x \) are terms over \( \Sigma \) such that \( V_E \) contains the variables in \( t_x \).

A solution of a recursive specification \( E = \{ x = t_x \mid x \in V_E \} \) is an interpretation of the variables in \( V_E \) as processes, such that the equations of \( E \) are satisfied. For instance, the recursive specification \( \{ x = x \} \) has any process as a solution for \( x \), and \( \{ x = a \cdot x \} \) has the infinite process \( a \cdot x \) as a solution for \( x \).

The following syntactical restriction on recursive specifications turns out to enforce unique solutions (modulo rooted branching bisimilarity).

**Definition 2.2.2 (Guardedness).** Let \( t \) be a term over some signature \( \Sigma \), and \( E = \{ x = t_x \mid x \in V_E \} \) a recursive specification over \( \Sigma \).

- An occurrence of a variable \( x \) in \( t \) is guarded if \( t \) has a subterm of the form \( a \cdot M \) with \( a \in A \), where \( x \) occurs in \( M \), and there are no occurrences of abstraction operators \( \tau \) in \( M \).
- The specification \( E \) is syntactically guarded iff all occurrences of variables in the terms \( t_x \) are guarded.
- The specification \( E \) is guarded iff there is a syntactically guarded specification \( E' = \{ x = t'_x \mid x \in V_E \} \) over \( \Sigma \) such that \( t_x = t'_x \) for all \( t_x \).

Now the signature \( \Sigma_{REC} \), containing representations of recursively defined processes, is defined as follows.

**Definition 2.2.3.** The signature \( \Sigma_{REC} \) is obtained by extending the signature \( \Sigma \) in the following way: for each guarded specification \( E = \{ x = t_x \mid x \in V_E \} \) over \( \Sigma \) a set of constants \( \{< x \mid E > \mid x \in V_E \} \) is added, where the construct \( < x \mid E > \) denotes the \( x \)-component of a solution of \( E \).

Some more notations: let \( E = \{ x = t_x \mid x \in V_E \} \) be a guarded specification over \( \Sigma \), and \( t \) some term over \( \Sigma_{REC} \). Then \( < t \mid E > \) denotes the term \( t \) where each occurrence of a variable \( x \in V_E \) is replaced by \( < x \mid E > \), e.g., the expression \( < a \cdot a \cdot x \mid \{ x = a \cdot x \} > \) denotes \( a \cdot a \cdot < x \mid \{ x = a \cdot x \} > \).

For the constants of the form \( < x \mid E > \) there are two axioms in Table 1. In these axioms the letter \( E \) ranges over guarded specifications. The axiom REC states that the constant \( < x \mid E > \) is a solution for the \( x \)-component of \( E \), so it expresses that each guarded recursive system has at least one solution for each of its (bound) variables. The conditional rule RSP (Recursive Specification
Principle) expresses that \( E \) has \textit{at most} one solution for each of its variables: whenever one can find processes \( p_x \) satisfying the equations of \( E \), notation \( E(p_x) \), then \( p_x = \langle x|E \rangle \).

\[
\text{(REC) } \quad \langle x|E \rangle = \langle t_x|E \rangle \quad \text{if } x = t_x \in E
\]

\[
\text{(RSP) } \quad \frac{E(p_x^z)}{p_x = \langle x|E \rangle} \quad \text{if } x \in \mathcal{V}_E
\]

Table 1: Axioms for guarded recursive specifications.

A convenient notation is to abbreviate \( \langle x|E \rangle \) for \( x \in \mathcal{V}_E \) by \( X \) once \( E \) is fixed, and to represent \( E \) only by its REC instances. We write \( \mathcal{W}_E \) for the set \( \{\langle x|E \rangle | x \in \mathcal{V}_E\} \).

The following example shows all notations concerning recursively specified processes, and illustrates the use of REC and RSP.

\textbf{Example 2.2.4.} Consider the guarded recursive specifications \( E = \{x = a \cdot x\} \) and \( E' = \{y = a \cdot y \cdot b\} \) over \( \Sigma(ACP') \). So by the convention just introduced, we write \( X = a \cdot X \) and \( Y = a \cdot Y \cdot b \). With REC and RSP one can prove \( ACP^* + \text{REC} + \text{RSP} \vdash X = Y \) in the following way. First note that \( X \cdot b = a \cdot X \cdot b \) by REC, so \( E(X \cdot b) \) is derivable. Application of RSP yields

\((i)\) \( X \cdot b = X \).

So \( X \cdot b \overset{\text{REC}}{=} a \cdot X \cdot b \overset{(i)}{=} a \cdot X \cdot b \cdot b \), and hence \( E'(X \cdot b) \) is derivable. A second application of RSP yields \( X \cdot b = Y \). Combining this with \( (i) \) gives the desired result.

Finally, we introduce the Cluster Fair Abstraction Rule, by which it is possible to abstract from cycles of internal actions in a specification. This rule originates from [21].

\textbf{Definition 2.2.5.} Let \( E \) be a guarded recursive specification over some process algebra signature \( \Sigma \), and \( I \subseteq A \). A set \( C \subseteq \mathcal{W}_E \) is called a \textit{cluster} of \( I \) in \( E \) if the following condition is satisfied. For all \( X \in C \) there are \( t_1, \ldots, t_m \in I \cup \{ \tau \}, X_1, \ldots, X_m \in C \), and terms \( t_1, \ldots, t_n \) over \( \Sigma_{\text{REC}} \), with \( m \geq 1, n \geq 0 \), such that the equation for \( X \) is of the form

\[
X = \Sigma_{1 \leq k \leq m} i_k \cdot X_k + \Sigma_{1 \leq j \leq n} t_j.
\]

The terms \( t_j \) are called the \textit{exits} of the cluster. The cluster is called \textit{conservative} if every exit is accessible from every \( X \in C \) in the cluster by doing a finite number of steps from \( I \cup \{ \tau \} \).

\textbf{Definition 2.2.6.} The \textit{Cluster Fair Abstraction Rule} is the following rule:

\[
\frac{E \text{ is a guarded specification, } I \subseteq A}{X \in C, \text{ where } C \text{ is a finite conservative cluster of } I \text{ in } E \quad \text{U is the set of exits from } C}
\]

\[
\tau \cdot \tau_I(X) = \tau \cdot \Sigma_{t \in U} \tau_I(t)
\]

\text{CFAR}
2.3 Axioms for action alphabets

In the following sections we reason intensively on the action alphabets of processes. We take the defining axioms from [4, 8], listed in Table 2. The expression $\pi_n(p)$ denotes the projection of process $p$ up to depth $n$. The axioms AB and

| (AB1) | $\alpha(\nu) = \emptyset$ for $\nu \in \{\delta, \tau\}$ |
| (AB2) | $\alpha(a) = \{a\}$ |
| (AB3) | $\alpha(\tau \cdot x) = \alpha(x)$ |
| (AB4) | $\alpha(a \cdot x) = \{a\} \cup \alpha(x)$ |
| (AB5) | $\alpha(x + y) = \alpha(x) \cup \alpha(y)$ |
| (AB6) | $\alpha(x) = \bigcup_{n \geq 1} \alpha(\pi_n(x))$ |
| (AB7) | $\alpha(\tau(x)) = \alpha(x) \setminus I$ |

Table 2: The axioms AB for action alphabets, where $a \in A$ and $I \subseteq A$.

AB7 are often necessary for handling recursively specified processes.

Table 3 contains the conditional alphabet axioms. They can be derived for closed terms over $\Sigma(ACP^\star)$.

| (CA1) | $\alpha(x) \mid (\alpha(y) \cap H) \subseteq H \Rightarrow \partial_H(x \parallel y) = \partial_H(x \parallel \partial_H(y))$ |
| (CA2) | $\alpha(x) \mid (\alpha(y) \cap I) = \emptyset \Rightarrow \tau_I(x \parallel y) = \tau_I(x \parallel \tau_I(y))$ |
| (CA3) | $\alpha(x) \cap H = \emptyset \Rightarrow \partial_H(x) = x$ |
| (CA4) | $\alpha(x) \cap I = \emptyset \Rightarrow \tau_I(x) = x$ |

Table 3: The conditional alphabet axioms CA, where $H, I \subseteq A$.

3 Language matching

In this section we introduce operators for language matching, and study their characteristic algebraic properties.

3.1 Basic definitions for traces and languages

We first present some notations and informal definitions concerning traces. A sequence of atomic actions from some set $A$ is called a trace over $A$, or a word from $A^\star$. We moreover have $\lambda \in A^\ast$, where $\lambda$ is the empty trace.

Trace concatenation is defined as usual, i.e., if $\sigma, \rho \in A^\ast$ are traces and $a \in A$ an atomic action then $\sigma \rho, a \sigma$ and $\sigma a$ are also traces. Furthermore we use the convention that $a \lambda = \lambda a = a$.

**Definition 3.1.1.** Let $a \in A$ and $\sigma \in A^\ast$. The set acts of actions in a trace is defined by $\text{acts}(\lambda) \overset{\text{def}}{=} \emptyset$ and $\text{acts}(a\sigma) \overset{\text{def}}{=} \{a\} \cup \text{acts}(\sigma)$. □
A language over a set $A$ of atomic actions is a set of traces, or in other words: a subset of $A^*$.

**Definition 3.1.2.** Let $2^A$ and $2^{A^*}$ denote the power sets of $A$ and $A^*$, respectively. For $a \in A$, $\sigma \in A^*$, and $L$ a language over $A$ we extend the function $\text{acts}$ by $\text{acts} : 2^{A^*} \rightarrow 2^A$, and define $*: A^* \times 2^{A^*} \rightarrow 2^{A^*}$ and $\partial L/\partial a : 2^{A^*} \rightarrow 2^{A^*}$ as follows:

- $\text{acts}(L) \overset{\text{def}}{=} \bigcup_{\rho \in L} \text{acts}(\rho)$,
- $\sigma * L \overset{\text{def}}{=} \{ \sigma \rho \mid \rho \in L \}$,
- $\partial L/\partial a \overset{\text{def}}{=} \{ \rho \mid a \rho \in L \}$.

We write $\sigma L$ for $\sigma * L$.

Note that $\partial \{a\}/\partial a = \{ \lambda \}$ by definition.

In a setting with recursion infinite behaviour of processes can be modelled, so that we may need the possibility to reason about traces of arbitrary length. Therefore we introduce an operator for iteration on languages.

**Definition 3.1.3.** Let $L$ be a language over $A$. The closure or Kleene star of $L$, denoted by $L^*$, is inductively defined by

\[ \{ \lambda \} \in L^* \quad \text{and} \quad \text{if} \quad l \in L \quad \text{and} \quad p \in L^* \quad \text{then} \quad lp \in L^*. \]

From now on we use the symbol $Z$ for denoting an arbitrary set of traces over some set of actions $A$.

### 3.2 An operator for language matching

We use the operator $\Delta_Z$ for symbolising language matching with language $Z$. The notation $\Delta_Z$ is chosen in accordance with the notation for the operators $\nabla_Z$ for restriction from [1]. The axioms for language matching with trace label $r \in A$ are given in Table 4. A process trace is cut off and labelled with $r$ if it does not match a given language. So if an action is not 'expected', the subsequent term is replaced by $r$ (see axiom LM4). The action $r$ behaves in all respects as a 'normal' atomic action, and may occur anywhere in process terms. The operators $\Delta_Z$ are linear in the sense of [24].

Since $a = a \cdot r$, language matching applied to atomic actions can be treated with axiom LM4. We see for instance that $\Delta_{\{ab\}}(a) = \Delta_{\{ba\}}(a \cdot r) = a \cdot r$. Below, we give a detailed example of language matching.

**Example 3.2.1.** Let $a, b, c \in A$. If $p = a \cdot (b \cdot c + b) + b \cdot (a + c)$ and $Z = \{ab\}$ we have that

\[
\Delta_Z(p) = \begin{cases}
\Delta_{\{ab\}}(a \cdot (b \cdot c + b) + b \cdot (a + c)) \\
\Delta_{\{ab\}}(a \cdot (b \cdot c + b)) + \Delta_{\{ab\}}(b \cdot (a + c)) \\
a \cdot (\Delta_{\{b\}}(b \cdot c) + \Delta_{\{b\}}(b)) + b \cdot r \\
\Delta_{\{b\}}(b \cdot c) + \Delta_{\{b\}}(b) + b \cdot r \\
\Delta_{\{b\}}(b \cdot c) \cdot (c \cdot r) + \Delta_{\{b\}}(b \cdot r) + b \cdot r \\
a \cdot a \cdot (b \cdot c \cdot \Delta_{\{r\}}(c \cdot r) + \Delta_{\{b\}}(b \cdot r)) + b \cdot r \\
a \cdot (b \cdot c \cdot \Delta_{\{r\}}(c \cdot r) + b \cdot \Delta_{\{r\}}(c \cdot r)) + b \cdot r \\
a \cdot (b \cdot c + b) + b \cdot r.
\end{cases}
\]
Table 4: The axioms LM for language matching with trace label $r \in A$, where $a \in A$ and $Z \subseteq A^*$. 

So the above derivation shows that the summand $b \cdot (a + c)$ does not match the language \{ab\}, and how it is replaced by the labelled term $b \cdot r$. □

We refer to the axiom system $ACP^r$ extended with the axioms LM for language matching as $ACP^r_{LM}$, which of course also includes the definitions concerning traces. With $\Sigma(ACP^r_{LM})$ the signature of $ACP^r_{LM}$ is denoted.

In Table 5 transition rules for the $\Delta_Z$ operators are given in the format as advocated in [19].

Table 5: Transition rules for language matching, where $a, r \in A$ and $Z \subseteq A^*$.

We have the following standard results.

Identities between closed terms over $\Sigma(ACP^r_{LM})$ can often be proved by structural induction. The Elimination Theorem below implies that we then only have to consider basic terms (see Section 2.1).

**Theorem 3.2.2 (Elimination)**. If $p$ is a closed term over $\Sigma(ACP^r_{LM})$ then there is a closed term $q$ over $\Sigma(BPA^r_\Sigma)$ such that $ACP^r_{LM} \vdash p = q$. 

437

**Theorem 3.2.3 (Conservativity).** $ACP_{LM}^{T}$ is a conservative extension of $ACP^{T}$, i.e., for all closed terms $p, q$ over $\Sigma(ACP^{T})$ we have: $ACP_{LM}^{T} \vdash p = q$ iff $ACP^{T} \vdash p = q$.

**Proof.** The transition rules for language matching in Table 5 form an operationally conservative extension of the rules for $ACP^{T}$. According to a general result in [25], $ACP_{LM}^{T}$ is a conservative extension of $ACP^{T}$. □

**Lemma 3.2.4 (Congruence).** If $R$ is a rooted branching bisimulation on closed terms over $\Sigma(ACP_{LM}^{T})$ then $R$ is a congruence w.r.t. the operators of $ACP_{LM}^{T}$.

**Proof.** Standard (although not easy). □

**Theorem 3.2.5 (Soundness).** Let $p, q$ be closed terms over $\Sigma(ACP_{LM}^{T})$. If $ACP_{LM}^{T} \vdash p = q$ then $p \equiv_{+b} q$.

**Proof.** The relation $\equiv_{+b}$ between closed terms over $\Sigma(ACP_{LM}^{T})$ is a congruence, and therefore respects the inference rules for equality ($=$). So soundness only has to be shown for the axioms LM, which is straightforward. □

**Theorem 3.2.6 (Completeness).** Let $p, q$ be closed terms over $\Sigma(ACP_{LM}^{T})$. If $p \equiv_{+b} q$ then $ACP_{LM}^{T} \vdash p = q$.

**Proof.** The transition rules for language matching in Table 5 form an operationally conservative extension of the rules for $ACP^{T}$. Moreover we have the Elimination Theorem 3.2.2 and the completeness of $ACP^{T}$ w.r.t. $\equiv_{+b}$. According to a result in [25] we then also have completeness of $ACP_{LM}^{T}$. □

### 3.3 Some properties of language matching

Some special and useful properties of the operators $\Delta_{Z}$ for language matching are studied. We use the symbol $\circ$ for function composition.

**Lemma 3.3.1.** Let $V, Z \subseteq A^{*}$. For all closed terms $p$ over $\Sigma(ACP_{LM}^{T})$ we have:

1. $\Delta_{b}(p) = p$,
2. $\Delta_{Z} \circ \Delta_{Z}(p) = \Delta_{Z}(p)$,
3. $\Delta_{V} \circ \Delta_{Z}(p) = \Delta_{Z} \circ \Delta_{V}(p)$.

**Proof.** By induction on the structure of $p$. □

We go in a few steps to our main result.

**Lemma 3.3.2.** Let $Z \subseteq A^{*}$, $H \subseteq A$. For all closed terms $p$ over $\Sigma(ACP_{LM}^{T})$ we have:

if $r \not\in H$ and $r \not\in \alpha \circ \delta_{H} \circ \Delta_{Z}(p)$ then $\delta_{H} \circ \Delta_{Z}(p) = \delta_{H}(p)$.

**Proof.** By induction on the structure of $p$. □
Lemma 3.3.3. Let \( Z \subseteq A^* \), \( H \subseteq A \). For all closed terms \( p, q \) over \( \Sigma(ACP_{LM}) \) we have:

\[
\text{if } r \notin H \text{ and } r \notin \alpha \circ \partial_H(\Delta_Z(p) \parallel q) \text{ then } \partial_H(\Delta_Z(p) \parallel q) = \partial_H(p \parallel q).
\]

Proof. By simultaneous induction on the structure of \( p \) and \( q \). \( \square \)

Example 3.3.4. Let \( a, a', a^*, b, b', b^* \in A \), \( H = \{a, a', b, b'\} \), \( \gamma(a, a') = a^* \), and \( \gamma(b, b') = b^* \). If \( p = a \cdot (b \cdot c + b) + b \cdot (a + c), q = a' \cdot b', \) and \( Z = \{ab\} \) we have that

\[
\Delta_Z(p) = a \cdot (b \cdot c + b) + b \cdot r.
\]

It is easy to prove that \( \partial_H(\Delta_Z(p) \parallel q) = a^* \cdot (b^* \cdot c + b^*) \), so according to Lemma 3.3.3 we may conclude that \( \partial_H(\Delta_Z(p) \parallel q) = \partial_H(p \parallel q) \). We see that the summand \( b \cdot (a + c) \) of \( p \) is redundant in the context \( \partial_H(\_ \parallel q) \). \( \square \)

Below, operators for abstraction are incorporated.

Lemma 3.3.5. Let \( Z \subseteq A^*, I \subseteq A \). For all closed terms \( p \) over \( \Sigma(ACP_{LM}) \) we have:

\[
\text{if } r \notin I \text{ and } \text{acts}(Z) \cap I = \emptyset \text{ then } \tau_I \circ \Delta_Z(p) = \Delta_Z \circ \tau_I(p).
\]

Proof. By induction on the structure of \( p \). \( \square \)

Theorem 3.3.6 (Redundancy). Let \( Z \subseteq A^* \), \( H, I \subseteq A \). For all closed terms \( p, q \) over \( \Sigma(ACP_{LM}) \) we have:

\[
\text{if } r \notin H \cup I \text{ and } \text{acts}(Z) \cap I = \emptyset \text{ and } r \notin \alpha \circ \partial_H(\tau_I \circ \Delta_Z(p) \parallel q)
\]

\[
\text{then } \partial_H(\tau_I \circ \Delta_Z(p) \parallel q) = \partial_H(\tau_I(p) \parallel q) \quad (\text{RED})
\]

Proof. Easy, use the two preceding lemmas. \( \square \)

The first condition in the Redundancy Theorem can always be satisfied by choosing the trace label \( r \) properly. The third condition is crucial: if \( r \) does not occur in the action alphabet of a system with language matching applied to one of its components, we immediately have a specification of the total system without operators for language matching.

Note that Lemma 3.3.3 is an instance of the Redundancy Theorem (take \( I = \emptyset \)).

We study the interaction between language matching and the generalised alternative composition \( \Sigma \) more closely; the following lemma is on the distribution of language matching over summation, where the language to be matched contains actions with data parameters that also occur in \( \Sigma \). If a trace \( \sigma \) contains actions with data parameter \( e \) we also write \( \sigma(e) \) for \( \sigma \).

Lemma 3.3.7. Let \( D \) be a finite set of data, \( \{\sigma(e) \mid e \in D\} \subseteq A^* \). If \( a(d) \notin \text{acts}(\sigma(e)) \) for all \( e \neq d \), and \( \partial(\sigma(e) \mid e \in D)/\partial a(d) \neq \emptyset \) for all \( d \in D \), then:

\[
\Delta_{\{\sigma(e) \mid e \in D\}^*} \circ \Sigma_{d \in D} a(d) \cdot x = \Sigma_{d \in D} a(d) \cdot \Delta_{\sigma(d)(\{\sigma(e) \mid e \in D\})^*} / \partial a(d)(x).
\]

439
Proof. We have that \( a(d) \not\in \text{acts} (\sigma(e)) \), so \( \partial \sigma(e) \{ \sigma(f) \mid f \in D^* \} \partial a(d) = \emptyset \) for all \( e \neq d \). Since in general \( L^* = \bigcup_{\alpha \in L} \alpha L^* \cup \{ \lambda \} \), and \( \partial \{ \sigma(e) \mid e \in D \} \partial a(d) \neq \emptyset \) it must be that

\[
(i) \quad \partial \{ \sigma(e) \mid e \in D \}^* \partial a(d) = \bigcup_{e \in D} \partial \sigma(e) \{ \sigma(f) \mid f \in D \}^* \partial a(d)
= \partial \sigma(d) \{ \sigma(e) \mid e \in D \}^* \partial a(d).
\]

Now the proof is as follows:

\[
\Delta_{\{ \sigma(e) \mid e \in D \}} \circ \Sigma_{d \in D} a(d) \cdot x \overset{\text{LM2}}{=} \Sigma_{d \in D} \Delta_{\{ \sigma(e) \mid e \in D \}} \{ a(d) \cdot x \}
= \Sigma_{d \in D} a(d) \cdot \Delta_{\sigma(\tau(\tau^{-1}(a)d))}(x).
\]

Application of LM2 is allowed since \( D \) is finite. \( \square \)

Remark 3.3.8. Let the expression \( tr(p) \) denote the prefix closed set of traces of \( p \). For the \( tr \) operator see [1, 8], where traces over sets \( A \cup \{ \} \) are considered, modulo the equalities \( \sigma \tau = \tau \sigma = \sigma \) (so \( \tau \) corresponds here with the empty trace \( \lambda \)).

The relation between \( tr \) and the operators for language matching is maybe best illustrated by the following result. It can be proved that

\[
\text{if } \tau \not\in \alpha(p) \text{ then } tr \circ \tau(\tau^{-1}) \circ \Delta_{Z}(p) \subseteq tr(p).
\]

From the viewpoint of trace semantics, a process with language matching applied to it can execute only a part of its original behaviour. If the language to be matched becomes larger, the process behaviour generally becomes more restricted. \( \square \)

4 A verification of the Concurrent Alternating Bit Protocol

4.1 Description and specification of the CABP

4.1.1 Description

The Concurrent Alternating Bit Protocol can be graphically represented as in Figure 4.1, where the various communication ports and processes are depicted.

The following data are used in the specification of the CABP:

- \( D \) : arbitrary finite set of data elements.
- \( \text{Bit} = \{0, 1\} \) : set of bits used for internal control, and with an 'inverse' function \( \text{inv} : \text{Bit} \rightarrow \text{Bit} \).
- \( PS = \{1, 2, 3, 4, 5, 6, 7, 8\} \) : set of numbers of ports through which data are sent or read. The input port has number 1 and the output port has number 2.
**Figure 4.1:** Ports and processes of the CABP.

- $DS = D \cup (D \times \text{Bit}) \cup \text{Bit} \cup \{ce, ac, ae\}$: set of all transferable data in the protocol. The special constants $ce$, $ac$ and $ae$ are used to denote a checksum error introduced by channel $K$, an acknowledgement, and an acknowledgement error introduced by channel $L$, respectively.

The sending part is split into two components. One component, the *sender* $S$, does the reading of data elements $d \in D$ via the input port, and it sends a continuous stream of frames $(d \times \text{Bit})$ into channel $K$. A frame is a simple structure composed of a data element $d$ and a control bit $b \in \text{Bit}$. The other component, the *acknowledgement receiver* $AR$, handles incoming acknowledgements. If the acknowledgement receiver receives the expected control bit from channel $L$, an acknowledgement $ac$ is sent to the sender, the sender reads a new data element at port 1, and sends it in a frame with the control bit inverted into channel $K$. Again in a continuous stream. The receiving part is also split into two components. One component, the *receiver* $R$, receives frames from channel $K$ and takes care of the output. When the expected frame arrives, the data element is sent via the output, port 2, and an acknowledgement is sent to the other component, the *acknowledgement sender* $AS$. The acknowledgement sender fires a continuous stream of bits into channel $L$. As soon as it receives an acknowledgement from the receiver the bit is inverted. To make the protocol work correctly all components except the channels are initialised with a certain value of the control bit $b$.

The sending of continuous streams of data through the channels is a method to overcome the possible loss of data, because the channels are supposed to be ‘lossy’. A disadvantage for the verification is that these (independent) streams cause a modest ‘state space explosion’. In the case that the channels are replaced by queues of infinite capacity this even leads to an infinite number of states. This is the essential difference between the CABP and the ABP.
4.1.2 Specification

Below, the protocol is specified over the signature of ACP\(^*\) with recursion in a PSF-like fashion.

Atomic actions:

Let \( p \in PS \setminus \{1, 2\} \) and \( dat \in DS \). We define the following atomic actions:

- \( i, j \): internal actions of the channels
- \( r_p(dat) \): read \( dat \) at port \( p \)
- \( s_p(dat) \): send \( dat \) at port \( p \)
- \( c_p(dat) \): communicate \( dat \) at port \( p \)
- \( r_1(dat) \): read \( dat \) at input port
- \( s_2(dat) \): send \( dat \) at output port

Communications:

\[
s_p(dat) \mid r_p(dat) = c_p(dat) \quad \text{where} \quad p \in PS \setminus \{1,2\} \quad \text{and} \quad dat \in DS
\]

Sets:

\[
H = \{s_p(dat), r_p(dat) \mid p \in PS \setminus \{1,2\}, dat \in DS\}
\]

\[
I = \{c_p(dat) \mid p \in PS \setminus \{1,2\}, dat \in DS\} \cup \{i, j\}
\]

Processes:

\[
S = RM(0)
\]

\[
RM(b) = \sum_{d \in D} r_1(d) \cdot SF(d, b)
\]

\[
SF(d, b) = s_3(d, b) \cdot SF(d, b) + r_4(ac) \cdot RM(inv(b))
\]

\[
K = \sum_{d \in D, b \in BU} r_3(d, b) \cdot K(d, b)
\]

\[
K(d, b) = (i \cdot s_4(d, b) + i \cdot s_4(ac) + i) \cdot K
\]

\[
R = RF(0)
\]

\[
RF(b) = \sum_{d \in D} r_4(d, b) \cdot s_3(d) \cdot s_5(ac) \cdot RF(inv(b)) + (\sum_{d \in D} r_4(d, inv(b)) + r_4(ac)) \cdot RF(b)
\]

\[
AS = AS(1)
\]

\[
AS(b) = r_5(ac) \cdot AS(inv(b)) + s_6(b) \cdot AS(b)
\]

\[
L = \sum_{b \in BU} r_6(b) \cdot L(b)
\]

\[
L(b) = (j \cdot s_7(b) + j \cdot s_7(ac) + j) \cdot L
\]

\[
AR = AR(0)
\]

\[
AR(b) = (r_7(ac) + r_7(inv(b))) \cdot AR(b) + r_7(b) \cdot s_8(ac) \cdot AR(inv(b))
\]

\[
C_{abp} = \delta_H(S \parallel K \parallel R \parallel AS \parallel L \parallel AR)
\]

Externally, the protocol has to behave as a one element buffer. Therefore a verification of the \( C_{abp} \) is a proof of the following theorem.

**Theorem 4.1.1 (Requirement).** It holds that

\[
ACP_{LM} + SC + REC + RSP + CFAR^5 + AB + CA + RED^+  
\]
The $\tau$-actions indicate that the protocol can start with some sequence of internal actions.

The remainder of this paper is dedicated to a proof of the above theorem.

4.1.3 Modular specification

As announced in the introduction, the specification of the external behaviour of the CABP is split into two modules. We follow the approach of [14]. One module $(P)$ consists of the components $S$, $K$ and $R$, and a second module $(Q)$ consists of the components $AS$, $L$ and $AR$. The idea behind this division is quite simple. In the CABP there are two constant, independent streams of messages. Both cause a lot of internal activity. One stream, the stream of frames, flows from $S$ to $R$. This stream is contained in module $P$. The other stream, the stream of acknowledgements from $AS$ to $AR$, is contained in module $Q$.

Modules:

\[
H_P = \{rp(dat), sp(dat) \mid p \in \{3, 4\}, dat \in DS\} \\
I_P = \{cp(dat) \mid p \in \{3, 4\}, dat \in DS\} \cup \{i\} \\
P = \partial_{H_P}(S \parallel K \parallel R) \\
H_Q = \{rp(dat), sp(dat) \mid p \in \{6, 7\}, dat \in DS\} \\
I_Q = \{cp(dat) \mid p \in \{6, 7\}, dat \in DS\} \cup \{j\} \\
Q = \partial_{H_Q}(AS \parallel L \parallel AR)
\]

We can now give a modular specification of the external behaviour of the CABP. The sets $H$ and $I$ are defined in Section 4.1.2.

Lemma 4.1.2. A modular specification of the external behaviour of the CABP is

\[
\tau_I(Cabp) = \tau_I \circ \partial_H(\tau_{I_P}(P) \parallel \tau_{I_Q}(Q)).
\]

Proof. Straightforward; use the conditional alphabet axioms CA, see also [14].

By some elementary manipulation of the above equation we also find

\[
\tau \cdot \tau_I(Cabp) = \tau \cdot \tau_I \circ \partial_H(\tau_{I_P}(P) \parallel \tau \cdot \tau_{I_Q}(Q)).
\]

4.2 Some remarks on the verification

The specifications of the modules $P$ and $Q$ can be linearised by a simple head-tail expansion strategy to equational specifications $E_P$ and $E_Q$ respectively. The right hand sides of the equations of these specifications are then in head normal form. When the specifications $E_P$ and $E_Q$ are put in parallel and expanded, an equational specification for the CABP may be derived that consists of an enormous number of equations in head normal form. It can be shown that in total at least a few hundred equations have to be derived this way, for both possible values of the control bit $b$. 
Therefore, it is desirable to reduce the specifications $E_P$ and $E_Q$ as much as possible before they are interleaved. Of course we have the $\tau$-axioms B1 and B2 and the Cluster Fair Abstraction Rule at our disposal for abstracting from internal activity. Another significant reduction, which can already be obtained before abstraction is applied, is made with the technique of language matching.

Both the modules $P$ and $Q$ have some expected behaviour, e.g. certain sequences of atomic actions are not supposed to happen when the root processes $P$ and $Q$ are put in parallel. However, there is always the possibility that the modules themselves display redundant behaviour, which is only removed when they are put in parallel, and encapsulation is applied. This is why we use language matching. If language matching is applied, all possibly redundant terms in the specifications for $P$ and $Q$ are labelled and not expanded any further. The Redundancy Theorem roughly tells us that if no trace labels occur in the parallel composition $\partial_H(P_1 || Q_1)$, where $P_1$ and $Q_1$ are labelled versions of the processes $\tau_{LP}(P)$ and $\tau_{LQ}(Q)$, then $\partial_H(P_1 || Q_1) = \partial_H(\tau_{LP}(P) || \tau_{LQ}(Q))$. We stipulate that the trace label $r$ satisfies $r \not\in H \cup I$.

In the verification we use the following strategy:

1. A linear, guarded specification $E_P$ with root process $P_0(0)$ is given. This process is equal to a labelled version of the root process $P$ of module $P$ (Lemma 4.3.1).

2. Abstraction is applied, and with the axioms of $\text{ACP}^r$, together with CFAR$^b$, a reduced specification $E_{P_{red}}$ is obtained. $E_{P_{red}}$ has $X_0(0) = \tau_{LP}(P_0(0))$ as root process (Lemma 4.3.2).

3. As 1, for module $Q$ (Lemma 4.4.1).

4. As 2, for module $Q$. $E_{Q_{red}}$ has $Y_0(0)$ as root process. Since $Q_0(0)$ is in a cluster, we find the equality $\tau \cdot Y_0(0) = \tau \cdot \tau_{LQ}(Q_0(0))$ (Lemma 4.4.2), where the $\tau$'s are a direct consequence of CFAR$^b$.

5. A linear, guarded specification for $\tau_{LP} \partial_H(\tau_{LP}(P_0(0)) || \tau_{LQ}(Q_0(0)))$ can be obtained. This specification is equal to a given specification $E_{Cabp}$ with root process $C_0(0)$. With the help of the Redundancy Theorem 3.3.6 it is proved that $\tau \cdot C_0(0) = \tau \cdot \tau\text{I}(\text{Cabp})$ (Lemma 4.5.1).

6. Finally $E_{Cabp}$ is reduced, and it is proved that $\tau \cdot C_0(0)$, and thus $\tau \cdot \tau\text{I}(\text{Cabp})$, satisfies the requirement in Theorem 4.1.1.

4.3 Linearisation and reduction of module $P$

The specification of module $P$ can be rewritten to a corresponding, linear equational specification $E_P$ by applying a head-tail expansion strategy. Language matching is applied, since in the context $\partial_H(\_ || \tau_{LQ}(Q))$ certain atomic actions are not expected to happen after certain other actions: there is an expected infinite sequence $(a_1 \cdot a_2 \cdot s_2(d) \cdot c_3(ac) \cdot c_2(ac))\omega$ of visible actions. For instance, we do not expect $c_2(ac)$ to happen immediately before $s_2(d)$. Therefore, we label those summands in the specification for module $P$ that are in traces that do not match the language containing concatenations of traces

$$\sigma_P(d) \overset{\text{def}}{=} r_1(d) s_2(d) s_3(ac) r_3(ac),$$
where \( d \in D \), with \( r \). If our intuitions are correct then in the verification of the C A B P the guards in front of the processes \( r \) are encapsulated and removed.

Consider the following guarded specification \( E_P \).

\[
E_P = \{ P_0(b) = \sum_{d \in D} r_1(d) \cdot P_1(d, b), \\
P_1(d, b) = r_8(ac) \cdot r + c_3(d, b) \cdot P_2(d, b), \\
P_2(d, b) = r_8(ac) \cdot r + i \cdot P_1(d, b) + i \cdot P_3(d, b) + i \cdot P_4(d, b), \\
P_3(d, b) = r_8(ac) \cdot r + i \cdot P_4(d, b), \\
P_4(d, b) = r_8(ac) \cdot r + c_4(ce) \cdot P_5(d, b), \\
P_5(d, b) = r_8(ac) \cdot r + s_2(d) \cdot P_6(d, b) + c_3(d, b) \cdot P_7(d, b), \\
P_6(d, b) = r_8(ac) \cdot r + s_4(ac) \cdot P_8(d, b) + c_3(d, b) \cdot P_9(d, b), \\
P_7(d, b) = r_8(ac) \cdot r + s_2(d) \cdot P_8(d, b) + i \cdot P_9(d, b) + i \cdot P_{10}(d, b) + i \cdot P_{11}(d, b), \\
P_8(d, b) = r_8(ac) \cdot P_0(inv(b)) + c_3(d, b) \cdot P_{12}(d, b), \\
P_9(d, b) = r_8(ac) \cdot r + s_6(ac) \cdot P_{12}(d, b) + i \cdot P_8(d, b) + i \cdot P_{13}(d, b) + i \cdot P_{14}(d, b), \\
P_{10}(d, b) = r_8(ac) \cdot r + s_2(d) \cdot P_{13}(d, b), \\
P_{11}(d, b) = r_8(ac) \cdot r + s_2(d) \cdot P_{14}(d, b), \\
P_{12}(d, b) = r_8(ac) \cdot P_{15}(d, b) + i \cdot P_6(d, b) + i \cdot P_{16}(d, b) + i \cdot P_{17}(d, b), \\
P_{13}(d, b) = r_8(ac) \cdot r + s_6(ac) \cdot P_{16}(d, b), \\
P_{14}(d, b) = r_8(ac) \cdot r + s_6(ac) \cdot P_{17}(d, b), \\
P_{15}(d, b) = \sum_{d \in D} r_1(e) \cdot P_{18}(d, e, b) + i \cdot P_0(inv(b)) + i \cdot P_{19}(d, b) + i \cdot P_{20}(b), \\
P_{16}(d, b) = r_8(ac) \cdot P_{19}(d, b) + c_4(d, b) \cdot P_{20}(d, b), \\
P_{17}(d, b) = r_8(ac) \cdot P_{20}(b) + c_4(ce) \cdot P_{20}(d, b), \\
P_{18}(d, e, b) = r_8(ac) \cdot r + i \cdot P_{21}(d, e, b) + i \cdot P_{22}(e, b) + i \cdot P_1(e, inv(b)), \\
P_{19}(d, b) = \sum_{d \in D} r_1(e) \cdot P_{21}(d, e, b) + c_4(d, b) \cdot P_0(inv(b)), \\
P_{20}(b) = \sum_{d \in D} r_1(d) \cdot P_{22}(d, b) + c_4(ce) \cdot P_0(inv(b)), \\
P_{21}(d, e, b) = r_8(ac) \cdot r + c_4(d, b) \cdot P_1(e, inv(b)), \\
P_{22}(d, b) = r_8(ac) \cdot r + c_4(ce) \cdot P_1(d, inv(b)) \}
\]

**Lemma 4.3.1.** The root process \( P \) of the specification of module \( P \), matched with the language \( \{ \sigma_P(e) \mid e \in D \}^* \), and with trace label \( r \), is equal to the root process of \( E_P \):

\[
\Delta_{(\sigma_P(e) \in D)^*}(P) = P_0(0).
\]

**Proof.** Expand \( P \) with the axioms of ACP^* and apply language matching \( \Delta_{(\sigma_P(e) \in D)^*} \). The result follows easily using RSP; it has to be verified that the processes \( \Delta_{(\sigma_P(e) \in D)^*}(P) \) and \( P_0(0) \) satisfy the same system of recursion equations.

All equations in \( E_P \) hold for both values of \( b \), so the total labelled equational specification of \( P \) consists of at least 46 recursion equations, depending on the size of the set \( D \).

The next step in the verification is to abstract from all internal actions of module \( P \) by applying the operator \( T \). The exact strategy to follow is 1) to identify the conservative clusters of \( T \) in \( E_P \) (in \( E_P \) there are 4 clusters for each value of \( b \)), 2) to apply CFAR^*, and 3) to reduce the specification thus obtained to a manageable size, using the axioms of ACP^*. This part of the verification is quite paper consuming and not really instructive; therefore we omit it.
The strategy sketched above yields a number of recursion equations that can be related to a reduced equational specification \( E_{P, d} \) as follows: let

\[
E_p \equiv \{ \begin{align*}
X_0(b) &= \Sigma_{d \in D} r_1(d) \cdot X_1(d, b), \\
X_1(d, b) &= r_6(ac) \cdot r + s_2(d) \cdot X_2(d, b), \\
X_2(d, b) &= r_6(ac) \cdot r + s_5(ac) \cdot X_5(d, b), \\
X_3(d, b) &= r_6(ac) \cdot X_0(inv(b)) + r_8(ac) \cdot X_4(d, b), \\
X_4(d, b) &= \Sigma_{d \in D} r_1(d) \cdot X_5(d, b) + \tau \cdot X_0(inv(b)), \\
X_5(d, b) &= r_6(ac) \cdot \tau + r \cdot X_1(d, inv(b)) \}
\]

Lemma 4.3.2. We have that:

\[ X_0(b) = r_{1, s} (P_0(b)). \]

4.4 Linearisation and reduction of module Q

The specification of module Q can be rewritten to a linear equational specification \( EQ \) by applying a head-tail expansion strategy, just as we did with module P in the previous section.

Again we apply language matching, since in the context \( \partial_H (r_{1, s} (P) \parallel \_ ) \) certain atomic actions are not expected to happen after certain other actions.

There is of course again the expected infinite sequence \( (\Sigma_{d \in D} r_1(d) \cdot s_2(d) \cdot c_5(ac) \cdot c_8(ac))^\omega \) of visible actions. So we label those summands in the specification for module Q that are in traces that do not match the language containing concatenations of

\[ \sigma_Q \overset{\text{def}}{=} r_5(ac) s_8(ac) \]

with \( r \).

Consider the following guarded specification:

\[
E_Q = \{ \begin{align*}
Q_0(b) &= r_5(ac) \cdot Q_1(b) + c_6(inv(b)) \cdot Q_2(b), \\
Q_1(b) &= r_5(ac) \cdot r + c_6(b) \cdot Q_3(b), \\
Q_2(b) &= r_5(ac) \cdot Q_4(b) + j \cdot Q_0(b) + j \cdot Q_5(b) + j \cdot Q_6(b), \\
Q_3(b) &= r_5(ac) \cdot r + j \cdot Q_1(b) + j \cdot Q_7(b) + j \cdot Q_8(b), \\
Q_4(b) &= r_5(ac) \cdot r + j \cdot Q_1(b) + j \cdot Q_9(b) + j \cdot Q_6(b), \\
Q_5(b) &= r_5(ac) \cdot Q_9(b) + c_7(inv(b)) \cdot Q_0(b), \\
Q_6(b) &= r_5(ac) \cdot Q_8(b) + c_7(ac) \cdot Q_0(b), \\
Q_7(b) &= r_5(ac) \cdot r + c_7(b) \cdot Q_{10}(b), \\
Q_8(b) &= r_5(ac) \cdot r + c_7(ac) \cdot Q_1(b), \\
Q_9(b) &= r_5(ac) \cdot r + c_7(normal(b)) \cdot Q_1(b), \\
Q_{10}(b) &= r_5(ac) \cdot r + c_6(b) \cdot Q_{11}(b) + s_8(ac) \cdot Q_0(inv(b)), \\
Q_{11}(b) &= r_5(ac) \cdot r + j \cdot Q_{10}(b) + j \cdot Q_{12}(b) + j \cdot Q_{13}(b) + \\
& \quad s_8(ac) \cdot Q_2(inv(b)), \\
Q_{12}(b) &= r_5(ac) \cdot r + s_8(ac) \cdot Q_5(inv(b)), \\
Q_{13}(b) &= r_5(ac) \cdot r + s_8(ac) \cdot Q_6(inv(b)) \}
\]

Lemma 4.4.1. The root process \( Q \) of the specification of module Q, matched with the language \( \{ \sigma_Q \}^* \), and with trace label \( r \), is equal to the root process of

\[ E_Q : \Delta_{\{ \sigma_Q \}^*}(Q) = Q_0(0). \]
Proof. Expand $Q$ with the axioms of $ACP^r$ and apply language matching $\Delta_{\{q\}}$. The result follows easily using RSP. 

So the labelled equational specification of module $Q$ consists of 28 recursion equations. Because of the binary variable $b$, only 14 of them are necessary to specify module $Q$.

The next step is to apply the abstraction operator $\tau_{I_{0}}$ to $Q_{0}(b)$, and to reduce the above specification with the help of the $\tau$-axioms of $ACP^r$ and CFAR. (In $E_{Q}$ there are 3 conservative clusters for each value of $b$.)

Abstraction and reduction yield some recursion equations that can be related to a reduced equational specification $E_{Q_{red}}$ as follows: let

$$E_{Q_{red}} = \{ Y_{0}(b) = \tau_{0}(ac) \cdot Y_{1}(b), \quad Y_{1}(b) = \tau_{5}(ac) \cdot \tau + s_{8}(ac) \cdot Y_{0}(\text{inv}(b)) \}$$

Lemma 4.4.2. We have that:

$$\tau \cdot Y_{0}(b) = \tau \cdot \tau_{I_{0}}(Q_{0}(b)).$$

4.5 Verification of the CABP

The specifications of the modules $P$ and $Q$ are sufficiently reduced to compute an equational specification of the CABP. For clarity we do not yet rename the actions $c_{5}(ac)$ and $c_{8}(ac)$ from the abstraction set $I$.

Consider the following guarded specification:

$$E_{CABP} = \{ C_{0}(b) = \Sigma_{d \in D} \tau_{1}(d) \cdot C_{1}(d, b) + \tau \cdot C_{2}(b), \quad C_{1}(d, b) = s_{2}(d) \cdot C_{3}(d, b) + \tau \cdot C_{4}(d, b), \quad C_{2}(b) = \Sigma_{d \in D} \tau_{1}(d) \cdot C_{4}(d, b), \quad C_{3}(d, b) = \tau \cdot C_{5}(d, b), \quad C_{4}(d, b) = s_{2}(d) \cdot C_{5}(d, b), \quad C_{5}(d, b) = \tau_{I}(c_{5}(ac)) \cdot C_{6}(d, b), \quad C_{6}(d, b) = \tau_{I}(c_{8}(ac)) \cdot C_{2}(d, \text{inv}(b)) + \tau_{I}(c_{8}(ac)) \cdot C_{7}(d, b), \quad C_{7}(d, b) = \Sigma_{d \in D} \tau_{1}(d) \cdot C_{4}(d, b) + \tau \cdot C_{2}(d, \text{inv}(b)), \quad C_{8}(d, b) = \tau \cdot C_{4}(d, \text{inv}(b)) \}$$

Lemma 4.5.1. The root process of $E_{CABP}$ is related to the external behaviour of the CABP as follows:

$$\tau \cdot C_{0}(0) = \tau \cdot \tau_{I}(CABP).$$

Proof. Using head-tail expansion and the lemmas 4.3.2 and 4.4.2, it can be proved with the axioms of $ACP^r$, via RSP, that

$$C_{0}(0) = \tau_{I} \circ \partial_{H}(\tau_{I_{P}}(P_{0}(0)) \parallel \tau \cdot \tau_{I_{Q}}(Q_{0}(0))).$$

According to the lemmas 4.3.1 and 4.4.1 we then have

$$C_{0}(0) = \tau_{I} \circ \partial_{H}(\tau_{I_{P}} \circ \Delta_{\{c_{5}(c) \in D\}}(P) \parallel \tau \cdot \tau_{I_{Q}} \circ \Delta_{\{c_{8}(c)\}}(Q)).$$

It is easy to verify that all the clauses in the Redundancy Theorem 3.3.6 are satisfied, so

$$C_{0}(0) = \tau_{I} \circ \partial_{H}(\tau_{I_{P}}(P) \parallel \tau \cdot \tau_{I_{Q}}(Q)).$$

Finally, with Lemma 4.1.2 we find the desired result. \qed
The last part of the verification is the reduction of $EC_{abp}$, such that it follows that the CABP is a correct communication protocol.

**Lemma 4.5.2.** We have that:

$$\tau \cdot C_0(b) = \tau \cdot \sum_{d \in D} r_1(d) \cdot s_2(d) \cdot C_0(b).$$

**Proof.** Follows from the specification $EC_{abp}$ by some elementary calculations, using the axioms of $ACP^*$ and RSP.

Finally, using Lemma 4.5.1 it follows immediately from the above lemma that the CABP satisfies the correctness requirement in Theorem 4.1.1.

5 Concluding remarks

We presented a technique, language matching, for labelling process terms, based on trace information. We provided a modular verification of the Concurrent Alternating Bit Protocol, which showed that language matching can be particularly useful for suppressing the effects of redundancies in a context $H(\_ \parallel x)$. Earlier studies of this protocol had already demonstrated that redundancies are exactly what makes this protocol so hard to verify ([10, 14, 26]). Our verification is relatively short and straightforward. We think that it is reasonable to expect that language matching can be a good help for the modular, algebraic verification of other communication protocols as well.

A very different, but maybe equally interesting application domain of language matching may be simulation, or the symbolic execution of process specifications (see for instance [17]). This is because language matching makes it possible to automatically decide whether or not traces generated by a simulator are expected, i.e. in the language to be matched.
References


<table>
<thead>
<tr>
<th>Author</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baeten, J.C.M.</td>
<td>213</td>
</tr>
<tr>
<td>Basten, T.</td>
<td>181</td>
</tr>
<tr>
<td>Bergstra, J.A.</td>
<td>213, 309</td>
</tr>
<tr>
<td>Bezhen, M.A.</td>
<td>105, 121</td>
</tr>
<tr>
<td>Bol, R.N.</td>
<td>105</td>
</tr>
<tr>
<td>Brovedani, C.</td>
<td>157</td>
</tr>
<tr>
<td>Brunekreef, J.J.</td>
<td>413</td>
</tr>
<tr>
<td>D'Argenio, P.R.</td>
<td>67</td>
</tr>
<tr>
<td>Delft, A. van</td>
<td>229</td>
</tr>
<tr>
<td>Deursen, A. van</td>
<td>195</td>
</tr>
<tr>
<td>Feijs, L.G.M.</td>
<td>347, 387</td>
</tr>
<tr>
<td>Fokkink, W.J.</td>
<td>139</td>
</tr>
<tr>
<td>Fredlund, L.</td>
<td>285</td>
</tr>
<tr>
<td>Glabbeek, R.J. van</td>
<td>1</td>
</tr>
<tr>
<td>Groote, J.F.</td>
<td>105, 285</td>
</tr>
<tr>
<td>Hillebrand, J.A.</td>
<td>81</td>
</tr>
<tr>
<td>Huizer, M.</td>
<td>347</td>
</tr>
<tr>
<td>Klusener, A.S.</td>
<td>157</td>
</tr>
<tr>
<td>Korver, H.P.</td>
<td>81, 285</td>
</tr>
<tr>
<td>Mauw, S.</td>
<td>367</td>
</tr>
<tr>
<td>Middelburg, C.A.</td>
<td>309</td>
</tr>
<tr>
<td>Ponse, A.</td>
<td>121</td>
</tr>
<tr>
<td>Reniers, M.A.</td>
<td>367</td>
</tr>
<tr>
<td>Stolk, L.</td>
<td>229</td>
</tr>
<tr>
<td>Veltink, G.J.</td>
<td>63</td>
</tr>
<tr>
<td>Vereijken, J.J.</td>
<td>245</td>
</tr>
<tr>
<td>Voorhoeve, M.</td>
<td>181</td>
</tr>
<tr>
<td>Vrancken, J.L.M.</td>
<td>23</td>
</tr>
<tr>
<td>Wamel, J.J. van</td>
<td>429</td>
</tr>
<tr>
<td>Zantema, H.</td>
<td>139</td>
</tr>
</tbody>
</table>
Computing Science Reports

In this series appeared:

93/01  R. van Geldrop
93/02  T. Verhoef
93/03  T. Verhoef
93/04  E.H.L. Aarts
         J.H.M. Korst
         P.J. Zwieselting
93/05  J.C.M. Baeten
         C. Verhoef
93/06  J.P. Veldkamp
93/07  P.D. Moerland
93/08  J. Vervoorsel
93/09  K.M. van Hee
93/10  K.M. van Hee
93/11  K.M. van Hee
93/12  K.M. van Hee
93/13  K.M. van Hee
93/14  J.C.M. Baeten
         J.A. Bergstra
93/15  J.C.M. Baeten
         J.A. Bergstra
         R.N. Bol
93/16  H. Schepers
         J. Hooyman
93/17  D. Alstein
         P. van der Stok
93/18  C. Verhoef
93/19  G.J. Houben
93/20  F.S. de Boer
93/21  M. Codish
         D. Dams
         G. Filé
         M. Bruynooghe
93/22  E. Poll
93/23  E. de Kogel
93/24  E. Poll
         and Paula Severi
93/25  H. Schepers
         and R. Gerth
93/26  W.M.P. van der Aalst
93/27  T. Kloks
         and D. Kratsch
93/28  F. Kamareddine
         and
         R. Nederpelt
93/29  R. Post
         and P. De Bra
93/30  J. Deogun
         T. Kloks
         D. Kratsch
         H. Müller

Department of Mathematics and Computing Science
Eindhoven University of Technology

Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
A continuous version of the Prisoner's Dilemma, p. 17
Quicksort for linked lists, p. 8.
Deterministic and randomized local search, p. 78.
A congruence theorem for structured operational semantics with predicates, p. 18.
On the avoidability of metastable behaviour, p. 29
Exercises in Multiprogramming, p. 97
A formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
The Design of an Online Help Facility for ExSpect, p. 21.
Freeness Analysis for Logic Programs - And Correctness, p. 24
A Typechecker for Bijective Pure Type Systems, p. 28.
Relational Algebra and Equational Proofs, p. 23.
Pure Type Systems with Definitions, p. 38.
Multi-dimensional Petri nets, p. 25.
Finding all minimal separators of a graph, p. 11.
A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
GOLD, a Graph Oriented Language for Databases, p. 42.
On Vertex Ranking for Permutation and Other Graphs, p. 11.
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/31</td>
<td>W. Körver</td>
<td>Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.</td>
</tr>
<tr>
<td>93/33</td>
<td>L. Loyens and J. Moeren</td>
<td>ILLAS, a sequential language for parallel matrix computations, p. 20.</td>
</tr>
<tr>
<td>93/34</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Real Time Process Algebra with Infinitesimals, p. 39.</td>
</tr>
<tr>
<td>93/36</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Non Interleaving Process Algebra, p. 17.</td>
</tr>
<tr>
<td>93/37</td>
<td>J. Bruneel and J-P. Katoen</td>
<td>Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.</td>
</tr>
<tr>
<td>93/38</td>
<td>C. Verhoef</td>
<td>A general conservative extension theorem in process algebra, p. 17.</td>
</tr>
<tr>
<td>93/39</td>
<td>W.P.M. Nuijten</td>
<td>Job Shop Scheduling by Constraint Satisfaction, p. 22.</td>
</tr>
<tr>
<td>93/40</td>
<td>P.D.V. van der Stok</td>
<td>A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.</td>
</tr>
<tr>
<td>93/41</td>
<td>A. Bijlsma</td>
<td>Temporal operators viewed as predicate transformers, p. 11.</td>
</tr>
<tr>
<td>93/42</td>
<td>P.M.P. Rambags</td>
<td>Automatic Verification of Regular Protocols in P/T Nets, p. 23.</td>
</tr>
<tr>
<td>93/43</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata construction algorithms, p. 87.</td>
</tr>
<tr>
<td>93/44</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata minimization algorithms, p. 23.</td>
</tr>
<tr>
<td>93/45</td>
<td>E.J. Luit</td>
<td>A precise clock synchronization protocol, p. 11.</td>
</tr>
<tr>
<td>93/48</td>
<td>R. Genet</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
<tr>
<td>94/01</td>
<td>P. America</td>
<td>The object-oriented paradigm, p. 28.</td>
</tr>
<tr>
<td>94/02</td>
<td>F. Kamareddine</td>
<td>Canonical typing and $\Pi$-conversion, p. 51.</td>
</tr>
<tr>
<td>94/04</td>
<td>J.C.M. Baeten</td>
<td>Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.</td>
</tr>
<tr>
<td>94/06</td>
<td>T. Basten</td>
<td>Time and the Order of Abstract Events in Distributed Computations, p. 29.</td>
</tr>
<tr>
<td>94/08</td>
<td>O.S. van Roosmalen</td>
<td>A Hierarchical Diagrammatic Representation of Class Structure, p. 22.</td>
</tr>
<tr>
<td>94/09</td>
<td>J.C.M. Baeten</td>
<td>Process Algebra with Partial Choice, p. 16.</td>
</tr>
</tbody>
</table>
94/10 T. Verhoeff The testing Paradigm Applied to Network Structure, p. 31.
94/13 R. Seljée A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14 W. Peremans Ups and Downs of Type Theory, p. 9.
94/16 R.C. Backhouse and H. Doornbos Mathematical Induction Made Calculational, p. 36.
94/17 S. Mauw An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18 F. Kamareddine and R. Nederpelt Refining Reduction in the Lambda Calculus, p. 15.
94/19 B.W. Watson The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20 R. Bloo Beyond β-Reduction in Church’s λ→, p. 22.
94/22 B.W. Watson The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23 S. Mauw and M.A. Reniers An algebraic semantics of Message Sequence Charts, p. 43.
94/24 D. Dams Abstract Interpretation of Reactive Systems: Abstractions Preserving VCTL*, 3CTL* and CTL*, p. 28.
94/25 T. Kloks K₁,₂-free and W₄-free graphs, p. 10.
94/26 R.R. Hoogerwoord On the foundations of functional programming: a programmer’s point of view, p. 54.
94/29 J. Hooman Correctness of Real Time Systems by Construction, p. 22.
94/30 J.C.M. Baeten J.A. Bergstra Gh. Ştefănescu Process Algebra with Feedback, p. 22.
94/33 T. Laan A formalization of the Ramified Type Theory, p.40.
94/35 J.C.M. Baeten D. Mauw Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36 F. Kamareddine and R. Nederpelt Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/38 A. Bijlsma C.S. Scholten Point-free substitution, p. 10.
On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160

The $\lambda$-cube with classes of terms modulo conversion, p. 16.

On $\Pi$-conversion in Type Theory, p. 12.

Fixed-Point Calculus, p. 11.

Process Algebra with Propositional Signals, p. 25.

A short and flexible proof of Strong Normalisation for the Calculus of Constructions, p. 27.

Listing simplicial vertices and recognizing diamond-free graphs, p. 4.

Traces and Logic, p. 81

A Partial Order Approach to Branching Time Logic Model Checking, p. 20.

The Construction of a small Communication Library, p. 16.

Formalizing Process Algebraic Verifications in the Calculus of Constructions, p. 49.

Concrete process algebra, p. 134.

An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.

A Type Inference Algorithm for Pure Type Systems, p. 20.

A Quantitative Analysis of Iterated Local Search, p. 23.

Drawing Execution Graphs by Parsing, p. 10.

Preservation of Strong Normalisation for Explicit Substitution, p. 12.

Discrete Time Process Algebra, p. 20

Mathpad: A System for On-Line Preparation of Mathematical Documents, p. 15
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/11</td>
<td>R. Seljée</td>
<td>Deductive Database Systems and integrity constraint checking</td>
<td>36.</td>
</tr>
<tr>
<td>95/12</td>
<td>S. Mauw and M. Reniers</td>
<td>Empty Interworkings and Refinement Semantics of Interworkings Revised</td>
<td>19.</td>
</tr>
</tbody>
</table>