Discrete time process algebra: absolute time, relative time and parametric time

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Discrete Time Process Algebra: 
Absolute Time, Relative Time and Parametric Time

by

J.C.M. Baeten and J.A. Bergstra

97/04

ISSN 0926-4515

All rights reserved
editors: prof.dr. R.C. Backhouse
prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Reports 97/04
Eindhoven, February 1997
Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time

J.C.M. Baeten
Department of Computing Science, Eindhoven University of Technology,
P.O.Box 513, 5600 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

Abstract. We discuss the key notions of discrete time process algebra in the setting of ACP. Time is measured in discrete slices. The emphasis is on absolute, relative and parametric time notation.

Note: Partial support received from ESPRIT Basic Research Action 7166, CONCUR2. Published in Fundamenta Informaticae 29 (1/2), 1997, pp. 51-76.

1. Introduction

In this paper, we will explain the key notions of discrete time process algebra in the setting of ACP. ACP is the process algebra of [14, 11], other well known process algebras are CCS [35] and CSP [26]. In real time process algebra, all timing is measured on a continuous time scale; in discrete time process algebra, time is divided into a countable number of slices: the (n+1)st slice starts at time n and ends at time n+1. Within a slice, only the sequential ordering of actions matters and no further quantification of time is provided. Real time process algebra is a flexible specification tool and it also allows some verification. However, real time process algebra verifications are complicated in comparison to their time-free counterparts. Discrete time can be used if we aim at programming. We consider language modeling and design to be our primary objective for timed process algebras and that explains our focus on discrete time.

An important ramification is in absolute, relative and parametric time. In absolute time, all timing refers to a global clock; in relative time, timing is measured with respect to the current time. Parametric time integrates the absolute and relative timing constructs. Except [2, 3, 5, 8], the process algebras known to use are relative time ones. In [3], parametric time is introduced. A parametric time process depends on its moment of initialisation in a possible nontrivial way. ATP [38] is a relative discrete time process algebra, SDL [41] uses absolute
discrete time and the discrete time TOOLBUS [15, 16] uses parametric discrete time. Clearly all systems can be specified using absolute time notation. But the use of relative time notation may drastically simplify notation. Parametric time becomes necessary for instance if a parallel composition is used of two processes, one specified in absolute time, the other in relative time. In [5], we have an example of a processor put in parallel with keyboard and printer. The processor contains a clock and therefore must be specified in absolute time. The printer and keyboard can be specified in relative time. The processor can be switched on and off from the keyboard. The whole system can be nicely specified in parametric time.

A further ramification occurs when we axiomatise discrete time process algebra: timestamped syntax versus two phase syntax. In timestamped syntax, the actions are labeled with the number of the time slice (in the case of absolute time) or the number of time slices visited since the last action (in the case of relative time). In two phase syntax actions are separated from timing information: we have constants denoting actions in the first time slice and a delay operator shifting a process to a later time slice (absolute time) or actions in the current time slice and a delay operator (relative time).

1.1 Relations to existing literature

We mention four sources of our approach: ACP [14] which it extends, ACPp [2, 29, 21], the absolute time real time version of ACP and its parametric time extension of [5], in which all discrete time constructs can be embedded. The fourth source is ATP [38]. ATP is a relative time process algebra. Our relative time process algebra is more or less the same be it that we use time stamped notation here. The main design decision taken from ATP was to adopt time factorisation, this means that progress of time by itself does not generate nondeterminism. A more formal explanation of discrete time process algebra can be found in [8].

In relative discrete time process algebra, we combine a unary operator for (unit) delay like in TCCS [36], time factorisation as in ATP [38], TCCS [36], TPCCS [24] and TPL [25], the interpretation of + as the weak choice of TCCS, but we notice a significant difference with TPL [25] because timed actions cannot idle (here, we follow TCCS, ACP and ATP).

Our definition of free merge in relative discrete time is similar to the definitions in TCCS, ATP, ACP and TPL (the correspondence with the TPL definition holds in the absence of communication; in the presence of communication TPL's merge will give priority to internal communications). So it follows that without communication the relation between merge and discrete time is not controversial. We depart from ATP by allowing time stops.

Parallel composition with communication in relative discrete time works just as the merge of TCCS [36] (taking weak choice of TCCS for +). It is also equivalent to the merge of ATP. It differs from merge in [22], set up in ACP. In fact, ACP contains axioms \( \sigma \parallel x = \delta \) and \( \sigma \parallel y = \delta \). Our main objection against these axioms (which occur in ATP as well) is that
they render it impossible to injectively embed $\text{ACP}_t$ into $\text{ACP}_p$ of [2] or its extension $\text{ACP}_p^\perp$ of [5].

Below, we will describe various forms of discrete time process algebra in more detail. Some axiom systems will be presented. We refer to [3, 8] for further information. Our description is based on a fixed semantic model. In this model timed processes are viewed as strong bisimulation equivalence classes of timed transition systems.

1.2 Objectives of discrete time process algebra

The role of discrete time process algebra concerns first of all conceptual analysis. Using a limited collection of constructors, process domains $P_{\text{abs, rel, par}}$ for absolute, relative and parametric time can be determined. The precise description of the interaction between absolute time and relative time notation is a primary goal. It has been recognized on various occasions that relative time notation only will not meet all practical needs, whereas a restriction to absolute time notation is often cumbersome.

A second objective of process algebra in general and discrete time process algebra in particular is to allow the development of a family of operators on top of the given set of constructors. These additional operators enhance the expressiveness of the algebraic notation in various directions. There is some debate about the usefulness of new operators. Indeed, many systems can be defined in a rather Spartan way, using a limited set of primitives. For practical applications, however, a tailored collection of primitives defined on top of the given framework (be it absolute, relative or parametric time) is often more promising.

Bisimulation oriented process algebra is ideally suited for the development of tailored operators aiming at specific applications. To convince the reader of the need of a large family of operators, we mention some extensions that have been introduced for time free process algebra in recent years. There is [20], who add preferential alternative composition (first try this, then try that) in order to model the protocol testing language TTCN [28]. Further, we mention the Discrete Time ToolBus of [16] which makes use of tailored notation for time bounds on actions as well as of process creation [12]. In [17], the description of a fragment $\phi\text{SDL}$ of SDL [41] uses the state operator of [1]. This operator is also used in the semantic model of language POOL in [42] and the model of MSC's in [30]. The interworking merge in [32] allows a concise modeling of interworkings, a system notation developed by researchers in Philips Kommunikations-Industrie in Nürnberg. [13] provides a process algebra version of Kleene's iteration construct. This is used in the ToolBus mentioned above as well as in the model of $\phi\text{SDL}$. [6] contains primitives for parallel inputs.

We notice that due to arbitrary interleaving the set of process constructors can be kept small. On the other hand, every new feature (operator) requires additional equations to ensure that arbitrary interleaving is reflected by expansion of parallel composition. Most operators in the present paper are standard, except perhaps for the conditional construct which (surprisingly) serves as a constructor in the parametric time case.
In our presentation, we will start by sketching a number of semantic domains corresponding to $P_{\text{abs}}$, $P_{\text{rel}}$, $P_{\text{par}}$. Using these semantic domains one may formulate soundness and completeness issues for appropriate axiom systems. In each case, the processes emerge as bisimulation equivalence classes of transition systems. We pay attention to be quite precise in the discussion of transition systems. This is done because it is not helpful if the use of process algebra notation confuses the understanding of the underlying world of automata and transition systems.

We notice that following [18] the domain of transition systems itself can be given an algebraic structure. This leads to so-called frame algebras, where frames are transition systems without designated root states and termination states. Frames provide a lower level of abstraction. The price to pay is that additional features (operators) introduce difficulties. Parallel composition, for instance, turns out to be neither commutative nor associative. With this remark we may illustrate the "experimental fact" in process algebra that bisimulation is an adequate level of abstraction. If one has a far less abstract setting, e.g. transition systems modulo isomorphism, the development of operators becomes harder. But equally, in a far more abstract setting like trace theory, the introduction of new operators may be more problematic. To give an example, the priority operator of [9] is a very useful operator that defeats definition in a trace theory based setting (see [10]).

2. Discrete time transition systems

We define a set of process graphs as in [11] with the transitions either labeled with atomic actions $a$ from a given finite set $A$, or with the time progression symbol $\sigma$. Time factorisation demands that every node has at most one outgoing $\sigma$-labeled edge. A node with no outgoing transitions is called a termination node. We distinguish three kinds of termination nodes: successful termination nodes (labeled $\checkmark$), immediate deadlock nodes (labeled $10$), and other termination nodes. The last kind denotes a deadlock at the end of the current time slice: further activity of other parallel components in the current time slice is possible, but passage to the next time slice is not allowed. A $\sigma$-edge may not lead to a $\checkmark$-node.

We consider transition systems using relative timing, absolute timing and parametric timing in turn. An example of a transition system is given in fig. 1. We notice that transition systems will serve as our basic semantic objects. Equivalence classes of transition systems will serve as the denotation of processes. In this way, one obtains the mathematical background for the equational calculus. In the relative timing case, a transition system inherits all timing during operation from the time of startup. So if in slice 17 the transition system is put in operation, this enacts that (dynamically) the root of the transition system $T$ is equipped with time stamp 16. In the case of figure 1, the successful termination node gets time stamp 18, the immediate deadlock node time stamp 17. The actual state space of the transition
system during operation is a subset of pairs of nodes of \( T \) and natural numbers, a subset of \( N(T) \times N \), \( N \) the node set of \( T \), and \( N \) represents the set of time stamps.

2.1 Relative timing

A rooted two-phase transition system in relative time over a set of action labels \( A \), a 2pTSr(\( A \)) is a process graph as indicated above. To state this precisely, a 2pTSr(\( A \)) is a sixtuple \((N, E, r, T, ID, \nabla)\) where \( N \) is the set of nodes, \( E \subset (N-T) \times A \cup \{\sigma\} \times N \) is the set of edges, \( r \in N \) is the root node, \( T \subset N \) is the set of termination nodes, and \( ID \subset T, \nabla \subset T-(ID \cup \{r\}) \), and with the condition:

- if \( s \xrightarrow{a} t \) and \( s \xrightarrow{\sigma} t' \), then \( t = t' \) and \( t \notin \nabla \).

We have the following intuitions:

1. An edge \( s \xrightarrow{a} t \) denotes an action execution in the current time slice: if a system is in state \( s \), execution of \( a \) within the time slice is possible, resulting in state \( t \).
2. An edge \( s \xrightarrow{\sigma} t \) denotes a time step, passage to the next time slice.
3. The root node denotes the starting point of the transition system.
4. A node labeled \( \nabla \) denotes successful termination.
5. A node labeled \( ID \) denotes immediate deadlock: no further activity is possible, no action step, no time step, no successful termination. Thus, this label indicates a time stop.
6. Termination nodes not labeled with either \( \nabla \) or \( ID \) allow further waiting within the time slice, but allow no time step.

We call a node that can be reached from the root by just following \( \sigma \)-edges an initial node. In particular, the root itself is always initial. The system in fig. 1 has three initial nodes.

2.2 Absolute timing

Next, in the case of absolute timing, all nodes of a transition system \( T \) are now equipped with a time stamp. \( ts \) is a mapping from the node set \( N \) to the timestamp collection \( N \) that determines the slice of a state. If \( ts(s) = n \) then the system modeled by \( T \) can reside in state \( s \) only during time slice \( n+1 \). We notice that \( ts \) of the root fixes the time of initialisation of a transition system in absolute time.
A rooted two-phase transition system in absolute time starting at time $k$ over a set of action labels $A$, a $2pTSa(A,k)$ is a septuple $(N, E, r, T, ID, \triangleright, ts)$, where $(N, E, r, T, ID, \triangleright)$ is a $2pTSr(A)$ and $ts$ is the so-called (absolute) time slice function, that assigns to each node a natural number, indicating the beginning of the current time slice, with the following properties:

1. $ts(r) = k$
2. if $s \xrightarrow{a} t$ then $ts(s) = ts(t)$
3. if $s \xrightarrow{\sigma} t$ then $ts(s) + 1 = ts(t)$.

Note that for each $n$, there can be at most one initial node with $ts$ equal to $n$, as nodes can have at most one outgoing $\sigma$-edge. Not every $2pTSr$ can be turned into a $2pTSa$ with the same set of nodes, as a $\sigma$-edge in a $2pTSa$ can never lie on a cycle. However, we do have the following constructions.

### 2.3 Initialisation operators on timed transition systems

For $T$ a relative time transition system and $k$ a time stamp, $k \triangleright T$ will denote the absolute time transition system that results when $T$ is initialised at time $k$. This will in fact be the key operational mechanism for relative time transition systems. Actual execution begins in some slice say $k+1$. That leads to an initial label $k$ which is assigned to the root of $T$. Progressively, during execution, further states obtain a $ts$ component. Formally, this works as follows.

If $T = (N, E, r, T, ID, \triangleright)$ is a $2pTSr(A)$, and $k$ is a natural number, then we can form $k \triangleright T$ in $2pTSa(A,k)$ as follows: the set of nodes is $N \times N$, $ts((s,n)) = n$, the root is $(r,k)$, $(s,n) \xrightarrow{a} (t,n)$ if $s \xrightarrow{a} t$ in $T$, $(s,n) \xrightarrow{\sigma} (t,n+1)$ if $s \xrightarrow{\sigma} t$ in $T$, and $\triangleright$ or $ID$ holds when it holds in $T$ for the first component.

$k \triangleright T$ is $T$ initialised at $k$, started up at time $k$. We write $IT_{\text{abs}}$ for $0 \triangleright T$. We can also go in the reverse direction, from a $2pTSa(A,k)$ to a $2pTSr(A)$, by just forgetting the time slice function. We denote the resulting relative time system by $IT_{\text{rel}}$. Notice that for a $2pTSa(A,k)$ $T$, we have $k \triangleright IT_{\text{rel}} = T$. We can also define initialisation on absolute time transition systems, $k \triangleright T$ denotes that part of $T$ that starts at $k$. If $T$ is a $2pTSa(A,k)$, and $n \in N$, then the $2pTSa(A,n) n \triangleright T$ is defined in two cases:

- if $n \geq k$, and there is an initial node $s$ with $ts(s) = n$, take that part of $T$ that has $s$ as root, and add as nodes all nodes that can be reached from $s$. If such $s$ does not exist, the graph of $T$ consists of a single node with $ts n$ and label $ID$.
- if $n < k$, take a new root $r$ with $ts(r) = n$, and add a series of $k-n \sigma$-edges from $r$ to the old root.
2.4 Parametric timing

The dependence of a process (or rather, a transition system), on its time of initialisation is modest, both in absolute time and in relative time. In the parametric time case, we intend to allow a completely flexible dependence between initialisation of a system and subsequent behavior. This is reflected in the notion of a parametric time transition system, which is a mapping from \( \mathbb{N} \) (the range of time stamps, viewed as initialisation time) to transition systems.

A rooted two-phase transition system in parametric absolute time over a set of labels \( A \), a \( 2p\text{TS}_\text{pa}(A) \) is a family of \( 2p\text{TS}_\text{sa}(A,k) \), one for each natural number \( k \). A rooted two-phase transition system in parametric relative time over a set of labels \( A \), a \( 2p\text{TS}_\text{pr}(A) \) is a family of \( 2p\text{TS}_\text{sr}(A) \) indexed by natural numbers. We can consider each \( 2p\text{TS}_\text{sr} \) as a \( 2p\text{TS}_\text{pr} \) by just taking the constant sequence; conversely, we can consider a \( 2p\text{TS}_\text{sr} T \) as a \( 2p\text{TS}_\text{pa} \) by taking the sequence \( (n \gg T)_n \in \mathbb{N} \). This sequence denotes the behaviour of the system for each initialisation time.

Initialisation of a transition system in parametric time \( T = (T_n)_n \in \mathbb{N} \) works as expected:
- \( n \gg T = T_n \) in the absolute time case, and
- \( n \gg T = n \gg T_n \) in the relative time case.

In absolute time, the root of \( T_n \) has time stamp \( n \), so \( T_n = n \gg T_n \). In general, we may say \( n \gg T = n \gg T_n \). The virtue of this is that \( T_n \), being just one component of \( T \), is simpler, it is either absolute time or relative time, whereas \( T \) itself is a parametrised family of transition systems.

2.5 Bisimulation

Bisimulation is a crucial equivalence concept for transition systems due to Park [39] for the time free case. Bisimulation turns out to be a very flexible concept in the sense that it can be tailored to many different kinds of transition systems. We will define bisimulation in the relative and absolute time cases whereas for parametric time we can use a reduction to the absolute time case by working pointwise.

Intuitively, two transition systems \( T \) and \( T' \) are bisimilar if there exists a relation \( R \) between nodes of \( T \) and nodes of \( T' \) which connects every (reachable) state of \( T \) to one or more reachable states of \( T' \), and conversely. Further, this relation has to realise a simulation of \( T \) by \( T' \) and at the same time a simulation of \( T' \) by \( T \). Technically, this means that the roots are related and if of a pair of related nodes one of them has an outgoing edge, the other also has an outgoing edge with the same label, such that the resulting pair is again related. Stated mathematically, the last requirement says that if \( R(s, s') \) and \( s \xrightarrow{a} t \) in \( T \), then this step can be simulated in \( T' \), i.e., there exists \( t' \) in \( T' \) such that \( s' \xrightarrow{a} t' \) and \( R(t, t') \) and conversely, if \( R(s, s') \) and \( s' \xrightarrow{a} t' \) in \( T' \), then this step can be simulated in \( T \), there exists \( t \) in \( T \) such that \( s \xrightarrow{a} t \) and \( R(t, t') \).

. \( \mathcal{F} \)
In the case of timed transition systems, we need some minor adaptations:

i. In all cases, if \( R \) is a bisimulation between \( T \) and \( T' \) then the two-way simulation also holds for time steps, and whenever two nodes are related, they always have the same label (so a \( \sqrt{\cdot} \)-node is only related to a \( \sqrt{\cdot} \)-node, an \( \{D\} \)-node to an \( \{D\} \)-node, a root node to a root node).

ii. In the absolute time case, there is the further requirement that the time slice of related nodes always coincides, so \( R(s, s') \) implies \( ts(s) = ts(s') \) (notice that it suffices to have this requirement for the roots).

We have the usual result that bisimulation is an equivalence relation on transition systems.

A relative time discrete time process over \( A \) is a bisimulation equivalence class of a \( 2pTSr(A) \). Similarly, an absolute time discrete time process is a bisimulation equivalence class of \( 2pTSA(A,k) \), where \( k \) is the time of initialisation. If we do not specify \( k \), we have \( k=0 \). Likewise, a parametric time process is a family of bisimulation equivalence classes of \( 2pTSA(A,k) \), one for each natural number \( k \), or a family of bisimulation equivalence classes of \( 2pTSr(A) \) indexed by natural numbers.

2.6 Time step operators

We define two time step operators on transition systems:

- \( \sigma_{abs} : 2pTSA(A,k) \rightarrow 2pTSA(A, k+1) \) is defined by increasing all time stamps by 1.
- \( \sigma_{rel} : 2pTSr(A) \rightarrow 2pTSr(A) \) is defined by putting a \( \sigma \)-step in front of the root. We can also define this operator on \( 2pTSPA(A) \), as the \( n \)th component is obtained by applying \( n \gg \) to the \((n+1)\)st component of the old system: \( n \gg \sigma_{rel}(T) = n+1 \gg T \).

The importance of the time step operators is that they may serve as process constructors in a process algebra.

3. Basic syntax and equations

3.1 Processes as bisimulation equivalence classes

We will use \( 2pTSpr \) transition systems modulo bisimulation equivalence as a universal domain. If necessary, we use conversion functions from absolute to relative time processes or vice versa. Further, we can use \( 2pTSPA \) objects, if we apply an appropriate conversion. So a process in the universal domain is a bisimulation equivalence class of a family of transition systems in \( 2pTSr(A) \). Further, we will restrict attention to countable transition systems, and consequently, our process domain has the cardinality of the continuum.

We write \( P_{abs} \) for the domain of \( 2pTSA(A,0) \) transition systems modulo bisimulation, the domain of absolute time processes, \( P_{rel} \) for the domain of \( 2pTSr(A) \) transition systems modulo bisimulation, the domain of relative time processes, \( P_{par} \) for the domain of
2pTSpr(A) transition systems modulo bisimulation, the domain of parametric time processes. Operators that we have introduced can all be extended to these equivalence classes in the canonical way, i.e. by applying them to a representative. We see that \( P_{\text{abs}} \) is the subalgebra of all \( P_{\text{par}} \) processes \( P \) satisfying \( P = |P|_{\text{abs}} \), and similarly \( P_{\text{rel}} \) is the subalgebra of all \( P_{\text{par}} \) processes \( P \) satisfying \( P = |P|_{\text{rel}} \).

A relative time process \( P \) is regular (or finite state) if there is a finite 2pTSr in its equivalence class. An absolute time process \( P \) is regular if \( |P|_{\text{rel}} \) is regular. For a parametric time process, we say that it is regular if all its components are regular and the sequence of components is eventually periodic.

### 3.2 Process algebra

Having available an appropriate domain of processes, we will now introduce a number of process algebras. Process algebras are subclasses of the universal domain, equipped with a number of operators. Process algebras show variation in the size of their domain and the number and kind of operators. If the domain of a process algebra is a subclass of \( P_{\text{abs}} \) then we will call this an absolute time process algebra, similarly for relative time and parametric time process algebras.

The syntax of process algebra will provide compositional means to denote processes, starting with the simplest building blocks: the timed atomic actions and the timestop. We will follow the stepwise approach of ACP. The basic system BPA (for basic process algebra) allows to generate processes from atomic actions by means of the two essential primitives: alternative composition, written \( X + Y \), and sequential composition written \( X \cdot Y \). The process \( X \cdot Y \) will start the execution of \( Y \) upon the successful termination of \( X \). The alternative composition \( X + Y \) is a process in a state from which it can proceed either as \( X \) or as \( Y \) (we notice that the + operator was introduced by Milner in CCS [34]). There is no explicit indication of a mechanism that enacts the choice between \( X \) and \( Y \). The + admits a flexible interpretation, ranging from \( X + Y \) chooses non-deterministically and autonomously between \( X \) and \( Y \), to \( X + Y \) allows its environment to determine a choice between \( X \) and \( Y \).

We enrich BPA with \( \delta \), the immediate deadlock, thus obtaining BPA\( ^\delta \). Immediate deadlock is a process that cannot occur in nature (or rather: computing practice), but it is useful for an efficient axiomatic treatment.

The next phase (in section 4) is to introduce parallel composition without communication (free merge). This leads to a system called PA. Parallel composition is denoted \( \parallel \) (merge). An auxiliary operator \( \parallel \) (left merge) is used to obtain an efficient algebra. The next step is to introduce communication. This involves the introduction of a communication function on atoms and an auxiliary operator communication merge, and an encapsulation operator. This leads to the level of full ACP from [14] equipped with discrete time. The ACP primitives allow to describe communication protocols such as sliding window protocols, the Ethernet and the token ring, see [31]. The papers in [31] use ACP augmented with algebraically
specified abstract data types. Using the discretely timed versions of ACP one may specify discretely timed formalisations of these protocols.

3.3 Atomic actions

In this section, a notation for atomic actions is introduced, that will serve as the basis for the process algebras to be developed. We consider two versions, two phase notation and timestamped notation.

The two phase notation is useful for theoretical purposes, in particular it is helpful for finding precise definitions and concise axiomatisations. The timestamped notation is preferable if readability of specifications matters.

3.4 Atomic actions in two phase notation

Let \( A \) be a given set of labels. For \( a \in A \), \( \text{fts}(a) \) (\( a \) in the first time slice) denotes the process in \( P_{\text{abs}} \) represented by the graph with two nodes, the first the root, the second a \( \sqrt{\cdot} \)-node, connected by an \( a \)-edge, and the time slice is 0 for both nodes. For \( a \in A \), \( \text{cts}(a) \) (\( a \) in the current time slice) denotes the process in \( P_{\text{rel}} \) defined as \( \text{fts}(a)_{\text{rel}} \) (i.e. just omit the time slice function).

3.5 Atomic actions in timestamped notation

The advantage of two phase notation is visible in theory: many nice equations can be found such as \( \sigma_{\text{rel}}(X) + \sigma_{\text{rel}}(Y) = \sigma_{\text{rel}}(X + Y) \). In [8] we provide finite axiomatisations in the style of ACP for finite processes in all three cases: absolute time, relative time and parametric time. The main disadvantage of two phase notation is that it renders complex expressions quite unreadable. This is the reason we have developed a timestamped notation for discrete time process algebra as well. There are several notations, useful in different circumstances. We follow [3] and use the following notation: \( \mathbf{a}(n) \) is in absolute timing, denoting \( a \) in the \( n \)th time slice, \( \mathbf{a}[n] \) is in relative timing, denoting \( a \) in the \( n \)th time slice from system startup. The interpretation of \( \mathbf{a}(n+1) \) as an object in \( 2pTSa(A,n) \) is the graph with two nodes, both with \( ts n \), an \( a \)-edge from the first node to the second, the first the root, the second with label \( \sqrt{\cdot} \). In order to have the interpretation as a \( 2pTSa(A,k) \), use the \( k \geq \) initialisation function on this graph. The interpretation of \( \mathbf{a}[n+1] \) as a \( 2pTSr(A) \) is a graph with \( n+1 \) nodes, the first \( n \) nodes connected by \( \sigma \)-edges, the last pair by an \( a \)-edge, the first node the root, the last node labeled \( \sqrt{\cdot} \). We have the following relations between two phase and timestamped notations.
absolute timing:  
\[ a(0) = \delta \]
\[ a(1) = \text{fts}(a) \]
\[ a(n+2) = \sigma_{\text{abs}}^{n+1}(\text{fts}(a)) \]
\[ \delta(0) = \delta \]
\[ \delta(n+1) = \sigma_{\text{abs}}^{n+1}(\delta) \]

relative timing:  
\[ a[0] = \delta \]
\[ a[1] = \text{cts}(a) \]
\[ a[n+1] = \sigma_{\text{rel}}^{n+1}(\text{cts}(a)) \]
\[ \delta[0] = \delta \]
\[ \delta[n+1] = \sigma_{\text{rel}}^{n+1}(\delta) \]

Parametric timing does not use any new notation for atomic actions.

We will provide an axiomatic treatment of discrete time process algebra using the notations \( a(n) \) and \( a[n] \). In a timestamped presentation \( \sigma_{\text{abs}} \) and \( \sigma_{\text{rel}} \) are not needed as constructors.

### 3.6 Sequential composition and alternative composition (BPA₈)

As a basis for an axiomatic theory, common to absolute, relative and parametric time versions, we start out from the time free theory BPA with the constant \( \delta \). This process will play a special role, it stands for immediate (and catastrophic) deadlock. It was introduced in [8]. We have operators + (alternative composition, choice) and \( \cdot \) (sequential composition). \( X + Y \) will execute either \( X \) or \( Y \), but not both; \( X \cdot Y \) will start the execution of \( Y \) upon successful termination of \( X \). These elements constitute the syntax of BPA₈. The theory BPA₈ of [8] has the axioms in table 1 below. The axioms A1-3 make that the alternatives form a set at each point; we have the right distributivity axiom A4, but not the left distributivity, as the moment of choice is important; A5 is the associativity of sequential composition. A6ID denotes that no immediate deadlock can occur as long as there is an alternative way to proceed; A7ID indicates that no further activity can occur if we are in a deadlock situation.

| A1 | \( X + \delta = X \) | A6ID |
| A2 | \( \delta \cdot X = \delta \) | A7ID |
| A3 | \( X + X = X \) | |
| A4 | \( (X + Y)Z = XZ + YZ \) | |
| A5 | \( (X \cdot Y)Z = X \cdot (YZ) \) | |

We turn our process domain into a model of BPA₈ by interpreting the signature elements. It is easiest to give the definitions in relative time; such definitions can subsequently be transposed to absolute or parametric time by the canonical embeddings defined in section 2. The constant \( \delta \) is mapped to the graph with one node, the root, labeled 1D, and no edges. In order to define alternative composition, take representatives of the bisimulation classes that are acyclic and have disjoint node sets. In case one graph is the \( \delta \)-graph, the sum is simply the other graph. Otherwise, the sum is the equivalence class of the transition system obtained
by identifying the initial nodes. Thus, we identify the roots, and if both roots have an outgoing \( \sigma \)-edge, both these edges are removed, and a new \( \sigma \)-edge is added to the sum of the graphs the original \( \sigma \)-edges where going to. If necessary, the procedure is repeated. In order to define sequential composition, append at each \( \sqrt{\cdot} \)-node of the first graph a copy of the second graph.

In absolute time, the node of the graph of \( \delta \) gets a ts 0. If we want to add a 2pTSa(A,k) to a 2pTSa(A,k), we identify the initial nodes with the same ts; in case \( T \) is a 2pTSa(A,k) and \( S \) is a 2pTSa(A,n) with \( k \) different from \( n \), say \( k < n \), we apply this to \( T \) and \( k \gg S \). For the sequential composition \( T \cdot S \), with \( T \) a 2pTSa(A,k) and \( S \) a 2pTSa(A,n), we start out from \( T \), and if \( T \) has a \( \sqrt{\cdot} \)-endpoint with ts \( m \), we append a copy of \( m \gg S \). In parametric time, we take one of the above definitions for each component of the sequence.

3.7 Basic process algebra in absolute discrete time

The constructors of discrete time process algebra with absolute timing are the signature elements of BPA\(_a\), and the constants \( a(n+1) \) (\( a \) in the nth time slice) and \( \delta(n+1) \) (deadlock at the end of the nth time slice). An example of a process expression in absolute time is \((a(3) \cdot b(4) + c(2)) \cdot (a(5) + c(7))\).

The initialisation operator \( \gg \) serves as an auxiliary operator for calculations in absolute discrete time.

An example: \( a(5) \cdot b(3) = a(5) \cdot (4 \gg b(3)) = a(5) \cdot \delta(4) = a(5) \cdot \delta \). This means if \( a(5) \) has been performed, time is at least in slice 5 and the goal to perform subsequently an action \( b \) in slice 3 has become obsolete (and unachievable), and immediate deadlock will occur.

Axioms for absolute discrete time basic process algebra, BPA\(_{adap}\), are those of BPA\(_a\) together with the axioms in table 2.

<table>
<thead>
<tr>
<th>Table 2. Additional axioms for BPA(_{adap})</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ a(0) = \delta(0) = \delta ]</td>
</tr>
<tr>
<td>[ \delta(n) \cdot X = \delta(n) ]</td>
</tr>
<tr>
<td>[ n &lt; m \Rightarrow n \gg a(m) = a(m) ]</td>
</tr>
<tr>
<td>[ n \geq m \Rightarrow n \gg a(m) = \delta(n) ]</td>
</tr>
<tr>
<td>[ n \gg \delta(m) = \delta(\max(n,m)) ]</td>
</tr>
<tr>
<td>[ n \gg (X + Y) = (n \gg X) + (n \gg Y) ]</td>
</tr>
<tr>
<td>[ n \gg (X \cdot Y) = (n \gg X) \cdot Y ]</td>
</tr>
</tbody>
</table>

3.8 Examples. We provide some examples of calculations with these axioms:

1. \( a(3) \cdot b(5) \cdot c(2) = a(3) \cdot b(5) \cdot \delta \)
2. \( a(3) \cdot (b(5) + c(2)) = a(3) \cdot b(5) \)

3.9 Basic process algebra in relative discrete time

The constructor signature for relative timing extends the signature of BPA\(_a\) with the constants \( a[n+1] \) (\( a \) in the nth time slice after the current one) and \( \delta[n+1] \) (deadlock at the end of the
The axioms for relative time basic process algebra, $\text{BPA}_{\text{dpr}}$, are those of $\text{BPA}_{\text{R}}$ together with the axioms in Table 3. In order to give the axioms for parametric time basic process algebra, we first consider parametric time conditions.

Table 3. Additional axioms for $\text{BPA}_{\text{dpr}}$

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta[0] = \delta[0] = \delta$</td>
<td>$n &lt; m \Rightarrow \delta[n] + \delta[m] = \delta[m]$</td>
</tr>
<tr>
<td>$\delta[n] \cdot X = \delta[n]$</td>
<td>$\delta[n] + \delta[n] = \delta[n]$</td>
</tr>
</tbody>
</table>

3.10 Basic process algebra in parametric discrete time

We introduce conditions that depend on time. For $c$ a condition, $n \succ c$ will be either true or false, determined by whether $c$ holds in slice $n+1$ or not. Conditions are generated by true, false, $sl(n)$, $sl< (n)$ and $\land, \lor, \neg$ (conjunction, disjunction, negation). $sl(n)$ holds just when the current time slice is slice $n$, $sl< (n)$ holds from slice $n+1$ onwards. Initialisation of conditions is given by the axioms on the left hand side of Table 4. Now conditions may be added to parametric time processes by the first three axioms on the right hand side of Table 4.

In the case of parametric time processes, the constructors are the union of those of absolute and relative time together with the conditional construct $X < b > Y$ (if $b$ then $X$ else $Y$). Here, the condition is to be taken from the domain of time dependent conditions that we outline below.

The axioms of $\text{BPA}_{\text{dpp}}$ are those of $\text{BPA}_{\text{dpr}}$ joined with those of $\text{BPA}_{\text{dpr}}$ together with the axioms in Table 4. For example, $2 \succ sl(3) = \text{true}$ and $k \succ sl(3) = \text{false}$ for $k \neq 2$.

Table 4. Additional axioms for $\text{BPA}_{\text{dpp}}$

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n \succ \text{true} = \text{true}$</td>
<td>$X &lt; \text{true} &gt; Y = X$</td>
</tr>
<tr>
<td>$n \succ \text{false} = \text{false}$</td>
<td>$X &lt; \text{false} &gt; Y = Y$</td>
</tr>
<tr>
<td>$n \succ sl(m) = (n+1=m)$</td>
<td>$n \succ (X &lt; c &gt; Y) = (n \succ X) &lt; (n \succ c) &gt; (n \succ Y)$</td>
</tr>
<tr>
<td>$n \succ sl&lt; (m) = (n+1&gt;m)$</td>
<td></td>
</tr>
<tr>
<td>$n \succ (c \land c') = (n \succ c) \land (n \succ c')$</td>
<td></td>
</tr>
<tr>
<td>$n \succ (c \lor c') = (n \succ c) \lor (n \succ c')$</td>
<td>$n \succ \delta[m+1] = \delta(n+m+1)$</td>
</tr>
<tr>
<td>$n \succ \neg c = \neg(n \succ c)$</td>
<td>$n \succ \delta[m] = \delta(n+m)$</td>
</tr>
</tbody>
</table>

The notation $\_ < \_ > \_$ for the conditional then $\_ \text{ if } \_ \text{ else } \_ \text{ was taken from } [27]$ and has been elaborated in the setting of process algebra in [4]. We have the further abbreviation:

$c :: \rightarrow X = X < c > \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \delta . \δ.
To obtain an axiomatisation of parametric time processes we add the axiom scheme TSE_k to BPA_{dpp}. The explanation of TSE_k is this: if X is initialised then exactly one of the guards sl(1), ..., sl(k+1), sl_{>}(k+1) will hold. If sl(n+1) holds then X is initialised at n (beginning of slice n+1).

The relevance of TSE_k is that it allows to split a process expression into absolute and relative time parts: each n \gg X is an absolute time expression. The useful property of the residue sl_{>}(k+1) ::= X is that for finite process expressions over the parametric time signature, it can be proven equal, for k sufficiently large, to a relative time process. In order to achieve this, we need to add axioms that show that the residue function distributes over all relevant operators. Note that for k > n, sl_{>}(k) ::= a(n) = \delta, so absolute timestamped actions can be eliminated. Thus, the last term is a process in relative time (for k sufficiently large). Readers comfortable with infinite sums may prefer the following time spectrum expansion:

\[ X = \sum_{n=0}^{\infty} sl(n+1) ::= (n \gg X). \]

3.11 Examples. We provide some examples of calculations with the axioms:

1. a(3) \cdot b[2] = a(3) \cdot b(5).
2. a[3] \cdot b(5) = sl(1) ::= a(3) \cdot b(5) + sl(2) ::= a(4) \cdot b(5) + sl(3) ::= a(5) \cdot b(5) + + sl_{>}(3) ::= a[3] \delta.

3.12 Head normal forms

We can reduce each process expression to a so-called head normal form. A process in head normal form will correspond directly to its transition system. We start by defining the notion of a head normal form starting at slice k for absolute time. An absolute time process is hereditarily head normalised if it has a head normal form starting at slice 0, and all its subexpressions are in head normal form starting at the slice of the preceding atomic action.

1. a(n), \delta(n) are head normal forms starting at slice k for any k \leq n;
2. if p is a head normal form starting at slice n, then a(n) \cdot p is a head normal form starting at slice k for any k \leq n;
3. if p, q are head normal forms starting at slice k, then so is p + q.

Using the axioms of BPA_{dap}, we can prove that every closed term over its signature can be written in hereditarily head normal form. We can prove that all axioms of BPA_{dap} are valid in the model of 2pTSa(A,k) bisimulation equivalence classes. Then, we can prove that the axiomatisation is even complete: an identity between closed terms can be proven from the axioms of BPA_{dap} exactly when the corresponding 2pTSa(A,0) graphs are bisimulation equivalent.

In relative time, things works similarly, only a little simpler. Hereditary head normalisation can be defined inductively:

1. a[n], \delta[n] are hereditarily head normalised;
2. If $p$ is hereditarily head normalised, then so is $\alpha[n] \cdot p$.
3. If $p, q$ are hereditarily head normalised, then so is $p + q$.

Again, any closed term over $BPA_{\text{DP}}$ can be written in hereditarily head normal form, and the axiomatisation of $BPA_{\text{DP}}$ is sound and complete for the model of 2pTSr(A) bisimulation equivalence classes.

In parametric time, we use the bounded time spectrum expansion axiom. We find that $BPA_{\text{DP}}$ with bounded time spectrum expansion constitutes a sound and complete axiomatisation of parametric time transition systems modulo (strong) bisimulation.

### 3.13 Extensionality and time spectrum abstraction

Next we introduce a very powerful abstraction mechanism. It is like lambda abstraction, but specific for the case where the parameter is process initialisation time. Using this so-called time spectrum abstraction mechanism, processes can be denoted in a flexible way.

The semantic justification of the abstraction mechanism is given by an extensionality principle (which guarantees the existence of the abstraction viewed as a mathematical function). Let $P, Q$ be two parametric time processes. Extensionality for parametric discrete time is the following rule:

\[
\text{for all } n, n \gg P = n \gg Q \quad \text{EPDT.}
\]

As a consequence we may define a new process $P$ by defining $n \gg P$ for all $n$.

Now, we exploit EPDT by introducing time spectrum abstraction: if for each $k$ $F[k/x]$ is a parametric time process, then the process $n \gg F$ is the parametric time process defined by $n \gg F = n \gg F[n/x]$. An example: the discrete time step operator $\sigma_{\text{rel}}$ on parametric time processes can be defined as follows: $\sigma_{\text{rel}}(P) = \sqrt{d}n \cdot (n+1) \gg P$.

### 3.14 Time free process algebra

Time free actions $a, \delta$ such as usual in ACP, CCS or CSP have the following interpretation:

\[
a = \sum_{n=1}^{\infty} a(n) = \sum_{n=1}^{\infty} a[n], \quad \delta = \sum_{n=1}^{\infty} \delta(n) = \sum_{n=1}^{\infty} \delta[n].
\]

This is the interpretation that untimed actions can happen in any time slice, and can be delayed arbitrarily long. Thus, we see that the interpretation of $a$ as a 2pTSr can be given as a graph with 2 nodes, the first the root, the second a $\sqrt{d}$-node, connected by an $a$-edge, and with a $\sigma$-edge from the root to itself. $\delta$ as a 2pTSr just has one node, with a $\sigma$-edge to itself. We see that both processes are regular. In absolute time, the interpretation of these constants requires an infinite graph. The interpretation of $\delta$ has countably many nodes with increasing $ts$, connected by a $\sigma$-"spine", the interpretation of $a$ has a terminating $a$-step at each node of the graph. In two-phase notation, definition of time free atoms takes the form of a recursive equation:

\[
a = cts(a) + \sigma_{\text{rel}}(a) \quad a = fts(a) + \sigma_{\text{abs}}(a).
\]
We remark that the present definitions (in timestamped and two-phase notation) are by no means the only or in any sense a canonical embedding of untimed atoms into discrete time process algebra. [7] uses the systematic notation $\text{ats}(a)$ ($a$ in any time slice) for the present interpretation of time free atoms, and also defines other interpretations. Since different embeddings are conceivable, we will use the convention that a "user" of discrete time process algebra should feel free to use the embedding (s)he prefers in a given setting.

For each process domain $P_{\text{abs}}(A), P_{\text{rel}}(A), P_{\text{par}}(A)$ we can consider the subalgebra of all processes $X$ that satisfy $X + \delta = X$, all processes that can wait arbitrarily long. Clearly, an atomic action $a$ is contained in these subalgebras, but not all timed processes are contained as e.g. $a(1)$ is not, as it cannot delay until the second time slice. On all three subalgebras, all laws of ACP are valid. We find that ACP specifies an SRM, a Subalgebra of a Reduced Model (the signature elements $a(n), a[n]$ are omitted). The phrase SRM was first used in the context of ACP in [6], where CCS was obtained as an SRM of an extension of ACP with action prefixing.

4. Parallel composition

Next, we introduce parallel composition by means of the merge operator. We start by describing a system with merge without communication, a so-called free merge.

4.1 Process algebra with free merge

We add to the signature of $\text{BPA}_{\text{dep}}$ the binary parallel composition operator $\parallel : P \times P \rightarrow P$ (merge) and the auxiliary left parallel composition operator $\parallel : P \times P \rightarrow P$ (left merge). $X \parallel Y$ denotes the interleaved concurrent composition of $X$ and $Y$, whereas $X \parallel Y$ denotes that part of $X \parallel Y$ which takes its first action from $X$. Axioms for these operators follow the lines of ACP [14, 11]. Their purpose is to allow expansion of the merge, i.e. rewriting $X \parallel Y$ for closed process expressions $X$ and $Y$ into a process expression made by means of + and · only. The first axiom for merge below expresses arbitrary interleaving (given our interpretation of the left merge). We do not commit ourselves philosophically to arbitrary interleaving in any way. Process algebra in the style of ACP, CCS or CSP exploits the simplifying assumption that parallelism may be modeled in an interleaved fashion. There is no implication that all phenomena related to concurrency, be it timed or untimed, are covered in the interleaving models. What we claim however is that quite often arbitrary interleaving is justified as a modeling strategy. That justifies the development of concurrency theory based on arbitrary interleaving.

Again we will distinguish three cases: absolute, relative and parametric time. We will not hesitate to use axiom schemes, a finite axiomatisation can be found in [8]. In each case, the
axioms are to be seen as the extension of the axiom set for the corresponding case without parallelism. Common to each case are the axioms in table 5.

Table 5. Axioms for free merge common to absolute, relative and parametric time

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \parallel Y = X \parallel Y + Y \parallel X$</td>
<td>$\delta \parallel X = \delta$</td>
</tr>
<tr>
<td>$(X + Y) \parallel Z = X \parallel Z + Y \parallel Z$</td>
<td>$X \parallel \delta = \delta$</td>
</tr>
</tbody>
</table>

4.2 Absolute timing

For absolute time, we have in addition 8 axiom schemes. We show 4 of them in table 6.

Table 6. Axioms for free merge in absolute timing

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m &lt; n \Rightarrow a(n) \cdot X \parallel (b(m) + Z) = a(n) \cdot X \parallel (\delta(m) + Z)$</td>
<td>$m &lt; n \Rightarrow a(n) \cdot X \parallel (b(m) \cdot Y + Z) = a(n) \cdot X \parallel (\delta(m) + Z)$</td>
</tr>
<tr>
<td>$m &lt; n \Rightarrow a(n) \cdot X \parallel \delta(m) = \delta(m)$</td>
<td>$a(n+1) \cdot X \parallel (n \gg Z) = a(n+1) \cdot (X \parallel Z)$</td>
</tr>
</tbody>
</table>

The other four axiom schemes are obtained from these by deleting $X$ in each case. Some explanation: consider $P = a(n) \cdot X \parallel (b(m) \cdot Y + Z)$. If $a(n)$ is the first action then $b(m)$ can never occur if $m < n$. So in that case it is replaced by $\delta(m)$. This term $\delta(m)$ may either vanish, in the presence of other summands that can wait longer, or otherwise all other summands of $Z$ vanish. In the latter case one finds $a(n) \cdot X \parallel \delta(m)$ with $m < n$. This process expression does not allow to wait until slice $n$, because during slice $m$ a deadlock occurs in the right-hand process.

Now suppose $Z$ has the form $\Sigma a_i(n_i) \cdot X_i + \Sigma b_j(m_j)$, with all $n_i, m_j > n$. Then $Z = n \gg Z$, so we may write $a(n+1) \cdot X \parallel Z = a(n+1) \cdot X \parallel (n \gg Z)$. Here the right hand side can wait sufficiently long to let the $a(n+1)$ take place first, so this term equals $a(n+1) \cdot (X \parallel Z)$.

Using the axioms in table 5 and 6, we can write each closed term over the present theory into hereditarily head normal form. Thus, the operators $\parallel, \parallel$ can be eliminated from closed terms. This is called the elimination theorem.

4.3 Examples

1. $(a(3) \cdot b(5)) \parallel c(4) = a(3) \cdot c(4) \cdot b(5)$.
2. $a(3) \parallel b(2) = \delta(2)$.
3. $a(3) \cdot (b(5) \parallel c(2)) = a(3) \cdot \delta$.

4.4 Transition systems

The interpretation of the new operators on absolute time transition systems is as follows.

- If $T,S$ are both $2pTsa(A,k)$, the $2pTsa(A,k)$ $T \parallel S$ has node set $\{(t,s) : t$ a node of $T$, $s$ a node of $S$, $ts(s) = ts(t)$, $ts((s,t)) = ts(s) (= ts(t))$}. If $t \overset{a}{\rightarrow} t'$ and $s \overset{a}{\rightarrow} s'$ and $ts(t) = ts(s)$, then $(t, s) \overset{a}{\rightarrow} (t', s')$. If $t \overset{a}{\rightarrow} t'$ and $ts(t) = ts(s)$ and $s$ does not have an ID-label, then $(t, s) \overset{a}{\rightarrow} \langle t', s' \rangle$. If $s \overset{a}{\rightarrow} s'$ and $ts(t) = ts(s)$ and $t$ does not have an ID-label, then $(t, s) \overset{a}{\rightarrow} \langle t', s' \rangle$. The root
is the pair of roots. A node gets a \( \vee \)-label only if both components had a \( \vee \)-label. A node gets an ID-label if one of the components had an ID-label. If \( T \) is \( 2pTSa(A,k) \) and \( S \) is a \( 2pTSa(A,n) \) with \( k \) and \( n \) different, say \( k < n \), apply the above construction to \( T \) and \( k \gg S \).

- If \( T \) is \( 2pTSa(A,k) \) and \( S \) is a \( 2pTSa(A,n) \), in order to obtain \( T \| S \), remove from \( T \| S \) all edges \( (t, s) \to (t, s') \) where \( (t, s) \) is an initial node. Further, unreachable parts of the transition system can be removed.

4.5 Relative time

This case is similar to the absolute time case. Again, 8 axiom schemes are needed. We present the first four in table 7. The fourth of these uses the relative time step operator instead of the initialisation operator of the absolute time case. We give some axioms for this operator as well. We again have an elimination theorem as in 4.2.

<table>
<thead>
<tr>
<th>Table 7. Axioms for free merge in relative timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m &lt; n \implies a[n]X | (b[m] + Z) = a[n]X | (\delta[m] + Z) )</td>
</tr>
<tr>
<td>( m &lt; n \implies a[n]X | (b[m]Y + Z) = a[n]X | (\delta[m] + Z) )</td>
</tr>
<tr>
<td>( m &lt; n \implies a[n]X | \delta[m] = \delta[m] )</td>
</tr>
<tr>
<td>( a[n+1]X | \sigma_{rel}^n(Z) = a[n+1](X | Z) )</td>
</tr>
</tbody>
</table>

4.6 Parametric time

In the parametric time case, we need all 16 axiom schemes for absolute and relative time. In addition we have the axioms in table 8. We see that initialising a parallel composition amounts to independent initialisation of both components. Using the full axiom set, we can obtain an elimination theorem for parametric time processes: the operators \( \|, \| \) can be eliminated from closed terms. We give a couple of examples in 4.7 below.

<table>
<thead>
<tr>
<th>Table 8. Axioms for free merge in parametric timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n \gg (X | Y) = (n \gg X) | (n \gg Y) )</td>
</tr>
</tbody>
</table>

4.7 Examples. We give some examples of the elimination of parallel composition. These examples realise our primary objective: to provide a clear and unambiguous relation between absolute, relative and parametric time notation. It is surprising that the conditional construct \( (::\to) \) appears as a key operator. Its role is central due to the use of time dependent conditions.

1. \( a[2] \| b(3) = s(1) ::\to a(2) \cdot b(3) + s(2) ::\to (a(3) \cdot b(3) + b(3) \cdot a(3)) + s(3) ::\to b(3) \cdot a(4) \).

2. \( a[2] \| (b(3) + c[1]) = s(1) ::\to (c(1) \cdot a(2) + a(2) \cdot b(3)) + s(2) ::\to (c(2) \cdot a(3) + a(3) \cdot b(3) + b(3) \cdot a(3)) + a(3) \cdot b(3) + b(3) \cdot a(3)) + c(3) \cdot b(3) + b(3) \cdot c(3) \).
+ sl(3) :→ (b(3)·a(4) + c(3)·a(4)) + sl(3) :→ c[1]·a[2].

4.8 Merge with communication

The case with communication is of course the most important one. The first axiom of table 9 is the expansion law, replacing the first axiom of table 5. Here, X | Y, the communication merge of X and Y, contains those parts of X||Y that begin with an action that embodies the simultaneous execution of a first action from X and a first action from Y. We assume that I is given on atomic actions and δ as a commutative and associative mapping with δ acting as a zero element. Again, we have three cases, requiring additional axioms for communication merge.

Table 9. Axioms for merge with communication common to absolute, relative and parametric time

<table>
<thead>
<tr>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>(X + Y)</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>(X + Y)</td>
</tr>
</tbody>
</table>

4.9 Absolute time

The following axiom schemes are needed. On the basis of these axiom schemes, we can achieve elimination: the operators ||, L, I can be eliminated from closed terms.

Table 10. Axioms for communication merge in absolute timing

<table>
<thead>
<tr>
<th>Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = m ⇒ a(n)·b(m) = δ(min(n, m))</td>
</tr>
<tr>
<td>a(n)·b(n) = a,b(n)</td>
</tr>
<tr>
<td>a(n)·X</td>
</tr>
<tr>
<td>a(n)·X</td>
</tr>
</tbody>
</table>

On absolute time transition systems, we modify the definition of parallel composition by adding in addition edges (t, s) → (t', s') whenever t → t', s → s', ts(t) = ts(s) and a | b = c is an atomic action. The definition of the left-merge is modified by also removing these additional edges from initial nodes. The transition system of T || S is obtained from T || S by removing all edges (t, s) → (t, s') and (t, s) → (t, s') from initial nodes (thus, the first action must be a communication action).

4.10 Example.

(a(3) + b(5)) || b(3) = a(3)·c(3) + b(3)·(a(3) + b(5)) + a | b(3).
4.11 Relative time and parametric time

The relative time case is like the absolute time case, where the axioms have square brackets instead of round ones. The parametric time axioms are similar to those in 4.6. Again, we will have an elimination theorem.

Table 11. Axioms for communication merge in parametric timing

\[ n \gg (X \mid Y) = (n \gg X) \mid (n \gg Y) \quad \text{sl}_2(n) ::= (X \mid Y) = (\text{sl}_2(n) ::= X) \mid (\text{sl}_2(n) ::= Y) \]

4.12 Example.

\[ \text{a}[2] \parallel \text{b}(3) = \text{sl}(1) ::= \text{a}(2) \cdot \text{b}(3) + \text{sl}(2) ::= (\text{a}(3) \cdot \text{b}(3) + \text{b}(3) \cdot \text{a}(3) + \text{a} \parallel \text{b}(3)) + + \text{sl}(3) ::= \text{b}(3) \cdot \text{a}(4). \]

4.13 Encapsulation

Like in ACP [14], an encapsulation operator is needed. The main application is to block failed communications at internal ports in a system. Thus, in \( \partial_H(X) \), all actions from \( X \) that are elements of the set of atomic actions \( H \), are blocked. On transition systems, \( \partial_H(T) \) is obtained from \( T \) by removing all \( a \)-edges with \( a \in H \). Further, unreachable parts of the transition system can be removed.

The first two axioms in table 12 are common to all three cases, the next two cover the absolute time case, the following two the relative case, and the last two are additional for the parametric time case. Also the encapsulation operator can be eliminated from closed terms. We give an example in the following.

Table 12. Axioms for encapsulation

\[
\begin{align*}
\partial_H(X + Y) + \partial_H(X) + \partial_H(Y) & \quad \partial_H(X \cdot Y) + \partial_H(X) \cdot \partial_H(Y) \\
\partial_H(a[n]) = a[n] & \text{if } a \notin H \\
\partial_H(a[n]) = a[n] & \text{if } a \in H \\
n \gg \partial_H(X) = \partial_H(n \gg X) & \\
\text{sl}_2(n) ::= \partial_H(X) = \partial_H(\text{sl}_2(n) ::= X)
\end{align*}
\]

4.14 Example. Suppose \( a \parallel b = c \) and \( H = \{a, b\} \). Then

\[
\partial_H(a[2] \parallel (b(3) + b(4))) = \text{sl}(1) ::= \delta(2) + \text{sl}(2) ::= c(3) + \text{sl}(3) ::= c(4) + \text{sl}(4) ::= \delta(4).
\]

4.15 Abstraction

In [7], the present theory was extended with the silent step \( \tau \), originally due to [34] for CCS. The use of silent steps allows the introduction of a notion of abstraction, essential in system design, where we need to consider a system at different levels of abstraction, and in system verification, where we need to abstract from internal behaviour.
Due to space limitations, we cannot discuss silent steps in technical detail. For those readers familiar with the use of silent steps in time free process algebra, we remark that in our view the primary role of silent steps is to be used in what Milner calls communication engineering. Communication engineering involves the composition of processes from independent building blocks. It is implicitly understood that communication is only a small part of behaviour and that most behaviour is inherited from the building blocks.

Another use for silent steps is in system verification. There however, it is often a handicap that bisimulation is still a rather low level of abstraction.

5. Example

We consider a very simple example in the theory presented in the previous sections. First, we need an iteration construct.

5.1 Binary Kleene star

The so-called binary Kleene star operator has defining equation $X^*Y = X \cdot (X^*Y) + Y$, and gives a simple form of recursion. If $X$ and $Y$ are regular processes, then so is $X^*Y$. For further information, see [13, 21].

5.2 Message passing

Suppose the set of atomic actions $A$ contains elements $\text{snd}(msg1)$, $\text{rcv}(msg1)$, $\text{comm}(msg1)$ (denoting resp. the sending, receiving and communication of message $msg1$) with defined communication $\text{snd}(msg1) \parallel \text{rcv}(msg1) = \text{comm}(msg1)$.

Similarly, we have these atomic actions for other messages. Define a sender process $P$ by

$$P = 0 \gg (\text{snd}(msg1)[61] * \delta).$$

This is an absolute time process that sends the message $msg1$ in slice 61, 121, 181, etc.

The process $Q$ is waiting for this message:

$$Q = (\text{rcv}(msg1) \cdot \text{snd}(msg2)[3]) * \delta.$$

$Q$ is a relative time process. After receiving $msg1$ in any time slice, it will send a $msg2$ 2 slices later.

Now we compose these two processes in parallel. Define

$$R = \partial_{\text{snd}(msg1), \text{rcv}(msg1)}(P \parallel Q).$$

$R$ is a parametric time expression, denoting an absolute time process, and describes a possible interaction between $P$ and $Q$. We notice that it seems entirely natural to view $P$ as an absolute time process and $Q$ as a relative time one.
6. Applications

We give some further remarks on the use of discrete time process algebra in programming. We discuss timed actions in the TOOLBUS [15, 16] that was already mentioned in the introduction. Further, we give some remarks on SDL [41], a specification language which may be considered a programming language just as well. For reasons of simplicity we discuss SDL in terms of flat SDL (flat SDL) a subset that is implicit in the official standard [41], in the sense that translations are given which remove constructs outside flat SDL.

First, however, we will discuss deadlocks and livelocks in the context of discrete time.

6.1 Deadlocks, time stops and livelocks

A process $a(7) \cdot \delta$ has an immediate deadlock, that occurs immediately after $a$ has been performed in slice 7. On the other hand, $a(7) \cdot \delta$ shows a livelock, after performing $a$ in slice 7. Immediate deadlocks will not occur in a system model other than as auxiliary objects. In contrast, livelocks are plausible elements of a system model.

We remark that there is a third form of time stop that warrants attention: consider the process $a(7) \cdot (b[1] * \delta)$. This process performs $a$ in slice 7 and then continues with an infinite sequence of $b$'s, all in slice 7. This is a so-called Zeno process which defeats execution. If we assume that the actions $b$ are internal and unobservable then $a(7) \cdot (b[1] * \delta)$ is very similar to $a(7) \cdot \delta(7)$ and it embodies a deadlock, though not an immediate one. In our comments on SDL [41] we will indicate that Zeno processes will occur as plausible elements of system models.

6.2 Discrete time features in the ToolBus

There are many practical uses conceivable for timed process algebras. As an example, we mention the TOOLBUS (see [15, 16]). The TOOLBUS is a software interconnection mechanism which allows a system programmer to put software components (tools) in parallel and to program their cooperation by means of a parallel composition of so-called T-scripts. These T-scripts are written in a program notation named T. The notation T is worth mentioning here because it is a slightly sugared version of a subalgebra of ACP with parametric discrete time. By using randomised symbolic execution the TOOLBUS implementation enacts that the axioms of process algebra can be viewed as correctness preserving transformations of T-scripts.

T allows time free atomic actions, and six kinds of timed primitive actions. As structuring primitives there are:
- alternative composition (+)
- sequential composition (\cdot)
- iteration
- parallel composition (free merge)
- conditionals
- process creation (not discussed in this survey).

In this algebra programming of the TOOLBuS using T-scripts takes place. The picture is still somewhat simplified because actions in T-scripts may change a global state and because we have omitted the value passing mechanism. These features, however, are entirely independent of timing and can be added in a standard fashion.

Because T is a conventional imperative program notation it is not conceivable that a time stop is programmed. This leads to a view of time free actions as actions that can be performed at any time and that a process attempts to perform as soon as possible. This is captured well by the atoms of time free process algebra (see 3.10). The timed actions will not introduce time stops either. We will discuss two of the six kinds of timed actions. First, if e,f denote terms of type \( \mathbb{N} \), possibly containing a free variable \( \text{time} \) denoting the current time, then we have

\[
\text{abs-interval}(a, e, f).
\]

The informal meaning is that when initialised at \( m \), \( a \) is to be performed (if possible) between \( e \) and \( f-1 \), with \( m \) substituted for \( \text{time} \). Similarly, we have

\[
\text{rel-interval}(a, e, f),
\]

denoting that when initialised at \( m \), \( a \) is to be performed (if possible) between \( m + e \) and \( m + f - 1 \), with again \( m \) substituted for \( \text{time} \).

Formally, the meaning of the two timed actions is as follows:

\[
\text{abs-interval}(a, e, f) = \bigvee_m \mathbb{N} \sum_{i = 1}^{\infty}(a(i) < e[m/time] \leq i < f[m/time] \geq \delta) + \delta
\]

\[
\text{rel-interval}(a, e, f) = \bigvee_m \mathbb{N} \sum_{i = 1}^{\infty}(a[i] < e[m/time] \leq i < f[m/time] > \delta) + \delta
\]

We recall that \( \bigvee_m \mathbb{N} F \) is defined by \( k \gg \bigvee_m \mathbb{N} F \) for all \( k \in \mathbb{N} \). The summands \( \delta \) ensure that as a default, the action can wait arbitrarily long thus avoiding the unintended introduction of time stops.

6.3 Connections with SDL

A process algebra view of SDL, or rather of its sublanguage \( \phi \text{SDL} \) (with \( \phi \) standing for flat: no blocks and no channels) has been given in [17]. The semantics given there also requires a subset of ACP with absolute discrete time. SDL [41] is a specification language that actually is executable. We will not provide any details, but we notice a key difference with the TOOLBuS. SDL is meant to be a specification language for telecommunication applications. Systems described in SDL will use very little processing time and spend most time waiting for external events. Therefore, processor load is not supposed to be a limiting factor and that is reflected by a maximal progress assumption which in turn shows up in the process algebra.
in the presence of timed actions. $\phi$SDL programs are such that time stops will not occur but an infinite sequence of actions within one time slice may be required. Since this is technically impossible, it amounts to a time stop, be it not an immediate one, but rather the form of time stop indicating a Zeno process.

6.4 Other applications for discrete time

The design of TOOLBUS scripts and the semantics of $\phi$SDL are just two applications of timed process algebra. In [33] one finds a simplified form of discrete time process algebra that has been extensively used for the specification and verification of digital circuitry. A similar matter is the semantic modeling of synchronous dataflow systems, for which discrete time process algebra is perfectly suited. Several proposals have been made to extend the specification language LOTOS with timing constructs, and these proposals have been based on forms of discrete time process algebra.

We expect that many more applications of discretely timed processes will emerge in coming years. A first example is [19].

References


Computing Science Reports

In this series appeared:

93/01 R. van Geldrop
Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.

93/02 T. Verhoef
A continuous version of the Prisoner's Dilemma, p. 17

93/03 T. Verhoef
Quicksort for linked lists, p. 8.

93/04 E.H.L. Aarts, J.H.M. Koest, P.J. Zwiertering
Deterministic and randomized local search, p. 78.

93/05 J.C.M. Baeten, C. Verhoef
A congruence theorem for structured operational semantics with predicates, p. 18.

93/06 J.P. Veltkamp
On the unavoidability of metastable behaviour, p. 29

93/07 P.D. Moerland
Exercises in Multiprogramming, p. 97

93/08 J. Verhoosel
A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.

93/09 K.M. van Hee

93/10 K.M. van Hee
Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.

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. Hoorna
A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27.

93/17 D. Aistein, P. van der Stok
Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.

93/18 C. Verhoef
A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.

93/19 G.J. Houben
The Design of an Online Help Facility for ExSpect, p. 21.

93/20 F.S. de Boer

93/21 M. Codish, D. Reis, G. Fitt, M. Bruynooghe
Freeness Analysis for Logic Programs - And Correctness, p. 24

93/22 E. Poll
A Typechecker for Bijective Pure Type Systems, p. 28.

93/23 E. de Kogel
Relational Algebra and Equational Proofs, p. 23.

93/24 E. Poll and Paula Severi
Pure Type Systems with Definitions, p. 38.

93/25 H. Schepers and R. Gerth

93/26 W.M.P. van der Aalst
Multi-dimensional Petri nets, p. 25.

93/27 T. Kloks and D. Kratsch
Finding all minimal separators of a graph, p. 11.

93/28 F. Kamareddine and R. Nederpelt
A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.

93/29 R. Post and P. De Bra
GOLD, a Graph Oriented Language for Databases, p. 42.

93/30 J. Deogun, T. Kloks, D. Kratsch, H. Müller
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. Korver</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. Moonen</td>
<td>ILIAS, 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>93/38</td>
<td>C. Verhoef</td>
<td>A general conservative extension theorem in process algebra, p. 17.</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. Ramburgs</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>93/48</td>
<td>R. Gerth</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
<tr>
<td>94/01</td>
<td>P. America, M. van der Kammen, R.P. Nederpelt, O.S. van Roosmalen, and H.C.M. de Swart</td>
<td>The object-oriented paradigm, p. 28.</td>
</tr>
<tr>
<td>94/02</td>
<td>F. Kamareddine and R.P. Nederpelt</td>
<td>Canonical typing and II-conversion, p. 51.</td>
</tr>
<tr>
<td>94/04</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.</td>
</tr>
<tr>
<td>94/05</td>
<td>P. Zhou and J. Hoornan</td>
<td>Formed Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.</td>
</tr>
<tr>
<td>94/06</td>
<td>T. Basten, T. Kunz, J. Black, M. Coffin, and D. Taylor</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 and J.A. Bergstra</td>
<td>Process Algebra with Partial Choice, p. 16.</td>
</tr>
</tbody>
</table>
The testing Paradigm Applied to Network Structure. p. 31.


A New Method for Integrity Constraint checking in Deductive Databases. p. 34.

Ups and Downs of Type Theory, p. 9.

Job Shop Scheduling by Local Search. p. 21.

Mathematical Induction Made Calculational, p. 36.

An Algebraic Semantics of Basic Message Sequence Charts, p. 9.

Refining Reduction in the Lambda Calculus, p. 15.

The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.

Beyond β-Reduction in Church's λ→, p. 22.

An introduction to the FIRE engine: A C++ toolkit for Finite automata and Regular Expressions.

The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.

An algebraic semantics of Message Sequence Charts, p. 43.

Abstract Interpretation of Reactive Systems: Abstractions Preserving YCTL*, 3CTL* and CTL*, p. 28.

K Bipartite-free and W 4 -free graphs, p. 10.

On the foundations of functional programming: a programmer's point of view, p. 54.


Correctness of Real Time Systems by Construction, p. 22.

Process Algebra with Feedback, p. 22.

A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.


A formalization of the Ramified Type Theory, p.40.

The Barendregt Cube with Definitions and Generalised Reduction, p. 37.

Delayed choice: an operator for joining Message Sequence Charts, p. 15.

Canonical typing and II-conversion in the Barendregt Cube, p. 19.

Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.

Point-free substitution, p. 10.

On the equivalence covering number of splitgraphs, p. 4.
<table>
<thead>
<tr>
<th>Page</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>94/40</td>
<td>D. Alstein</td>
<td>Distributed Consensus and Hard Real-Time Systems, p. 34.</td>
</tr>
<tr>
<td>94/41</td>
<td>T. Kloks, D. Kratsch</td>
<td>Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.</td>
</tr>
<tr>
<td>94/43</td>
<td>R.C. Backhouse, M. Bijserveld</td>
<td>Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.</td>
</tr>
<tr>
<td>94/45</td>
<td>G.J. Heuben</td>
<td>Tutorial voor de ExSpect-bibliotheek voor &quot;Administratieve Logistiek&quot;, p. 43.</td>
</tr>
<tr>
<td>94/46</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>The λ-cube with classes of terms modulo conversion, p. 16.</td>
</tr>
<tr>
<td>94/47</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>On II-conversion in Type Theory, p. 12.</td>
</tr>
<tr>
<td>94/48</td>
<td>Mathematics of Program Construction Group</td>
<td>Fixed-Point Calculus, p. 11.</td>
</tr>
<tr>
<td>94/50</td>
<td>H. Gevers</td>
<td>A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.</td>
</tr>
<tr>
<td>94/52</td>
<td>W. Penczek, R. Kuiper</td>
<td>Traces and Logic, p. 81</td>
</tr>
<tr>
<td>95/01</td>
<td>J.J. Luikken</td>
<td>The Construction of a small Communication Library, p.16.</td>
</tr>
<tr>
<td>95/02</td>
<td>M. Bezem, R. Bol, J.F. Groote</td>
<td>Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.</td>
</tr>
<tr>
<td>95/03</td>
<td>J.C.M. Baeten, C. Verheijen</td>
<td>Concrete process algebra, p. 134.</td>
</tr>
<tr>
<td>95/04</td>
<td>J. Hidders</td>
<td>An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.</td>
</tr>
<tr>
<td>95/05</td>
<td>P. Severi</td>
<td>A Type Inference Algorithm for Pure Type Systems, p.20.</td>
</tr>
<tr>
<td>95/06</td>
<td>T.W.M. Vossen, M.G.A. Verhoeven, H.M.M. ten Eikelder</td>
<td>A Quantitative Analysis of Iterated Local Search, p.23.</td>
</tr>
<tr>
<td>95/07</td>
<td>G.A.M. de Bruyn, O.S. van Roosmalen</td>
<td>Drawing Execution Graphs by Parsing, p. 10.</td>
</tr>
<tr>
<td>95/08</td>
<td>R. Bloo</td>
<td>Preservation of Strong Normalisation for Explicit Substitution, p. 12.</td>
</tr>
<tr>
<td>95/09</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete Time Process Algebra, p. 20.</td>
</tr>
<tr>
<td>95/10</td>
<td>R.C. Backhouse, R. Verheijen, O. Weber</td>
<td>MathJpad: A System for On-Line Preparation of Mathematical Documents, p. 15</td>
</tr>
<tr>
<td>95/11</td>
<td>R. Seljeé</td>
<td>Deductive Database Systems and integrity constraint checking, p. 36.</td>
</tr>
<tr>
<td>95/12</td>
<td>S. Mauw and M. Reniers</td>
<td>Empty Interworkings and Refinement</td>
</tr>
</tbody>
</table>
Semantics of Interworkings Revised, p. 19.

95/13 B.W. Watson and G. Zwaan

95/14 A. Ponce, C. Verhoef, S.F.M. Vlijmen (eds.)
De proceedings: ACP'95, p.

95/15 P. Niebert and W. Penczek
On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.

95/16 D. Dams, O. Grumberg, R. Gerth
Abstract Interpretation of Reactive Systems: Preservation of CTL*., p. 27.

95/17 S. Mauw and E.A. van der Meulen
Specification of tools for Message Sequence Charts, p. 36.

95/18 F. Kamareddine and T. Laan

95/19 J.C.M. Baeten and J.A. Bergstra
Discrete Time Process Algebra with Abstraction, p. 15.

95/20 F. van Raamsdonk and P. Severi
On Normalisation, p. 33.

95/21 A. van Deursen
Axiomatizing Early and Late Input by Variable Elimination, p. 44.

95/22 B. Arnold, A. v. Deursen, M. Res

95/23 W.M.P. van der Aalst
Petri net based scheduling, p. 20.

95/24 F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen

95/25 L. Feija
Synchronous Sequence Charts In Action, p. 36.

95/26 W.M.P. van der Aalst
A Class of Petri nets for modeling and analyzing business processes, p. 24.

95/27 P.D.V. van der Stok, J. van der Wal

95/28 W. Fokkink, C. Verhoef

95/29 H. Mars

95/30 P. Katsy, D. Dams and R. Gerth
Practical Symbolic Model Checking of the full Mu-calculus using Compositional Abstractions, p. 17.

95/32 W.M.P. van der Aalst

95/33 J. Engelfriet and J.J. Vereijken
Context-Free Graph Grammars and Concatenation of Graphs, p. 35.

95/34 J. Zwanenburg
Record concatenation with intersection types, p. 46.

95/35 T. Basten and M. Voorhoeve
An algebraic semantics for hierarchical P/T Nets, p. 32.

95/36 M. Voorhoeve and T. Basten
Process Algebra with Autonomous Actions, p. 12.

95/37 P. de Bra and A. Aerts
Multi-User Publishing in the Web: DresSS, A Document Repository Service Station, p. 12

95/38 W.M.P. van der Aalst

95/39 S. Mauw
Example specifications in phi-SDL.

96/05 T. Basten and W.M.P. v.d. Aalst

96/06 W.M.P. van der Aalst and T. Basten
Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.

96/07 M. Voorhoeve
Structural Petri Net Equivalence, p. 16.

96/08 A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk

96/09 F. Dignum, H. Weigand, E. Verharen

96/10 R. Bloo, H. Geuvers

96/11 T. Laan
AUTOMATH and Pure Type Systems, p. 30.

96/12 F. Kamareddine and T. Laan
A Correspondence between NuPrl and the Ramified Theory of Types, p. 12.

96/13 T. Borgfors
Priorean Tense Logics in Modal Pure Type Systems, p. 61

96/14 S.H.J. Bos and M.A. Reniers
The \( L^2 \) C-bus in Discrete-Time Process Algebra, p. 25.

96/15 M.A. Reniers and J.J. Vereijken
Completeness in Discrete-Time Process Algebra, p. 139.

96/16 P. Hoogendijk and O. de Moor
What is a data type?, p. 29.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1996/7</td>
<td>E. Boiten and P. Hoogendijk</td>
<td>Nested collections and polytypism, p. 11.</td>
</tr>
<tr>
<td>1996/9</td>
<td>M.A. Reniers</td>
<td>Static Semantics of Message Sequence Charts, p. 71</td>
</tr>
<tr>
<td>1996/20</td>
<td>L. Feijs</td>
<td>Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.</td>
</tr>
<tr>
<td>1996/23</td>
<td>W.M.P. van der Aalst</td>
<td>Structural Characterizations of sound workflow nets, p. 22.</td>
</tr>
<tr>
<td>1996/24</td>
<td>M. Voorhoeve and W. van der Aalst</td>
<td>Conservative Adaption of Workflow, p.22</td>
</tr>
<tr>
<td>1996/25</td>
<td>M. Vaccari and R.C. Backhouse</td>
<td>Deriving a systolic regular language recognizer, p. 28</td>
</tr>
</tbody>
</table>