Structured operational semantics of chi

Citation for published version (APA):

Document status and date:
Published: 01/01/1999

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

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.
Structured Operational Semantics of $\chi$

By

V. Bos and J.J.T. Kleijn

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
Structured Operational Semantics of $\chi$

V. Bos and J.J.T. Kleijn

Abstract

An operational semantics of the systems engineering specification formalism $\chi$ is given using an SOS-style (structured operational semantics) semantics [PIo81]. The semantics is given in two steps. First, a minimal version of $\chi$ is defined in order to decrease the number of syntactic constructs. In order to make the constructs context insensitive, the resulting language is generalized by relaxing some syntactic constraints. Next, the generalized language is assigned an operational semantics. Some properties of the semantics are proven and various examples are given to illustrate the definitions.

1 Introduction

$\chi$ is a systems engineering specification formalism suited to model and analyze industrial systems. An industrial system is considered to be a factory or an autonomous part of a factory. $\chi$ is developed at the Systems Engineering Group of the Eindhoven University of Technology (The Netherlands). The research and development activities subjected to $\chi$ are justified by recent trends in modern industry: First of all, due to the high investments involved, one cannot afford mistakes in building new factories. Furthermore, step by step improvement of a factory is not an option anymore — the final system has to be optimal from the start. Second, the increasing complexity of modern factories requires new tools and methodologies to perform efficient analysis. Finally, since products are more rapidly replaced by new products, new factories have to be realized in less time.

The research activities of the Systems Engineering Group resulted in the specification formalism $\chi$ [vdMFRvdN95, Are96] and a discrete-event simulator for $\chi$ specifications [NA98]. Currently, there exist two versions of $\chi$: discrete $\chi$ and hybrid $\chi$. Discrete $\chi$ can be used to specify discrete event systems, see [Roo96]. In discrete $\chi$, the values of the variables change at discrete time points. Hybrid $\chi$ can be used to specify mixtures of continuous systems and discrete event systems, see [FvBR98, vBR98]. In addition to discrete variables, hybrid $\chi$ has continuous variables that change continuously while time passes. Case studies of real-life production systems proved the effectiveness of $\chi$ and the discrete-event simulator [vdMFR97]. However, analysis based on simulation is not always sufficient: simulation can never validate statements about all possible behaviors of the system simulated. A more rigorous approach should be taken in order to meet hard requirements involving just these kind of statements. In these situations, formal analysis of $\chi$ specifications is the only option left. In [KRR98], a $\chi$ specification is formally analyzed using the process algebra ACP [BV95, BW95]. This showed that formal verification is very well suited to analyze all possible behaviors of a $\chi$ specification. In addition, it also showed that, due to formal verification techniques, it is possible to investigate (parts of) a parameterized $\chi$ specification in isolation. Thus, properties of a class of systems can be verified. However, as the authors mention, there is a problem: the translation from $\chi$ into ACP is carried out informally, leaving possibilities for different interpretations of $\chi$ specifications. To overcome this problem, a formal semantics of $\chi$ is needed.

A formal semantics can be given in different styles, axiomatic (or algebraic), denotational, or operational (see [HJ98], Chapter 0). Which style is better depends on the analysis to be performed. An axiomatic semantics postulates a, usually small, set of axioms describing equality between two (syntactic) terms. Using the axioms as rewrite rules, one can ‘calculate’ if a term
has certain properties. A denotational semantics assigns each language construct a meaning in terms of a mathematical entity (e.g., a set, a function, or a relation). The, usually well known, mathematical theory of these entities is then applied to analyze properties of syntactic terms. Finally, an operational semantics describes how a program or process written in a formalism is executed. Typically, execution is described by defining a set of possible states and a transition relation on these states.

We concentrate on the operational semantics of $\chi$ for several reasons. First, an operational semantics describes the dynamic behavior of $\chi$ specifications. By using the $\chi$ simulator for small exercises as well as larger case studies, $\chi$ users have a thorough, albeit informal, intuition about the dynamics of $\chi$. This intuition served as a starting point for the formalization of the operational semantics. On the other hand, while at the moment simulation is the only means to analyze $\chi$ specifications, both knowledge about the meaning of $\chi$ in terms of mathematical entities, and a clear intuition about equality of syntactic $\chi$ terms still has to be developed. Third, plans are made to design a new simulator which is able to simulate both discrete and hybrid $\chi$ specifications. A clear description of the operational semantics is indispensable for a good design of the new simulator.

In [vdMF95], an operational semantics of $\chi$ has been defined. The semantics served as a guide during the implementation of the discrete-event simulator. By careful inspection, it turned out that there were some discrepancies between the semantics and the behavior of the simulator. After discussions with members of the Systems Engineering Group, it appeared that the definition of the semantics was difficult to understand, which had led to the incorrect implementation of the simulator. Therefore, we present an operational semantics of $\chi$ written in a special notation SOS (Structured Operational Semantics) which is due to Plotkin [Plo81]. It is generally believed that sos-definitions are intuitive and, therefore, we think the semantics defined in this report is easier to understand. The major differences between the two semantics are listed in Section 5.

Since we are interested in the dynamic aspects of $\chi$, we only consider a minimal subset which we call $\chi^-$. This subset has several context sensitive constructs which would require complex sos-definitions. Therefore, we add some new constructs to $\chi^-$ and relax some syntactic constraints. This results in the language $\chi^+$. The relation between the two languages is described by a translation that maps $\chi^-$ constructs onto $\chi^+$ constructs. Using the translation and the formal semantics of $\chi^+$, we can formally analyze $\chi^-$ processes.

Overview  Section 1.1 lists the design decisions underlying $\chi$. The decisions were made during the development of $\chi$ and should be considered requirements for the operational semantics. Section 1.2 introduces notations and conventions used throughout this report. Section 2 defines the syntax of $\chi^-$ and illustrates it with some examples. Section 3 is the main part of this report. It starts off with the definition of $\chi^+$ (Section 3.1). Section 3.2 defines the translation from $\chi^-$ constructs into $\chi^+$ constructs. The first part of the operational semantics of $\chi^+$ is defined in Section 3.3. This part does not involve priority rules of $\chi$. Section 3.4 extends the operational semantics to accommodate for these priority rules. The sos-definitions, including those involving priority rules, are illustrated by examples in Section 3.5. In Section 3.6, we briefly discuss some differences between $\chi^-$ and $\chi^+$. Section 4 describes how the sos-definitions were implemented in the ASF+SDF meta environment [Kli93]. This environment provides tools to quickly design and prototype formal languages. We have used the tools to implement a (very basic) simulator for $\chi^+$. Section 5 deals with the differences between $\chi$, as it is used in real life, and the minimized versions $\chi^-$ and $\chi^+$. In addition, the main differences between the previous semantics described in [vdMF95] and the current semantics are listed. Finally, Section 6 mentions some possible future research areas.

Acknowledgement  We want to thank the following people for contributing to fruitful discussions about $\chi$ and for carefully reading preliminary versions of this report: J.C.M. Baeten, D.A. van Beek, G. Fábián, A.T. Hofkamp, S. Mauw, J.M. van de Mortel-Fronczak, M.A. Reniers, and J.E. Rooda.
1.1 Design decisions of $\chi$

In this section we list some design decisions made during the development of $\chi$. The decisions are requirements for the semantics we give in the following sections.

1. $\chi$ has one-to-one unidirectional communication: unidirectional communication channels connect the sender with the receiver.

2. Communication is synchronous: if one party wants to communicate, it has to wait until the other party is ready to communicate.

3. Synchronization channels allow two processes to synchronize. Synchronization is a special case of communication in which no information is communicated.

4. $\chi$ has global, continuous time.

5. Normal actions (assignments and skips) have higher priority than communications and time passing.

6. Communications have higher priority than time passing.

7. The longer a communication has been waiting, the higher its priority with respect to other communications.

In addition to the requirements listed above, the semantics should be defined compositionally. With a compositional definition we mean that every compound language construct should be described solely in terms of the semantics of its components. To put it differently, the meaning of a language construct should not depend on the context in which the construct occurs.

A reason why we need a compositional semantics is that it allows us to study components in isolation, verify properties of the components, and take advantage of the properties when studying systems that are built up from the components. If the meaning of the components depend on the context, it is not clear if the properties verified still hold for the components of the system.

Finally, the $\chi$ constructs have to be described directly without using many artificial constructs. This is one of the reasons why we did not describe the $\chi$ constructs by translating them into an existing formalism. However, as will become clear in the following sections, we cannot meet this requirement completely and need to define a few additional constructs.

1.2 Notations and conventions

In the remainder, we will use the following notations and conventions:

- **Expr**: The set of real expressions and boolean expressions with typical element $e$. If the types of expressions are relevant, we use subscripts, e.g., $e_r$ denotes a real expression and $e_b$ denotes a boolean expression. We will not define the syntax of expressions, but assume they are built up from constants, variables, and conventional operators like: $+, -, \cdot$ and $\land, \lor, \lnot$.

- **Var**: The set of variables, with typical elements $x, y, z$. We use subscripts to denote the type of variables: $x_r$ is a real variable and $y_b$ is a boolean variable.

- **R**: The real numbers, with typical elements $r$ and $j$. The sets $\mathbb{R}^+$ and $\mathbb{R}^{0,+}$ denote the positive real numbers and the nonnegative real numbers, respectively.

- **B**: The set of booleans true and false, with typical element $b$.

- **Value**: The union of the reals and the booleans: $\mathbb{R} \cup \mathbb{B}$. A typical element of Value is denoted by $c$ (constants).

- **Chan**: The set of channel names with typical elements $m, n$. 


We will only consider real and boolean expressions and leave it to the reader to include other \( \chi \) expressions, like naturals, integers, and lists (see [Ro096] for a description of \( \chi \) expressions). Furthermore, if it is not necessary to distinguish the type of an expression (real or boolean), we will drop the subscripts and just write \( e \) and \( x \). For example, in the syntax definition, see Table 1, we make no distinction between real-assignments and boolean-assignments and just write \( x := e \). We implicitly assume that the type of the variable \( x \) corresponds to the type of the expression \( e \).

2 Minimal \( \chi \)

The subset of discrete \( \chi \) we discuss in this report is called \( \chi^- \). A BNF definition of \( \chi^- \) is presented in Table 1.

It follows from the definition of the grammar that we do not need to define operator-priorities. For example, \( x := 2 ; y := x + 1 \parallel z := 3 \) can only be parsed as \( (x := 2 ; y := x + 1) \parallel z := 3 \). Consequently, we do not need to put parentheses in \( \chi^- \) statements. We assume \( [\, ]; \, \) and \( [\, ] \) are left-associative. In the remainder of this section, we will informally describe the semantics of \( \chi^- \).

\( PS \) is the class of parallel statements and \( S \) is the class of (sequential) statements. A parallel statement is built up from one or more sequential statements. Note that it is not allowed to use the parallel operator inside a sequential statement.

The first sequential statement is \texttt{skip}: the "do-nothing" statement. Actually, \texttt{skip} does something, since it performs the skip action. This action, however, does not change the current state.

The assignment \( x := e \) assigns the value of expression \( e \) to variable \( x \), under the assumption that the variable and the expression have the same type.

The class \( E \) with typical elements \( ev \) contains so-called events. Events are either communication statements (like \texttt{send: mle} or \texttt{receive: m?x}), or delta statements (like: \texttt{\Delta 2.5}). The delta statement explicitly describes the number of time units to wait. The communication statements can wait implicitly for a number of time units. Since \( \chi \) has synchronous communication, a statement that wants to send something to another statement has to wait until the receiving statement is ready (and vice versa).

We call \texttt{skip}, \( x := e, m!e, m?x \), and \( \Delta e_r \) \textit{simple statements}. Simple statements are not built up from operators and other statements. The other statements, the \textit{compound statements}, are built up from simple statements and the operators.

Sequential composition is expressed by means of the \( ; \) operator, for example, \( x := 1 ; x := x + 1 \) is a sequential composition of two assignments.

The class \( A \) describes lists of alternatives or \textit{selection statements}. A selection statement is either a \textit{guarded commands statement}: \[ e_{b,1} \Rightarrow s_1 \] \ldots \[ e_{b,n} \Rightarrow s_n \], where \( e_{b,i} \) are boolean expressions and \( s_i \) elements of \( S \); or a \textit{selective waiting statement}: \[ e_{b,1} ; e_{m,n} \Rightarrow s_1 \] \ldots \[ e_{b,n} ; e_{m,n} \Rightarrow s_n \], where \( e_{b,i} \) are boolean expressions, \( e_{m} \) are elements of \( E \), and \( s_i \) are elements of \( S \).

In a guarded commands statement, one of the statements \( s_i \) following a true guard \( e_{b,i} \) is chosen nondeterministically. If none of the guards is true, the guarded commands statement aborts (i.e., it leads to \textit{unsuccessful termination} or \textit{deadlock}).

In a selective waiting statement, each alternative starts with a boolean expression followed by an event. The boolean expressions are evaluated and then the selective waiting statement waits. As soon as one or more of the events following true guards can be executed, the selective waiting statement chooses one of them according to particular priority rules (see 3.4). After that, the selective waiting statement continues with the statement following the executed event.

Repetition in \( \chi \) is expressed by means of the \( * \) operator preceding a guarded commands statement or a selective waiting statement. The repetition ends (terminates successfully) if none of the guards is true. Note that this is different from non-repetitive guarded commands or selective waiting statements (see Example 2.1.1). Non-repetitive guarded commands or selective waitings abort (unsuccessful termination) if there is no true guard, whereas repetitions end (successful termination) if there is no true guard.
\[ PS ::= S \mid PS \parallel PS \]
\[ S ::= \text{skip} \mid x := e \mid E \mid A \mid \star A \mid S ; S \]
\[ E ::= m ! e \mid n ? x \mid \Delta e_r \]
\[ A ::= \{GC\} \mid \{SW\} \]
\[ GC ::= e_b \mapsto S \]
\[ \mid GC \parallel GC \]
\[ SW ::= e_b ; E \mapsto S \]
\[ \mid SW \parallel SW \]

<table>
<thead>
<tr>
<th>Table 1: The $\chi^-$ language</th>
</tr>
</thead>
</table>

2.1 Examples in $\chi^-$

In this section we provide some $\chi^-$ examples illustrating the main characteristics of the language. The examples are described informally and only provide some intuition about $\chi$. Some examples are accompanied with an annotated execution of a statement. The general form of an annotated execution is:

\[
E = \begin{cases} 
  p_{S_1} & \rightarrow \{ \text{comments}_1 \} \\
  p_{S_2} & \rightarrow \{ \text{comments}_2 \} \\
  p_{S_3} & \rightarrow \ldots
\end{cases}
\]

Here, each $p_{S_i}$ is a $\chi^-$ statement ($p_{S_i} \in PS$), and each comment$_i$ describes informally how execution of $p_{S_i}$ leads to $p_{S_{i+1}}$.

**Example 2.1.1 (Selection and repetition)** Repetitive selection statements behave differently from the non-repetitive selections. Take, for example, the guarded commands statement \[[e_b \mapsto x := x - 1]\]. It aborts if the value of $e_b$ is false. The reason for this is that the designers of the $\chi$ language considered this a specification error; the specification writer should have described all possible alternatives. In this case, the alternative $\neg e_b \mapsto \ldots$ should have been added to the selection statement, e.g., \[[e_b \mapsto x := x - 1 \parallel \neg e_b \mapsto \text{skip}]\]

In the context of a finite repetition, however, it is necessary to leave out some alternatives, since this is the only way to describe repetitions that eventually terminate. For example, the repetition $\star[e_b \mapsto x := x - 1]$ terminates whenever the value of $e_b$ is false, whereas the repetition $\star[e_b \mapsto x := x - 1 \parallel \neg e_b \mapsto \text{skip}]$ never terminates.

**Example 2.1.2 (Time passing)** The delta statement, $\Delta e_r$, denotes time passing. In $\chi$, time is global\(^1\) and continuous\(^2\), meaning that the set of time points form a continuum. Time can make progress if all concurrent statements are at a point where time passing is allowed, i.e., they are either about to execute a delta statement, or they are waiting to communicate.

\(^1\) Actually, in the previous description of the semantics of $\chi$, see [vdMF95], each process had its own local time. However, due to its synchronization behavior, this semantics did not allow one to exploit the local times of processes other than rather artificially. It appeared that processes with local times would be more complicated, without having real advantages for the specification writer.

\(^2\) A synonym for continuous time is dense time. Sometimes, it is even called real time; however, this term is also used in timing constraints for systems that have to meet certain deadlines imposed by their environments. Since these systems are an application domain of $\chi$, we will not use the term real time as a synonym for continuous time.
For example, the statement \( x := 3; \Delta 4 \| \Delta 1 \), which is parsed as \( (x := 3; \Delta 4) \| \Delta 1 \), first executes the assignment, then the \( \Delta 1 \) statement, and finally the (remainder of the) \( \Delta 4 \) statement:

\[
x := 3; \Delta 4 \| \Delta 1 \\
\rightarrow \{ \text{execute assignment} \}
\Delta 4 \| \Delta 1 \\
\rightarrow \{ \text{execute } \Delta 1 \text{ and change } \Delta 4 \}
\Delta 3 \\
\rightarrow \{ \text{execute } \Delta 3 \}
\]

**Example 2.1.3 (Communication)** \( \chi \) has one-to-one synchronous communication over unidirectional channels. Channels can be defined between a sender-statement and a receiver-statement. An actual communication takes place if a sender performs a send action on the channel and a receiver performs a receive action on the same channel. If one of the statements is not ready to communicate, the other statement is blocked. For example, in \( x := 1 ; m?x \| m!(y + 1) \), the second statement (the sender) has to wait until the first statement (the receiver) has executed the assignment.

**Example 2.1.4 (Priority)** skip statements and assignments have higher priority than communication statements; and communication statements have higher priority than delta statements. So, in \( \text{skip}; x := 3 \| m!3 \| m?y \), the skip statement and the assignment are executed before the communication statement:

\[
\text{skip}; x := 3 \| m!3 \| m?y \\
\rightarrow \{ \text{execute skip statement} \}
x := 3 \| m!3 \| m?y \\
\rightarrow \{ \text{execute assignment} \}
m!3 \| m?y \\
\rightarrow \{ \text{communication} \}
\]

**Example 2.1.5 (Priority)** If we prefix the assignment \( x := 3 \) of the previous example with a delta statement \( \Delta 3 \), we get: \( \text{skip}; \Delta 3 ; x := 3 \| m!3 \| m?y \). In this statement, the communication will take place before the assignment (but still after the skip statement):

\[
\text{skip}; \Delta 3 ; x := 3 \| m!3 \| m?y \\
\rightarrow \{ \text{execute skip statement} \}
\Delta 3 ; x := 3 \| m!3 \| m?y \\
\rightarrow \{ \text{communication} \}
\Delta 3 ; x := 3 \\
\rightarrow \{ \text{execute delta statement} \}
x := 3 \\
\rightarrow \{ \text{execute assignment} \}
\]

**Example 2.1.6 (Negative delta statements)** A delta statement can have a negative argument, e.g., \( \Delta -1.333 \). This means the statement is late; it should have been executed already. Therefore, such delta statements have even higher priority than skip statements and assignments. For example, in \( [ \text{true} ; m!3 \mapsto s_1 \| \text{true} ; \Delta -1 \mapsto s_2 ] \), the second alternative has higher priority and will therefore be chosen.
Note: Negative delta statements are considered specification errors and should be prevented. However, due to decisions made during the development of \( \chi \), we cannot treat negative delta statements similar as the deadlock statement \( \delta \). Negative delta statements should not allow the specification writer to encode priorities. Therefore we assume all negative delta statements have the same priority.

Example 2.1.7 (Longest waiting communication first) \( \chi \) has some sort of fairness principle: communications that waited longer have higher priority than communications that waited shorter. Consider the statement:

\[
\Delta_3; m!10 \parallel \Delta_4; \square \{ \text{wait } 2 \text{ time units} \}
\]

In the statement, first \( \Delta_3 \) is executed. During this time, the third statement, \( n!10 \), is waiting to communicate. After execution of \( \Delta_3 \), what is left of \( \Delta_4 \) is executed. During this time both the first and the third statement are waiting to communicate. Finally, the selective waiting statement can be executed. Both guards are true, so one of the alternatives could be chosen nondeterministically. However, since the third statement has waited longer than the first, communication over channel \( n \) is chosen. This means the second alternative of the selective waiting statements will always be chosen.

**Example 2.1.8 (Timeouts)** A timeout can be expressed by placing a delta statement as an event in a selective waiting statement. Consider, e.g., the boxed delta statement in:

\[
\square \{ \text{execute assignment} \}
\]

The \( \Delta_2 \) statement denotes the maximal time the selective waiting statement can wait: if the value of expression \( e_r \) in the second parallel statement is greater than 2, the timeout elapses and communication over \( n \) takes place. If the value of \( e_r \) is less than or equal to 2, the timeout does not elapse and communication over \( m \) takes place.

Note that timeouts are different from normal delta statements. For example, if the value of \( e_r \) is 2, we get the following execution:
As we see, the timeout reduces to \( \Delta 0 \), whereas the delta statement disappears. The timeout cannot behave similarly to the delta statement, i.e., disappear, since that would mean the second alternative of the selective waiting is chosen, which is not the behavior the designers of \( \chi \) had in mind.

A drawback of the difference between delta statements and timeouts is that syntactically they are the same; only the context of the statement determines whether it behaves as a delta statement or as a timeout. Since we would like to define the semantics compositionally (meaning that every composite construct is defined in terms of its components), we introduce a new symbol to denote timeouts, see next section.

3 \( \chi^+ \): A generalized version of \( \chi^- \)

In this section we will define another language, called \( \chi^+ \), which will be used to describe the formal semantics of \( \chi^- \). \( \chi^+ \) is a generalized version of \( \chi^- \), in which some syntactic constraints enforced by the grammar of \( \chi^- \) are ignored. In addition, the syntax of some statements is extended in order to include information needed to define the formal semantics. \( \chi^+ \) is designed in such a way that:

1. The informal meaning of \( \chi^- \) specifications coincides with the formal meaning of their corresponding \( \chi^+ \) specifications.
2. The translation from \( \chi^- \) to \( \chi^+ \) is straightforward.

The main reason for not describing the formal semantics of \( \chi^- \) directly is that \( \chi^- \) has two complicated \( n \)-ary operators: the guarded commands operator and the selective waiting operator. The definition of the behavior of these operators would be too complex for our goal: a clear operational semantics of \( \chi \).

Another reason is that the behavior of some \( \chi^- \) operators is context sensitive (e.g., see Examples 2.1.7 and 2.1.8) and therefore, it is not possible to provide a compositional semantics for \( \chi^- \).

In Section 3.1, we define the syntax of \( \chi^+ \). In Section 3.2 we will define the translation from \( \chi^- \) to \( \chi^+ \). This will probably give the reader some intuition about the \( \chi^+ \) language. The formal semantics of \( \chi^+ \) is given in Section 3.3. To keep things simple, we will not include a formal treatment of \( \chi^- \)'s priority rules in Section 3.3, but postpone it to Section 3.4. Section 3.5 describes some examples in \( \chi^+ \). Finally, Section 3.6 discusses some of the differences between \( \chi^+ \) and \( \chi^- \).

3.1 Syntax of \( \chi^+ \)

The syntax of \( \chi^+ \) is given in Table 2. The first line resembles the basic statements of \( \chi^- \). Except for being written differently a bit, the send and receive statement have an additional parameter \( j \) (a real constant, see Section 1.2). This parameter denotes the time the concerning statement has been waiting. For example, \( !(m, 3.2, e) \) is the \( \chi^+ \) variant of the send statement \( m!e \) that has been waiting 3.2 time units.
The second line introduces four additional simple statements. $\epsilon$ is the empty statement. The empty statement cannot do an action, but it can terminate successfully. It is different from the skip, since skip can perform the skip action (see below).

The second statement, $e_b$, is a boolean expression. Depending on its value, true or false, it can either terminate successfully, or it deadlocks. So, true, viewed as a statement, is exactly the same as $\epsilon$ and false is the same as the deadlock statement $\delta$. As we will see below, using boolean expressions as statements in this way simplifies the semantics of guarded commands and selective waiting.

The empty statement is a timeout statement; its behavior is similar to the delta statement $\Delta e_r$, except that in some situations it can perform a timeout action (see below).

The third up to the seventh line define the operators of $\chi^+$. The grammar is ambiguous and we have to use parentheses to disambiguate the syntax. To reduce the number of parentheses, we assume the operators have different binding-priorities. In Table 2, the operators are listed in order of decreasing binding-priority. That is, the $*$ operator has the highest priority and the $\|$ operator has the lowest priority. For example,

$$x := 1 ; y := x + 1 \mid \neg b \mapsto !((m,0,0) \mid b \mapsto ?(n,0,y); z := y + 3 \mid \delta !((n,0,1); x := 3$$

should be read as:

$$(x := 1 ; y := x + 1) \mid (\neg b \mapsto !((m,0,0) \mid b \mapsto ?(n,0,y); z := y + 3)) \mid (\delta !((n,0,1); x := 3)$$

The final line of Table 2 shows that we can use parentheses to group statements.

The repetition operator $*$ executes its operand repeatedly. If the operand cannot perform an action, the repetition operator can terminate successfully. This means that both $*\epsilon$ and $*\delta$ terminate successfully.

The sequential operator $;$ is known from $\chi^-$. If $p$ and $q$ are $\chi^+$ statements, then $p ; q$ is the sequential composition of $p$ and $q$. This statement first behaves as $p$ and then as $q$, provided $p$ terminates successfully.

The select operator $\Rightarrow$ is a kind of sequential composition operator that inserts an extra action: $p \Rightarrow q$ first behaves as $p$ and then, provided that $p$ terminates successfully, it executes an extra action, and finally it behaves as $q$. The extra action is called the select action. In the context of a choice operator (see next paragraph), the select action can determine a choice.

---

$$P ::= \text{skip} \mid x := e \mid !((m,j,e) \mid ?(m,j,x) \mid \Delta e_r$$

$$\mid \epsilon \mid \delta \mid e_b \mid \Theta e_r$$

$$\mid *P$$

$$\mid P ; P$$

$$\mid P \Rightarrow P$$

$$\mid P \parallel P$$

$$\mid (P)$$

---

Table 2: Syntax of $\chi^+$

The choice operator $\parallel$ and the select operator cooperate in the following way. The statement $p_0 \Rightarrow p_1 \parallel q_0 \Rightarrow q_1$ first behaves as the interleaving of $p_0$ and $q_0$. That is, it executes actions of both $p_0$ and $q_0$ in arbitrary order. Suppose $p_0$ terminates before $q_0$ has terminated. Then the $\Rightarrow$ operator following $p_0$ executes the select action which means the $p_0$-alternative of the $\parallel$ operator is chosen. Subsequently, the actions of $p_1$ will be executed. Similarly, the $q_0$ alternative will be chosen if $q_0$ terminates before $p_0$ has terminated. This cooperation enables us to use the
select operator and the choice operator to define the guarded commands and the selective waiting statements of $\chi^-$. If the select operator occurs outside an operand of the choice operator, it still can execute the select action but it will not make a choice (so, this makes the select operator rather useless outside an operand of a choice operator). If the operands of a choice operator do not contain a select operator, the choice operator describes all possible interleavings of the actions of its operands.

The parallel composition operator $\parallel$ interleaves the actions of its operands. In addition, if one of its operands can execute a send action and the other can execute a receive action (on the same channel), then the parallel composition can execute a communication action. This distinguishes the parallel operator from the choice operator with operands not containing select operators.

### 3.2 Translation from $\chi^-$ to $\chi^+$

Table 3 defines the translation from $\chi^-$ to $\chi^+$. The first column shows the various $\chi^-$ constructs. The second column shows their translations. The translations of compound constructs are given in terms of $\chi^+$ operators and translations of the components of the $\chi^-$ construct. For example, $*p$ is translated into $*p'$, in which $p'$ is the translation of $p$.

<table>
<thead>
<tr>
<th>$\chi^-$ statement</th>
<th>$\chi^+$ statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>skip</td>
</tr>
<tr>
<td>$x := e$</td>
<td>$x := e$</td>
</tr>
<tr>
<td>$m!e$</td>
<td>$\langle m, 0, e \rangle$</td>
</tr>
<tr>
<td>$m?x$</td>
<td>$\langle m, 0, x \rangle$</td>
</tr>
<tr>
<td>$\Delta e_r$</td>
<td>$\Delta e_r$</td>
</tr>
<tr>
<td>$\Theta e_r$</td>
<td>$\Theta e_r$</td>
</tr>
<tr>
<td>$p; q$</td>
<td>$p'; q'$</td>
</tr>
<tr>
<td>$b_1 \mapsto s_1$</td>
<td>$b_1 \mapsto s'_1$</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>$b_n \mapsto s_n$</td>
<td>$b_n \mapsto s'_n$</td>
</tr>
<tr>
<td>$b_1 ; ev_1 \mapsto s_1$</td>
<td>$b_1 ; ev'_1 \mapsto s'_1$</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>$b_n ; ev_n \mapsto s_n$</td>
<td>$b_n ; ev'_n \mapsto s'_n$</td>
</tr>
<tr>
<td>$*p$</td>
<td>$*p'$</td>
</tr>
<tr>
<td>$p \parallel q$</td>
<td>$p' \parallel q'$</td>
</tr>
</tbody>
</table>

**Table 3: Translation from $\chi^-$ to $\chi^+$**

**Note** Even though it is not directly clear from Table 3, the translation from $\chi^-$ into $\chi^+$ can be automated. The only difficulties are the translations of $\Delta e_r$ statements; if it is a timeout it should be translated into $\Theta e_r$ and if not, it should be translated into $\Delta e_r$. However, it is possible to define the syntax of the $\chi^-$ language with a BNF-grammar that distinguishes timeouts from "normal" delta statements. Given such a grammar, the automatic translation is straightforward.

The most interesting part of the translations concerns the guarded commands and the selective waitings. These constructs are translated using the sequential composition operator, the select operator, and the choice operator from $\chi^+$. In addition, delta statements that behave as timeouts in $\chi^-$ are translated into a timeout statements.

To illustrate the translation, we give an example. Consider the $\chi^-$ statement from Example 2.1.8:

\[
[ \text{true} ; m!10 \mapsto s_1 \mid \text{true} ; \langle 2 \rangle \mapsto n!1 ] \parallel \Delta e_r ; m?x \parallel n?y
\]
In the statement, the boxed delta statement is a timeout. So, it should be translated differently from the other delta statement, \( \Delta e_r \). The translation of the complete statement reads:

\[
(true; !(m,0,10) \rightarrow s'_1 \parallel true; \Theta 2 \rightarrow !(n,0,1)) \parallel \Delta e_r; ?(m,0,x) \parallel ?(n,0,y)
\]

Example 3.5.2 on page 20 explains the difference between timeout statements and delta statements.

### 3.3 Operational semantics of \( \chi^+ \)

The semantics of \( \chi^+ \) statements depends on the value of the variables occurring in the statements. The value of variables is described by variable valuations:

**Definition 1 (Variable valuation)** A correctly typed function \( v : \text{Var} \rightarrow \text{Value} \) is a variable valuation. The set of all variable valuations is called \( \text{Val} \).

The set \( \text{Value} \) is the union of \( \mathcal{R} \) and \( \mathcal{B} \) (see page 3) and typical elements of \( \text{Value} \) are denoted by \( c \) (constants). If \( v \) is a variable valuation, then \( v[x\leftarrow c] \) is the same variable valuation overridden at \( x \) with value \( c \), assuming \( x \) and \( c \) have the same type. A concrete variable valuation will be written as a sequence of variable-value pairs separated by commas and enclosed in square brackets: \([x=13, y=true, x_2=10.71] \]

**Definition 2 (\( \chi^+ \) process)** A \( \chi^+ \) process is a tuple \( (p,v) \) where \( p \in P \) is a \( \chi^+ \) statement, and \( v \in \text{Val} \) a variable valuation.

The operational semantics of \( \chi^+ \) defines if a process may terminate successfully and if it may perform (or execute) an action. We will use a special notation to define the semantics. This notation, which is due to Plotkin [Pl0811], is called structured operational semantics (sos) and has sos-rules of the form:

\[
P_1, P_2, \ldots, P_n \quad \frac{}{C}
\]

Here, the \( P_i \) and \( C \) are predicates. The \( P_i \) are called premises and \( C \) is the conclusion; if all \( P_i \) hold (are true), then \( C \) holds as well.

In the actual sos-rules defined below, we assume the existence of an evaluation function for expressions: \( \text{eval} : \text{Expr} \times \text{Val} \rightarrow \text{Value} \). The function \( \text{eval} \) can be defined using sos-rules, but that would be outside the scope of this document\(^3\).

**Successful termination**

Table 4 defines the successful termination predicate on processes. If a process \( (p,v) \) terminates successfully, it has reached an accepting final state. This is denoted with the postfix predicate \( \downarrow \): \( (p,v) \downarrow \).

The first rule in Table 4 says that the empty process, \( (e,v) \), can terminate successfully. As expected, there is no termination rule for the deadlock process, \( (\delta,v) \).

The second rule defines whether a guard may terminate successfully. In this rule we use the function \( \text{eval} \) to determine the value of the boolean expression \( e_b \).

The third, fourth, and fifth rule define when a process built up from a binary operator can terminate. It follows from the table that all binary operators are equal with respect to successful-termination behavior.

The sixth rule defines when a repetition can terminate successfully. This rule has a negative premise \( (p,v) \rightarrow \) saying that \( (p,v) \) cannot execute an action. Below we define the ternary predicate \( \rightarrow \). The notation \( (p,v) \rightarrow \) is an abbreviation for \( \neg (\exists a, (p',v') : (p,v) \xrightarrow{\Delta} (p'v')) \).

---

\(^3\) Actually, we have defined the eval function in the ASF+SDF version of the sos-rules of \( \chi^+ \), see Section 4.
The repetition operator terminates if its argument cannot do an action. As described in Example 2.1.1, a repetition of a guarded commands statement or a repetition of a selective waiting statement terminates successfully if all guards are false. If we translate this into $\chi^{1}$-terms, we see that the repetition operator has to terminate successfully whenever its argument cannot do an action. This explains Rule 6.

\[
\begin{align*}
(1) & \quad (e, v) \downarrow \\
(2) & \quad \text{eval}(e, v) \downarrow \\
(3) & \quad (p, v) \downarrow, (q, v) \downarrow \\
(4) & \quad (p, v) \downarrow, (q, v) \downarrow \\
(5) & \quad (p, v) \downarrow, (q, v) \downarrow \\
(6) & \quad (p, v) \rightarrow \\
(7) & \quad (*p, v) \downarrow
\end{align*}
\]

Table 4: Successful termination

Executing actions

We will now describe what happens if a process executes (or performs) an action. If a process performs (or executes) an action, it becomes a new process of which both the statement and the variable valuation may be different. This is denoted by $(p, v) \xrightarrow{a} (p', v')$, where $p$ and $p'$ are statements, $v$ and $v'$ are variable valuations, and $a$ is an action. Before we give the sos-rules defining the ternary predicate $\rightarrow$, we first have to define the set of possible actions.

Note Some actions are syntactically very similar to particular statements. However, actions and statements should not be confused. Processes are built up from statements (and variable valuations) and can execute actions.

**Definition 3 (Atomic actions)** Let $x \in \text{Var}, m \in \text{Chan}, c \in \text{Value}, r \in \mathbb{R}, j \in \mathbb{R}^{0,+}$, and let $\Theta$ and $\omega$ be two new different symbols not occurring in $\text{Var}$, $\text{Chan}$, and $\text{Expr}$. The set of all atomic actions is called $\text{AC}$ and contains:

- $x := c$ Assignments
- $!(m, j, c)$ Send actions
- $?!(m, j, x)$ Receive actions
- $c(m, j, x, c)$ Communication actions
- $r$ Time passing actions
- $\Theta$ Timeout action
- $\omega$ The selection action

The assignment action $x := c$ can be executed by an assignment statement $x := e$ if the value of the expression $e$ equals $c$. Note that no expressions occur in assignment actions, but only constant values.

Similarly, the send action $!(m, j, c)$ can be executed by a send statement $!(m, j, e)$ if the value of the expression $e$ equals $c$. Just like assignment actions, send actions do not contain expressions.

---

1The reason for the symbol $\omega$ is that it is executed by the select operator $\Rightarrow$ (an arrow) and the Greek word for arrow is $\alpha\iota\sigma\tau\rho\omicron$ (alpha).
The receive action \((m,j,x)\) can be executed by a receive statement \((m,j,x)\). The result of
the receive action \((m,j,x)\) is that \(x\) is assigned a value. However, since we do not know this
value, the receive action does not change \(x\). Still, the receive actions are useful, since, together
with send actions, they enable communication actions.

The communication action \(c(m,j,x,e)\) is a combination of a send action \(l(m,j_1,e)\) and a
receive action \((m,j_2,x)\), where \(j_1 \max j_2 = j\). That is, if the send action and the receive action
are possible at the same time, we also have a third option, namely a communication action. In such
situations, we say the send action and the receive action match. The time this communication
action has been waiting is, per definition, the maximum of the waiting times of the send and
receive actions.

Time passing actions are denoted by a real number \(r\). If \(r \in \mathbb{R}^+\), the time passing action can
be executed by \(\Delta e_r\) and \(\Theta e_r\) statements provided that the value of \(e_r\) is greater than or equal to \(r\).
Furthermore, if \(r \in \mathbb{R}^+\), send statements \((m,j,e)\) and receive statements \((m,j,x)\) can execute
time passing actions \(r\) as well. We will call positive time passing actions \(r \in \mathbb{R}^+\) delay actions.
By executing delay actions, send statements and receive statements can wait for each other. Below
we will see that they will not wait longer than needed, i.e., as soon as matching send and receive
actions are enabled, the corresponding send and receive statements cannot execute delay actions
anymore. If \(r \notin \mathbb{R}^+\), i.e., \(r \leq 0\), only \(\Delta e_r\) and \(\Theta e_r\) statements with \(e_r\) equal to \(r\) can execute the
action.

If \(r < 0\), then \(r\) denotes a negative time passing action. This action can only be executed
by \(\Delta e_r\) statements and \(\Theta e_r\) statements if the value of \(e_r\) equals \(r\). The developers of \(\chi\) consider
the occurrence of negative time passing actions specification errors. Therefore, it should not
be possible to take advantage of the behavior of negative time steps. Later, we will see some
consequences of this attitude towards negative time steps.

The timeout action \(\Theta\) can be performed by a timeout statement \(\Theta e_r\) provided the value of \(e_r\)
equals 0. The situation where \(e_r\) is 0 distinguishes the \(\Delta e_r\) statement and the \(\Theta e_r\) statement, since
\(\Delta e_r\) can only execute the zero-length time step 0, whereas the \(\Theta e_r\) statement can only execute
the timeout action \(\Theta\). If \(e_r\) is not equal to 0, the statements can execute exactly the same actions.

Simple processes

The rules for simple processes (processes with simple statements) are given in Table 5. The skip
statement can perform the skip action. The skip action does nothing, i.e., it does not change the
variable valuation \(v\).

The assignment statement can execute the corresponding assignment action. Note that the
value of the expression \(e\) has to be determined in order to perform the assignment action.

There are two rules for delta statements: rule 9 and rule 10. The first rule is applicable if the
time passing action has the same size as the delta statement. The second rule is applicable if these
sizes are different. Note that rule 10 only describes positive time passing actions. Therefore, if the
value of \(e_r\) in \(\Delta e_r\) is not positive, the statement can only perform the action described by rule 9.

Rules 11, 12, and 13 describe the timeout statement. If the value of \(e_r\) in \(\Theta e_r\) is negative, a
negative time passing action can be performed. If \(e_r = 0\), the timeout action can be performed. In
Section 3.4 we will see the influence of the \(\Theta\) action. Finally, if the value of \(e_r\) is greater than zero,
the timeout statement can perform positive time passing actions. The size of the actions should be
less than or equal to the size of \(e_r\). If the sizes are equal, the result will be the \(\Theta 0\) statement and
not \(e_r\), as could be expected. The \(\Theta 0\) statement can then perform the timeout action \(\Theta\). Therefore,
positive timeout statements will always perform the \(\Theta\) action before they terminate.

Rules 14 and 15 show that send and receive statements can wait, i.e., perform arbitrary positive
time passing actions. This is necessary, since communication in \(\chi\) is synchronous which means
that processes have to wait for each other if they want to communicate.

The last two rules, rules 16 and 17, describe the send and receive actions that can be executed
by their corresponding statements. Similar to the assignment, the value of the expression \(e\) in
a send statement \((m,j,e)\) has to be determined, in order to determine the corresponding send
action.

13
Sequential processes

The rules for sequential processes are given in Table 6. There are two rules for sequential composition, Rule 18 and 19. According to Rule 18, the sequential composition can perform the same actions as its left operand can. In addition, if the left operand can terminate successfully, the actions of the right operand are also allowed (Rule 19).

The actions that can be performed by the choice process are described by Rules 20-26. If none of the operands of the \(\sim\) operator can perform the select action \(o\) and one of the operands can perform an action that is not a positive time step \((r \in \mathbb{R}^+)\), the choice statement can perform the action too (Rules 20, and 21). The result is that the operand performing the action is changed and the other operand is left unchanged. Similarly, if one of the operands can perform the select action, the choice statement can perform it as well (Rules 22 and 23). However, now a choice is actually made and the result is determined by the operand that executed the select action.

Rules 24-26 describe when the choice statement can perform a positive time passing action. This is only possible if both its operands can perform the same positive time step, or if one of the operands can perform a time step and the other cannot perform any action at all.

Rule 27 describes when the select operator can execute the select action. This is allowed if the first operand of the select operator can terminate successfully. According to Rule 28, the select-process can execute an action if its left operand can execute the action.

Finally, Rule 29 shows that the repetition process can perform any action its operand can. The resulting statement is the sequential composition of the result of its operand performing the particular action and the original repetition statement.

Parallel processes

A parallel process is built up from two processes and the parallel operator \(\|\). Table 7 defines the sos-rules for parallel processes. If an operand can perform an action that is not a positive time step, the parallel composition can perform the action as well (Rules 30 and 31). In addition, if one of the operands can perform a send action \(!(m, j_1, e)\) and the other operand can perform a receive
action ?(m, j, x), then the parallel process can perform a communication action c(m, j, x, e) where 

\[ j = j_1 \max j_2 \]

Finally, Rules 34–36 define when the parallel composition can execute positive time steps. This is allowed in the same situations as for the choice operator (see Rules 24–26).

An interesting observation of the sos-rules given above is that a \( \chi^+ \) process never has both the option to terminate and the option to perform an action at the same time. That is, if a process can terminate successfully, it has reached a state in which there are no actions left to be executed. Similarly: if a process can execute actions, it does not have the option to terminate successfully.

The following theorem describes this property of \( \chi^+ \) processes.

**Theorem 1** For all \( \chi^+ \) processes \((p, v)\) we have:

1. \((p, v) \xrightarrow{\alpha} (p', v') \Rightarrow \neg (p, v) \downarrow\)
2. \((p, v) \downarrow \Rightarrow (p, v) \Rightarrow \)

**Proof** We will only give a proof of property 1. Suppose \((p, v) \xrightarrow{\alpha} (p', v')\). We have to show that \(\neg (p, v) \downarrow\). If \((p, v)\) can terminate, some of the Rules 1–6 have to be applicable. Since \((p, v)\) can perform an action, \(p\) cannot be the empty statement \(\varepsilon\) or a guard \(g_0\). Therefore, Rules 1 and 2 cannot be applicable. Furthermore, Rule 6 cannot be applicable, since its premise requires that \((p, v)\) cannot execute an action.

This leaves us to prove that if \((p, v)\) can execute an action, it cannot terminate according to one of the Rules 3–5. Since these cases are similar, we will only prove the second one. For Rule 4 to
Table 7: Parallel processes

be applicable the \( (p, v) \) process has to be of the form \( (p_1 \parallel p_2, v) \). Moreover, \( (p_1, v) \downarrow \land (p_2, v) \downarrow \) must hold. We will now show that this cannot be true.

If we look at the sos-rules describing when a choice process can perform an action (Rules 20–26) we can conclude that \( (p_1, v) \) can only perform an action if \( (p_2, v) \) can.

So, we know that there exists an \( a' \in AC \), such that \( (p_1, v) \rightarrow (p_1', v') \) or \( (p_2, v) \rightarrow (p_2', v') \). Now, by induction, we find that \( ((1', v) . j.) \) or \( ((1'2, v). j. ) \). Therefore, Rule 4 cannot be applicable.

The following theorem shows that time passing only occurs if no other actions are possible except for send or receive actions. Therefore, progress of time cannot influence the number of actions a process can execute. This is called strong time factorization (see [Ver97], page 43).

**Theorem 2** Consider a \( \chi^+ \) process \((p, v)\) and a delay action \( \tau \in R^+ \). If \((p, v) \xrightarrow{\tau} (p', v')\), then

\[
\forall a \in AC \ ( (p, v) \xrightarrow{a} ) \Rightarrow a \in R^+ \cup \{ (m, j, c), ?(m, j, x) \}
\]

**Proof** If \((p, v)\) can execute a positive time step, some of Rules 9, 10, 13, 14, 15, 18, 19, 24, 25, 26, 28, 29, 34, 35, and 36 have to be applicable.

If Rules 9, 10 and 13, are applicable to process \((p, v)\), then it is clear that the process cannot execute actions other than positive time steps.

Suppose Rules 14 or 15 are applicable. This means \( p \) is a send or receive statement. Besides delay actions, send and receive statements can only execute send and receive actions. So, these two rules do not contradict the theorem.

Suppose Rules 18 or 19 are applicable to \((p, v)\) and \((p, v)\) can execute a delay action. This means that one of the operands of the ; operator can execute a delay action. By induction, it is clear that the concerning operand cannot execute an action other than a delay action, a send action, or a receive action. Therefore, the sequential composition \((p, v)\) cannot execute an action other than a delay action, a send action, or receive action.

Rules 24, 25, and 26 describe when a choice process can perform delay actions. If some of these rules are applicable to \((p, v)\), then either both of the operands can perform a delay action, or one
of the operands can and the other one cannot perform an action at all. In both cases, by induction we see that the operands that can perform a delay action can only perform delay actions, send actions, or receive actions. Consequently, the choice process can only execute delay actions, send actions, or receive actions.

Rule 28 enables a select process, built up from a select statement, to execute any action its left operand can execute. By induction, we see that if the select process can execute a delay action, then the left operand can only execute delay actions, send actions, or receive actions. Therefore, the select process itself can only execute delay actions, send actions, or receive actions.

If Rule 29 is applicable to \((p, v)\), it means \(p\) is if the form \(\ast p'\). The process \((\ast p', v)\) can execute the same actions as \((p', v)\) can. By induction, we see that if \((p', v)\) can execute a delay action, it can only execute delay actions, send actions, or receive actions. Therefore, the repetition process \((\ast p', v)\) can only execute delay actions, send actions, or receive actions.

Finally, suppose some of Rules 34, 35, and 36 are applicable to \((p, v)\). By induction and similar reasoning as for the \(\|\) operator, we see that the operands of the \(\|\) operator can only execute delay actions, send actions, or receive actions. For the last two Rules, 35 and 36, we can conclude that the parallel composition can only execute delay actions, send actions, or receive actions. Rule 34 allows both of the operands to perform send and receive actions. If there are matching send and receive actions, this could result in a communication action. However, in that case Rule 34 is not applicable (due to the premise \((p \parallel q, v) \not\in (m, d, c)\)). Therefore, the parallel composition can only execute delay actions, send actions, or receive actions.

The following theorem shows that if a choice-process can execute a select action, it cannot do another action. This ensures that as soon as a choice can be made, it will be made. We will only sketch the proof.

**Theorem 3** If a \(\chi^+\) process \((p \parallel q, v)\) can perform a select action \(\sigma\), i.e., \((p \parallel q, v) \xrightarrow{\sigma} (p', v')\), then it cannot perform another action.

**Proof** Rules 20–26 describe when a choice process can perform an action. We are only interested in the rules concerning non-select actions: Rules 20, 21, 24, 25, and 26. The first two show that a choice process can perform a non-select action if one of its operands can perform the action and the other operand cannot perform a select action. By induction, we see that the operand performing the non-select action cannot perform the select action.

Rules 24, 25 and 26 describe when a choice process can execute a delay action. However, Theorem 2 says that a process can only execute a positive time step if it cannot execute any other actions, except send and receive actions. So, if a process can execute a select action, it cannot execute a delay action. Therefore, these rules cannot be applicable.

The following theorem shows that if a process can execute a communication action, it must have subprocesses that can execute arbitrary delay actions. Intuitively this should be clear, since a process that can execute a communication action, can execute matching send and receive actions, too. The only statements that can execute send and receive actions are send and receive statements and these statements can execute arbitrary delay actions as well. We will not give a formal proof of this theorem, since that would need a formalization of subprocesses which is outside the scope of this report.

**Theorem 4** If a \(\chi^+\) process \((p, v)\) can execute a send action or a receive action, then there must be a subprocess \((p', v)\) of \((p, v)\) that can execute the action as well and \((p', v)\) can execute arbitrary positive time steps.

### 3.4 Priority in \(\chi^+\)

In this section we define a priority relation on the atomic actions of \(\chi^+\). The relation is based on the following priority rules of \(\chi\) (see also Section 1.1 about the \(\chi\) design decisions):
1. Normal actions (assignments and skips) have higher priority than communication actions and time passing actions.

2. Communication actions have higher priority than time passing actions.

3. The longest waiting communication actions have higher priority than other communication actions.

4. Negative time steps have higher priority than normal actions, communication actions, and time passing actions.

The rules can directly be defined for $\chi^+$ actions, but we also have to take into account the translation from $\chi^-$ to $\chi^+$ (see Table 3). In addition, some $\chi^+$ actions do not exist in $\chi^-$. First of all, we have to decide about the priority of the select action $o$. Since this action does not exist in $\chi^-$, it is not captured by any of the rules mentioned above. Theorem 3 says that in the context of a choice operator, the select action will always be executed whenever it is enabled. Therefore, the sos-rules already define the priority of the select action and we do not have to include the select action in the priority relation.

Second, we should decide what to do with negative time steps. One option would be to assign higher priorities to more negative time steps. However, negative time steps are considered specification errors and should be prevented by the specification writer. Furthermore, we do not want the specification writer to be able to take advantage of negative time steps, e.g., by encoding priorities with negative time steps. Therefore, each negative time has the same priority.

The zero-length time step $0$ has the same priority as the assignments and the skip action. The reason for this is that we want it to behave as a skip action.

Next come the actions: $!(m, j, c)$, $?!(m, j, x)$, and $c(m, j, x, c)$. These actions have lower priority than assignments, skips, and zero-length time steps. Furthermore, the longer a communication action has been waiting, the higher its priority with respect to other communication actions (see item 2 above). Now it becomes clear why we needed the parameter $j$ in the $\chi^+$ syntax of send and receive statements: it denotes the waiting time of the action, which is used to determine the priorities between different communication actions.

Due to Theorem 4, we know that a process $(p, v)$ that can execute a send or a receive action, has a subprocess that can execute arbitrary delay actions. We cannot assign send and receive action higher priorities than positive time steps, for that would mean the send and receive statements will never execute time steps. On the other hand, if time passing gets higher priority than send and receive actions, the latter will be delayed as long as possible. Consequently, send and receive actions will never be executed. However, usually we are not interested in send and receive actions, but in communication actions (i.e., combined send and receive actions). We assign communication actions higher priority than delay actions. Therefore, if matching send and receive actions are enabled at the same time, the corresponding communication will be performed without delay. Summarizing this paragraph: send and receive actions get lower priority than positive time steps and communication actions have higher priority than positive time steps.

The timeout $\Theta$ action has lower priority than communication actions, but higher priority than positive time steps. If a communication action is possible and, at the same time, a timeout action is possible, the communication action will be executed.

It follows that time passing has the lowest priority. However, the sos-rules already ensure that no positive time step will be executed if an action that is not a send or receive action is enabled, see Theorem 2. Therefore, the priority relation does not have to define an order between positive time steps and other actions. Still, the sos-rules do not say anything about different priorities for different positive time steps. This is a problem, since a process could end up performing infinitely many smaller and smaller time steps, without really making progress. This behavior is called Zeno behavior and we prevent this by assigning higher priorities to greater positive time steps. Note that it is possible to explicitly specify a process that exhibits Zeno behavior, see Example 3.5.3.
Notation Before we define the priority relation on $\chi^+$ actions formally, we first define two subsets of $\mathcal{A}C$.

$\text{SA}$ The set of simple actions, containing the assignment actions $x := c$, the skip action skip, and the zero-length time-step $0$.

$\text{SR}$ The set of send and receive actions, i.e., the set of all $!(m,j,c)$ and $?(m,j,x)$.

Table 8 defines the priority relation on actions. The relation is denoted by $a \prec a'$ and should be read as: if $a \prec a'$ then action $a$ has higher priority than action $a'$.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r^- \prec \text{sa}$</td>
<td>Negative time steps have higher priority than simple actions</td>
</tr>
<tr>
<td>$\text{sa} \prec c(m,j,x,c)$</td>
<td>Simple actions have higher priority than communications</td>
</tr>
<tr>
<td>$c(m_1,j_1,x_1,c_1) \prec c(m_2,j_2,x_2,c_2)$ if $j_1 &gt; j_2$</td>
<td>The longer a communication waited, the higher its priority.</td>
</tr>
<tr>
<td>$c(m,j,x,c) \prec \Theta$</td>
<td>Communications have higher priority than timeouts</td>
</tr>
<tr>
<td>$\Theta \prec r^+$</td>
<td>The timeout action has higher priority than delay actions</td>
</tr>
<tr>
<td>$r_1^+ \prec r_2^+$ if $r_1^+ &gt; r_2^+$</td>
<td>Greater positive time steps have higher priority than smaller positive time steps</td>
</tr>
<tr>
<td>$r^+ \prec \text{sr}$</td>
<td>Time passing has higher priority than send and receive actions</td>
</tr>
<tr>
<td>$a_1 \prec a_3$ if $a_1 \prec a_2 \land a_2 \prec a_3$</td>
<td>Transitivity of $\prec$</td>
</tr>
</tbody>
</table>

Where $r^- \in \mathbb{R}^-$, $\text{sa} \in \text{SA}$, $r^+, r_1^+, r_2^+ \in \mathbb{R}^+$, $\text{sr} \in \text{SR}$, and $a_1, a_2, a_3 \in \mathcal{A}C$

Table 8: Priority relation on atomic actions

Definition 4 (Priority transition relation) Let $a$ and $b$ be arbitrary actions and let $(p,v)$ and $(p',v')$ be $\chi^+$ processes. The priority transition relation $\rightarrow$ is defined by Rule 37 given below. The notation $(p,v) \rightarrow (p',v')$ is an abbreviation for $\exists p',v' (p,v) \rightarrow (p',v')$.

$$(37) \quad (p,v) \xrightarrow{a} (p',v'), \ (\forall b ((p,v) \xrightarrow{b}) \Rightarrow \neg(b \prec a)) \Rightarrow (p,v) \xrightarrow{a} (p',v')$$

3.5 Examples in $\chi^+$

In this section we will give some examples of $\chi^+$ processes and their executions. The style in which we denote the executions is similar to the examples in Section 2.1. However, we can now be more formal: if process $(p,v) \xrightarrow{a} (p',v')$ and $(p',v') \xrightarrow{a'} (p'',v'')$, we write:

$(p,v)$

$\xrightarrow{a}\{\text{A brief explanation for this transition}\}$

$(p',v')$

$\xrightarrow{a'}\{\text{A brief explanation for this transition}\}$

$(p'',v'')$

$\downarrow\{\text{Successful termination of the process}\}$
If we are interested in the priority transition relation, we use the \( \rightarrow \) arrow defined by Rule 37. Here is a concrete example of the execution of the process \((x := x + 1; \text{skip}, [x = 0])\):

\[
\begin{align*}
\text{\(x := 3\)} & \quad \text{\{Value of expression \(x + 1\) for valuation \([x = 2]\) is 3\}} \\
\text{\((x = 3)\)} & \quad \text{\{According to Rules 19 and 7\}} \\
\text{\((x, [x = 3])\)} & \quad \downarrow
\end{align*}
\]

The execution of this process is completely deterministic. Therefore, the result would be the same if we had used the priority transition relation \(\Rightarrow\).

**Example 3.5.1 (Guarded commands)** The process \(((x > 0 \mapsto x := x - 3 \parallel x \geq 0 \mapsto x := x - 5 \parallel x < 0 \mapsto \text{skip}), [x = 5])\) has three alternatives. The first two alternatives overlap, since both guards are true if \(x > 0\). Consequently, the process has two possible traces under the \(\rightarrow\) transition relation:

\[
\begin{align*}
\text{\((x > 0 \mapsto x := x - 3 \parallel x \geq 0 \mapsto x := x - 5 \parallel x < 0 \mapsto \text{skip}), [x = 5]\)} & \quad \xrightarrow{\sigma} \text{Both guards are true, choose second alternative} \\
\text{\((x := x - 5, [x = 5])\)} & \quad \text{\{evaluate \(x - 5\) and assign result to \(x\}\}} \\
\text{\((x, [x = 0])\)} & \quad \downarrow
\end{align*}
\]

\[
\begin{align*}
\text{\((x > 0 \mapsto x := x - 3 \parallel x \geq 0 \mapsto x := x - 5 \parallel x < 0 \mapsto \text{skip}), [x = 5]\)} & \quad \xrightarrow{\sigma} \text{Both guards are true, choose first alternative} \\
\text{\((x := x - 3, [x = 5])\)} & \quad \text{\{evaluate \(x - 3\) and assign result to \(x\}\}} \\
\text{\((x, [x = 2])\)} & \quad \downarrow
\end{align*}
\]

Since the process chooses between two actions that are not \(\prec\)-related to each other (in fact, both actions are \(\emptyset\) actions), the execution is the same under the \(\rightarrow\) priority transition relation.

**Example 3.5.2 (Delta statements and timeouts)** In Example 2.1.8 on page 7, we argued that in \(\chi^-\) delta statements are different from timeouts statements, even though the same notation is used. In \(\chi^+\) we have a different notation for timeouts: \(\Theta_{e_r}\). In this example, we illustrate the difference between delta statements and timeouts. We will leave out the variable valuation, since it is not relevant here. Translating the \(\chi^-\) example into \(\chi^+\), we get the following process (assuming \(e_r = 2\)):

\[
\text{(true ; !}(m, 0, 10) \mapsto s_1 \parallel \text{true} ; \Theta \mapsto !}(n, 0, 1) \parallel \text{\(\Delta 2\)} ; ?(m, 0, x) \parallel ?(n, 0, y)
\]

Under the priority transition relation \(\Rightarrow\), the process has the behavior described in the \(\chi^-\) example, i.e., first the delta statement is executed, then communication over channel \(m\) takes place.
The transition \( c(m, 2, x, 10) \rightarrow \) is preferred over \( \Theta \), since communication actions have higher priority than the timeout action. If we had considered the execution of the process under the normal transition relation \( \rightarrow \), the process would make a nondeterministic choice between \( c(m, 2, x, 10) \) and \( \Theta \).

In \( \chi^+ \), it is allowed to write a \( \Delta r \) as an event in a guard. For example, consider the process:

\[
(\text{true} ; !(m, 0, 10) \rightarrow s_1 \equiv \text{true} ; \Delta 2 \rightarrow !(n, 0, 1)) \parallel \Delta 2 ; ?(m, 0, x) \parallel ?(n, 0, y)
\]

The behavior of this process is described by the following trace (note that the nondeterministic choice does not occur. Hence, under the \( \rightarrow \) transition relation, the execution will be the same):

\[
(\text{true} ; !(m, 0, 10) \rightarrow s_1 \equiv \text{true} ; \Delta 2 \rightarrow !(n, 0, 1)) \parallel \Delta 2 ; ?(m, 0, x) \parallel ?(n, 0, y)
\]

\[
\xrightarrow{2} \{ \text{delay of 2 time units} \}
\]

\[
(!(m, 2, 10) \rightarrow s_1 \equiv \text{true} ; ?(m, 0, x) \parallel ?(n, 0, 1)) \parallel \epsilon \parallel ?(n, 2, y)
\]

\[
\xrightarrow{1} \{ \text{possible action: } c(m, 2, x, 10) \text{ and } \Theta, \text{ but } c(m, 2, x, 10) \prec \Theta \}
\]

\[
(\epsilon \rightarrow s_1 \equiv \text{false} \equiv !(n, 0, 1)) \parallel \epsilon \parallel ?(n, 2, y)
\]

\[
\xrightarrow{\epsilon} \{ \text{selection action } \epsilon \text{ only possibility} \}
\]

\[
s_1 \equiv \epsilon \parallel ?(n, 2, y)
\]

\[
\xrightarrow{\ldots}
\]

**Example 3.5.3 (Zeno behavior)** In Section 3.4, we argued that greater positive time steps should have higher priority than smaller positive time steps in order to prevent Zeno behavior. However, we also mentioned that it is possible to specify a process that exhibits Zeno behavior. In this example, we give such a process.

\[
x := 1 ; * (\Delta x ; x := x/2) \parallel \Delta 2, [x =?]\]
\]

\[
x := 1 \rightarrow
\]

\[
(\epsilon ; * (\Delta x ; x := x/2) \parallel \Delta 2, [x = 1])
\]

\[
\xrightarrow{1} \{ \text{delay of 1 time unit} \}
\]

\[
(\epsilon ; x := x/2 ; * (\Delta x ; x := x/2) \parallel \Delta 1, [x = 1])
\]

\[
x := 1/2 \rightarrow
\]

\[
(\epsilon ; * (\Delta x ; x := x/2) \parallel \Delta 1, [x = 1/2])
\]

\[
\xrightarrow{1/2} \{ \text{delay of } 1/2 \text{ time units} \}
\]

\[
(\epsilon ; x := x/2 ; * (\Delta x ; x := x/2) \parallel \Delta 1/2, [x = 1/2])
\]

\[
x := 1/4 \rightarrow
\]

\[
\ldots
\]

21
In this example, time does not progress to 2. Positive time passing actions will be executed repeatedly, but their cumulative effect is a time advance of less than two time units. Therefore, the second process, \( \Delta 2 \), will never terminate.

### 3.6 Discussion

The \( \chi^+ \) language is more general than the \( \chi^- \) language; for every \( \chi^- \) statement there is a \( \chi^+ \) statement with the same semantics, but there are \( \chi^+ \) statements for which there is no \( \chi^- \) variant. This is achieved mainly by using simpler operators in \( \chi^+ \). For example, the selective waiting operator of \( \chi^- \) is translated using the sequential composition operator, the select operator, and the choice operator of \( \chi^+ \). Since the simpler operators can be used on their own, we can construct \( \chi^+ \) statements for which no \( \chi^- \) equivalents exist. An example is the second process of Example 3.5.2; that process has a \( \chi^+ \) delta statement as an event in a guard, which is different from the \( \chi^- \) delta statement. Another example is:

\[
\begin{align*}
&\{ \text{true ; !(m}_{1},0,10) ; ?(n_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \text{true ; ?(n}_{2},0,y) ; y := y \times x ; !(m}_{2},0,x) \} \quad \rightarrow \quad \text{skip}
\end{align*}
\]

If we try to translate this into \( \chi^- \), we see that we need a generalized version of the selective waiting statement in which we could put more than one event in the guards. In addition, we should also be able to place assignments in the guards.

It is unclear whether such a general selective waiting statement is useful to \( \chi \) specification writers, since the meaning of the statement has some peculiarities. For example, suppose \( q \) is a \( \chi^+ \) statement, a possible trace of the parallel composition of the statement given above and \( q \) with valuation \([x = 1, x_q = 1, y = 1]\) could be:

\[
\begin{align*}
&\{ \text{true ; !(m}_{1},0,10) ; ?(n_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \text{true ; ?(n}_{2},0,y) ; y := y \times x ; !(m}_{2},0,x) \} \quad \rightarrow \quad \text{skip}
\end{align*}
\]

\[
\begin{align*}
&\{ \{x = 2, x_q = 1, y = 1\} \} \quad \rightarrow \quad q
\end{align*}
\]

\[
\begin{align*}
&\{ \epsilon ; ?(n}_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \epsilon ; ?(n}_{2},0,y) ; y := y \times x ; !(m}_{2},0,x) \} \quad \rightarrow \quad \text{skip}
\end{align*}
\]

\[
\begin{align*}
&\{ \epsilon ; !m_{1},0,10 ; n_{2},0,10) \} \quad \rightarrow \quad{\{\text{suppose } q \text{ receives over } m_1 \text{ in variable } x_1\}} \\
&\{ \epsilon ; ?(n}_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \epsilon ; ?(n}_{2},0,y) ; y := y \times x ; !(m}_{2},0,x) \} \quad \rightarrow \quad \text{skip}
\end{align*}
\]

\[
\begin{align*}
&\{ \{x = 2, x_q = 10, y = 1\} \} \quad \rightarrow \quad q'
\end{align*}
\]

\[
\begin{align*}
&\{ \epsilon ; ?(n}_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \epsilon ; !m_{2},0,10 ; n_{2},0,10) \} \quad \rightarrow \quad{\{\text{suppose } q' \text{ sends over } n_2 \text{ value 10}\}} \\
&\{ \epsilon ; ?(n}_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \epsilon ; !m_{2},0,10 ; n_{2},0,10) \} \quad \rightarrow \quad{\{\text{assignment has higher priority than communication}\}} \\
&\{ \epsilon ; !m_{2},0,10 ; n_{2},0,10) \} \quad \rightarrow \quad{\{\text{assignment has higher priority than communication}\}} \\
&\{ \epsilon ; ?(n}_{1},0,x) \} \quad \rightarrow \quad x := x - 10 \\
&\{ \epsilon ; !m_{2},0,10 ; n_{2},0,10) \} \quad \rightarrow \quad{\{\text{assignment has higher priority than communication}\}} \\
&\{ \epsilon ; !m_{2},0,10 ; n_{2},0,10) \} \quad \rightarrow \quad{\{\text{assignment has higher priority than communication}\}}
\end{align*}
\]

\[
\begin{align*}
&\{ x = 2, x_q = 10, y = 10\}
\end{align*}
\]

\[
\begin{align*}
&\{ y := 20 \}
\end{align*}
\]

\[
\begin{align*}
&\{ x = 1, x_q = 10, y = 20\}
\end{align*}
\]
So, the generalized selective waiting can execute several events from each alternative. As soon as all events of one alternative have been executed, i.e., as soon as the event of one alternative becomes the empty statement, the select operator will execute a selection action, thereby actually choosing the corresponding alternative.

Consequently, whenever an alternative is chosen, it is possible that some (but not all) events of other alternatives have been executed too. The specification writer has to make sure these events can be ignored safely, or otherwise take appropriate actions after the selection action \( o \).

The repetition is another statement that is more general in \( \chi^+ \) than the repetition of \( \chi^- \). However, this generalized statement does have an interpretation very close to the intuition of the repetition statements in \( \chi^- \). The main difference is that in \( \chi^- \) we can only make repetitions from guarded commands or selective waitings, whereas in \( \chi^+ \) we can make repetitions from arbitrary statements.

4 \( \chi^+ \) semantics in ASF+SDF

The sos-rules given in the previous section have been implemented using the ASF+SDF meta-environment. The ASF+SDF meta-environment is an interactive development environment for formal languages (see [Kli93]). The implementation of the rules in ASF+SDF is straightforward, for example, Rule 21 is:

\[
\begin{align*}
(p, v) & \rightarrow (p', v'), \ a \not\in \{o\} \cup R^+ \\
(q \parallel p, v) & \rightarrow (q \parallel p', v')
\end{align*}
\]

The ASF+SDF implementation of this rule reads:

\[
\text{[alt1]} \quad (q,p,v) \rightarrow \text{true} \\
\text{when } a \neq o, \quad \text{not(isPosDelay(a)) = true}, \\
\text{not(isNegDelay(a)) = true}, \\
(p,v) \rightarrow (p',v') = \text{true}
\]

Since ASF+SDF does not have a concept of truth values, but only deals with equalities and inequalities, we had to define the rules using equalities and inequalities. Therefore, we define a valid rule \( r \) to be equal to true: \( r = \text{true} \).

In addition to the sos-rules, we have implemented a simple simulator for \( \chi^+ \). The simulator can compute all possible traces and intermediate states of a \( \chi^+ \) process. During the development of the sos-rules of \( \chi^+ \), the simulator was used to validate the sos-rules against the "real" \( \chi \) simulator [NA98] and the previous version of the semantics of \( \chi \), see [vdMF95].

5 Differences

The complete \( \chi \) language has many constructs with which one can write clear and concise specifications. A lot of the constructs can be considered syntactic sugar and therefore not interesting.
while describing the semantics of \( \chi \). In this document, a minimal subset of discrete \( \chi \) was considered that still has enough features to describe communicating parallel processes with global continuous time. In this section, we list the main differences between \( \chi, \chi^-, \) and \( \chi^+ \). We also describe the main differences between the operational semantics of \( \chi \) given in [vdMF95] and the operational semantics defined in the previous sections.

\( \chi \)-features not included in \( \chi^- \) and \( \chi^+ \)

**Tuples, and sets**

In \( \chi \) one can construct list types, tuple types, and set types. If used properly, these "compound types" increase the readability of a specification.

In this report we treated expressions and types informally as we concentrated on the dynamic aspects of the language. Consequently, we could not introduce complex expressions and types, since that would need a more thorough treatment of their syntax and semantics.

**Constants**

Constant expressions can be given a name in \( \chi \). The name is an abbreviation that can be used in expressions, thereby increasing the readability of the specification.

**Functions**

In \( \chi \) one can define (recursive) functions. Since we treated expressions only informally, there was no need to formally describe function definitions.

**Systems**

Systems in \( \chi \) are parallel compositions of other systems and/or sequential processes. The class \( PS \) of Table I defines the \( \chi \) systems. In \( \chi^+ \), we did not make a distinction between systems and processes.

**Declarations and instantiations**

In \( \chi \) one declares processes and systems in a "name-parameters-body style". The declarations are generic descriptions of processes and systems that function similarly to class definitions written in object-oriented programming languages (e.g., C++). Instantiations of processes and systems can be used for simulations. The behavior of the running instantiations is analyzed. Therefore, in this report we only considered instantiated processes.

**Distributions and samples**

\( \chi \) has constructs to generate samples according to particular distributions. This can be used to model stochastic behavior, e.g., machine failure. In future, these features will be included in the formal semantics of \( \chi \).

**Experiments**

An experiment in \( \chi \) defines a fully instantiated system and is mainly used to tell the \( \chi \) simulator what system should be simulated.

**Bundles**

Bundles are a syntactic means to define and use an array of channels. The channels in a bundle are accessed using indices over a user-defined integer range. Bundles can drastically increase the readability of complex specifications. For example, if a buffer is connected to ten machines, using one bundle-parameter in the declaration of the buffer is probably better than ten channel-parameters. Since bundles are purely syntactic abbreviations, they were outside the scope of this report.
Differences with previous semantics

Global vs local time

In the report mentioned, \( \chi \) processes have local times. The local time of a process advances independently of the local times of other processes. If two processes communicate, the local times will be synchronized.

We did not assign each process its own local time for several reasons:

1. Most specification writers using \( \chi \) do not think in terms of processes with local times.
2. The \( \chi \) simulator has global time.
3. Due to synchronization of local times during communications, it is difficult to take advantage of local times. Local times can be independent only in the case of two (groups of) processes that do not communicate with each other. However, such (groups of) processes might as well be specified separately.
4. We think that assigning local times to processes will make the semantics of processes more complicated.

Open systems/processes

The current definition of the semantics does not demand every channel to be connected to a reader process and a sender process. In the previous semantics, such systems were called open systems (or processes) and their semantics was not defined.

6 Future research

The sos-rules given in the previous sections constitute one step in formal analysis of \( \chi \) specifications. There are several interesting research areas related to this that could be investigated. First of all, the operational semantics itself could be investigated. This could result in other semantics of \( \chi \) (denotational or axiomatic) and theorems relating the different semantics. In the end, this increases the formal understanding of \( \chi \) which is necessary to reason formally about \( \chi \) specifications.

Second, a translation of \( \chi \) into an existing formalism, like CSP [Hoa85, HJ98], ACP [BW95, BV95], or \( \mu \)CRL [Gro97], could prove to be useful. Well known formalisms have the advantage of a large repertoire of theorems that simplify establishing proofs of specific properties. In addition, such a translation probably clarifies similarities and differences between \( \chi \) and other formalisms, which in turn could lead to a better understanding of \( \chi \).

Another direction could be to implement the sos-rules in existing tools. A first step in this direction has been taken already by implementing the sos-rules in the ASF+SDF meta-environment. However, at the moment this implementation is rather primitive and not suitable for real life \( \chi \) specifications. Other kinds of existing tools are model checkers, e.g., SPIN [Hol91] or Caesar Aldebaran [FGM+92] and interactive proof assistants, e.g., PVS [Sha96, ORR+96] or HOL [GM93]. The type of verification one can perform with model checkers is very different from that of proof assistants. Therefore, it depends on the particular analysis one wants to perform on \( \chi \) specifications which kind of tool is more suitable.

Finally, the semantics and its usefulness should be put to the test by analyzing case studies of industrial systems. This is the only way to detect flaws in the formal definitions and to improve them. Furthermore, "doing case studies" will also show what kind of formal reasoning really adds to writing better \( \chi \) specifications.

References


In this series appeared:

- 96/02 P. de Bra and A. Aerts: Multi-User Publishing in the Web: DmSS, A Document Repository Service Station, p. 12.
- 96/04 S. Mauw: Example specifications in phi-SDL.
  Inheritance = Encapsulation + Abstraction, p. 15.
- 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.
- 96/12 F. Kamareddine and T. Laan: A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
- 96/13 T. Borghuis: Priorean Tense Logics in Modal Pure Type Systems, p. 61.
- 96/16 E. Buisen and P. Hoogendijk: Nested collections and polytypism, p. 11.
- 96/20 L. Feijs: Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
- 96/23 W.M.P. van der Aalst: Structural Characterizations of sound workflow nets, p. 22.
- 96/24 M. Voorhoeve and W. van der Aalst: Conservative Adaption of Workflow, p. 22.
- 96/25 M. Vaccari and R.C. Backhouse: Deriving a systolic regular language recognizer, p. 28.
- 97/07 J.C.M. Baeten and J.A. Bergstra: Bounded Stacks, Bags and Queues, p. 15.
- 97/08 P. Hoogendijk and R.C. Backhouse: When do datatypes commute? p. 35.


97/13 W.M.P. van der Aalst Exploring the Process Dimension of Workflow Management, p. 56.

97/14 J.F. Groote, F. Monin and J. Springintveld A computer checked algebraic verification of a distributed summation algorithm, p. 28.

97/15 M. Franssen λP*: A Pure Type System for First Order Logic with Automated Theorem Proving, p. 35.

97/16 W.M.P. van der Aalst On the verification of inter-organizational workflows, p. 23.

97/17 M. Vaccari and R.C. Backhouse Calculating a Round-Robin Scheduler, p. 23.


98/02 M. Voorhoeve State / Event Net Equivalence, p. 25.

98/03 J.C.M. Baeten and J.A. Bergstra Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.


98/05 D. Dams Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers, P. 22.


98/10 T. Basten and J. Hooman Process Algebra in PVS

98/11 J. Zwanenburg The Proof-assistant Yarrow, p. 15.
