An algebraic semantics of message sequence charts

Citation for published version (APA):

Document status and date:
Published: 01/01/1994

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.
An algebraic semantics of Message Sequence Charts

by

S. Mauw and M.A. Reniers

94/23
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.
An algebraic semantics of Message Sequence Charts

S. Ma.uw  M.A. Reniers
Dept. of Mathematics and Computing Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands.
sjouke@win.tue.nl  michelr@win.tue.nl

Abstract

Message Sequence Charts is a graphical and textual language for the description and specification of the interactions between system components and their environment. The language is standardized by the ITU-TS (the Telecommunication Standardization section of the International Telecommunication Union, the former CCITT). The main area of application for Message Sequence Charts is as an overview specification of the communication behavior of real-time systems, in particular telecommunication switching systems. Message Sequence Charts may be used for requirement specification, interface specification, simulation and validation, test-case specification and documentation of real-time systems.

In this paper we present a formal semantics of Message Sequence Charts exploiting techniques from process algebra. This semantics is proposed for standardization within the ITU-TS. We start with the semantics of the core language of Message Sequence Charts, Basic Message Sequence Charts, and subsequently add other features such as process creation and termination, refinement and timer handling.

1 Introduction

Message Sequence Charts is a graphical language for the description of the interactions between entities, standardized by the ITU-TS (the Telecommunication Standardization section of the International Telecommunication Union, the former CCITT). Besides the graphical syntax there is also a textual syntax for Message Sequence Charts. ITU recommendation Z.120 [CCI92] contains both the graphical and the textual syntax and an informal explanation of the semantics. The current goal in the process of standardization is the definition of a formal semantics of the language. The need for a formal semantics became evident when even experts in the field of Message Sequence Charts could not always agree on the interpretation of specific features. Furthermore validation of computer tools for Message Sequence Charts only makes sense if an exact meaning is available. Finally a formal semantics will help to harmonize the use of Message Sequence Charts.

Several attempts have been made to achieve such a formal semantics. We mention approaches based on automata theory [LL94], Petri net theory [GGR93] and on process algebra [dM93, MvWW93]. None of these papers contains a formal semantics of the complete language. Although all approaches have their advantages and disadvantages, it has been decided by the standardization committee to use process algebra for the formal definition. In this paper\(^1\) a formal semantics of Message Sequence Charts based on process algebra will be defined for the textual representation.

---

\(^1\)This document contains the proposal for the formal semantics of Message Sequence Charts as submitted to the ITU-TS d.d. April 18, 1994. The final recommendation will be based on this document.
of Message Sequence Chart. The relation between the Message Sequence Charts in one Message Sequence Chart document is in general not explicitly stated. Therefore, we will only describe the semantics of a single Message Sequence Chart with its corresponding Sub Message Sequence Charts.

This work is related to the formal semantics of Interworkings in [MvWW93]. A difference is that we consider asynchronous communication whereas the theory of Interworkings is only concerned with synchronous communication. Furthermore, Message Sequence Charts and Interworkings have a different approach with respect to their textual representation. Interworkings are event oriented, which means that an Interworking is a list of communications and other events, whereas Message Sequence Charts are instance oriented. This means that a Message Sequence Chart is described by giving the behavior of every instance in separation.

The formal semantics presented in this paper is based on the algebraic theory of process description ACP (Algebra of Communicating Processes) [BW90]. ACP is an algebraic theory in many ways related to the algebraic process theories CCS (Calculus of Communicating Systems) [Mil80] and CSP (Communicating Sequential Processes) [Ho85]. The process algebra ACP is a useful framework for the description of the formal semantics of Message Sequence Charts since all features incorporated in the theory of Message Sequence Charts are related to topics already studied in process algebra such as the state operator and the global renaming operator. Since we consider asynchronous communication and since Message Sequence Charts may be "empty", we use PA, i.e. ACP without the communication function and with the empty process [BW90].

This paper is structured in the following way. First we introduce Basic Message Sequence Charts, the core language of Message Sequence Charts. Within a Basic Message Sequence Chart only local actions and communications can be specified. These are the features that are incorporated in most languages comparable to Message Sequence Charts. The other features incorporated in Message Sequence Charts are introduced one by one. The static requirements imposed on Message Sequence Charts which are of importance for the definition of a formal semantics are given. Next we define the algebraic theory we use as a formal framework for the definition of the semantics. After that we define the semantic function which maps Message Sequence Charts into process terms. First we define the semantics of Basic Message Sequence Charts, and we add subsequently the special features until we have reached a formal semantics for the complete language of Message Sequence Charts.

2 Message Sequence Charts

2.1 Introduction

In this section we introduce Message Sequence Charts. Message Sequence Charts have both a graphical and a textual representation. The language is best illustrated by the graphical representation, but where the definition of a formal semantics is concerned, the textual representation is preferred.

First we introduce the core language of Message Sequence Charts, which is called Basic Message Sequence Charts. A Basic Message Sequence Chart concentrates on communications and local actions only. These are the features encountered in most languages comparable to Message Sequence Charts such as Extended Sequence Charts, Arrow Diagrams, Information Flow Diagrams, Sequence Charts, Message Flow Diagrams, Siemens-SCs, and Interworkings. The static requirements imposed on Basic Message Sequence Charts, as far as they are of importance to the definition of the formal semantics in Section 4, are given. The static requirements are not formalized. In [CC92] the static requirements for Message Sequence Charts are given informally.
Thereafter we introduce the other primitives incorporated in the language of Message Sequence Charts. These primitives are process creation and process termination, timer handling, coregions, conditions, and refinement.

In this paper we do not use the concrete textual syntax for Message Sequence Charts as defined in [CCI92], but an alternative syntax. We feel that the original concrete textual syntax is not suited for the definition of the formal semantics and the auxiliary functions used in the definition thereof. In Appendix A both the original concrete textual syntax and the alternative syntax are presented. The languages generated by the both syntaxes are equal with respect to those constructs which have a semantical meaning.

2.2 Basic Message Sequence Charts

A Basic Message Sequence Chart is a finite collection of instances. An instance is an abstract entity on which message outputs, message inputs and local actions may be specified. An instance is denoted by a vertical axis. The time along each axis is running from top to bottom. The events specified on an instance are totally ordered in time, no notion of global time is assumed. No two events are executed at the same time. An instance is labelled with a name, the instance name. This name is placed above the axis representing the instance.

A local action is denoted by a box on the axis with an identifying name, the action text, placed in that box. A message between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. A message is split into a message output and a message input. A message sent by an instance to the environment is represented by an arrow from the sending instance to the exterior of the Message Sequence Chart. A message received from the environment is represented by an arrow from the exterior of the Message Sequence Chart to the receiving instance. A message may be labelled with a parameter list. It is denoted between brackets behind the message name.

In Figure 1 we consider the messages m1, m2, m3 and m4. Message m0 is sent to the environment. The behavior of the environment is not specified. For instance i2 we also define a local action a.

![Figure 1: Example Basic Message Sequence Charts](image)

The only dependencies between the timing of the instances come from the restriction that a message must be sent before it is consumed. In Figure 1 this implies that message m3 is received by i4 only after it has been sent by i3, and, consequently, after the consumption of m2 by i3. Thus m1 and m3 are ordered in time, while for m4 and m3 no order is specified. The execution of a local action is only restricted by the ordering of events on the instance it is defined on. The second Basic Message Sequence Chart in Figure 1 defines the same Basic Message Sequence Chart (from a semantic point of view), but in an alternative drawing.

3
Since we have asynchronous communication, it would even be possible to first send $m_3$, then send and receive $m_4$, and finally receive $m_3$. Another consequence of this mode of communication is that we allow overtaking of messages, as expressed in Figure 2.

Although the application of Message Sequence Charts is mainly focussed on the graphical representation, they have a concrete textual syntax. This representation was originally intended for exchanging Message Sequence Charts between computer tools only, but in this paper we use it for the definition of the semantics.

The textual representation of a Basic Message Sequence Chart is instance oriented. This means that a Basic Message Sequence Chart is defined by specifying the behavior of all instances. A message output is denoted by "out $m_i$ to $i_j$" and a message input by "in $m_i$ from $i_j$". The Basic Message Sequence Charts of Figure 1 have the following textual representation.

```plaintext
msc example1;
  instance i1;
    out $m_0$ to env;
    out $m_1$ to $i_2$;
    in $m_4$ from $i_2$;
  endinstance;
  instance $i_2$;
    in $m_1$ from $i_1$;
    out $m_2$ to $i_3$;
    action a;
    out $m_4$ to $i_1$;
  endinstance;
  instance $i_3$;
    in $m_2$ from $i_2$;
    out $m_3$ to $i_4$;
  endinstance;
  instance $i_4$;
    in $m_3$ from $i_3$;
  endinstance;
endmsc;
```

In the graphical representation the correspondence between message outputs and message inputs is given by the arrow construction. In the textual representation this correspondence is given by message name and message instance name identification.

The grammar defining the textual syntax of Basic Message Sequence Charts is given in Table 1. The nonterminals $<msc\ name>$, $<inst\ name>$, $<mn>$, $<min>$, $<at>$ and $<par\ name>$ represent identifiers. The symbol $<>$ denotes the empty string. The following identifiers are reserved keywords: action, endinstance, endmsc, env, from, in, instance, msc, out and to.
<msc> ::= msc <msc name>; <msc body> endmsc;
<msc body> ::= <> | <inst def> <msc body>
<inst def> ::= instance <inst name>; <inst body> endinstance;
<inst body> ::= <> | <event> <inst body>
<event> ::= <out> | <in> | <action>
<out> ::= out <mn> [,<min>] [(<par list>)] to <address>;
<in> ::= in <mn> [,<min>] [(<par list>)] from <address>;
<action> ::= action <at>;
<address> ::= <inst name> | env
<par list> ::= <par name> [,<par list>]

Table 1: The concrete textual syntax of Basic Message Sequence Charts

The language generated by a nonterminal \( X \) in the grammar of Table 1 will be denoted by \( L(X) \).

We formulate the static requirements for Basic Message Sequence Charts as follows. All instances that are defined within one Basic Message Sequence Chart must have different instance names. If the address of a message event is an instance name, then there has to be an instance with that name within the Basic Message Sequence Chart. It is not allowed that there are two message outputs or two message inputs with the same message identifier (i.e., the message name and the message instance name) within one Basic Message Sequence Chart. For every message output sent by an instance to an instance there has to be a corresponding message input. For every message input received by an instance from an instance there has to be a corresponding message output. It is not allowed that a message output is causally depending on the corresponding message input, directly or via other messages. This is the case if the temporal ordering of the events imposed by the Basic Message Sequence Chart specifies that a message input is executed before its corresponding message output. Consider, for example, the first chart in Figure 3. Since the events which are specified on one instance are temporally ordered from top to bottom, the message input is executed before the corresponding message output. The chart therefore violates the static requirements. In this example the message output is depending on its corresponding message input in a direct way.

![Figure 3: Two charts that violate the static requirements](image)

As an example of the indirect causal dependency between a message output and a message input we consider the second chart in Figure 3. We have, amongst others, the following temporal orderings:

1. the input of message \( m \) precedes the output of message \( n \),
2. the output of message \( n \) precedes the input of message \( n \), and

3. the input of message \( n \) precedes the output of message \( m \).

Therefore, the chart specifies that the input of message \( m \) precedes the output of message \( m \). So the chart violates the static requirements, and is therefore not a Basic Message Sequence Chart.

### 2.3 Process creation and process termination

In the language of Message Sequence Charts a primitive is incorporated for the dynamic creation of an instance by another instance. Such a creation is denoted by a dashed arrow from the creating instance to the top symbol of the created instance. An instance can be created only once. No events before the creation of an instance may refer to that instance. This requirement introduces a new dependency between the timing of the instances: If there is a message sent from an instance \( i \) to a created instance \( j \), then the message output defined on instance \( i \) may not be executed before instance \( j \) is created. As was the case for message events, a create event may be labelled with a parameter list.

An instance can terminate by executing a process stop event. Execution of a process stop is allowed only as a last event in the description of an instance. A process stop is denoted by replacing the bottom symbol of the instance by a cross. In Figure 4 a Message Sequence Chart with three instances is given. Instance \( i \) creates instance \( j \), instance \( k \) sends a message \( m \) to instance \( j \), and instance \( j \) receives the message \( m \) from instance \( k \) after it is created and then terminates.

![Figure 4: Message Sequence Chart with process creation and termination](image)

The output of message \( m \) refers to instance \( j \) and may therefore only be executed after the creation of instance \( j \). So the events specified in the chart must be executed in the following order:

1. instance \( i \) creates instance \( j \);
2. instance \( k \) sends a message \( m \) to instance \( j \);
3. instance \( j \) consumes the message \( m \) sent by instance \( k \);
4. instance \( j \) terminates.

In the textual representation the creation of an instance with name \( j \) is denoted by "create \( j \);" and the termination of an instance by "stop;". We extend the grammar for Basic Message Sequence Charts in Table 1 with the rules in Table 2. The identifiers create and stop are reserved keywords.
With respect to Basic Message Sequence Charts extended with process creation and termination we add the following static requirements. Only instances that are defined within a Basic Message Sequence Chart may be created. An instance may be created only once and an instance may not create itself. It is not allowed that the creation of an instance \( j \) is causally depending on the execution of an event that refers to instance \( i \). This is the case if the temporal ordering of the events, imposed by the Message Sequence Chart, specifies that an event that refers to a created instance is executed before the execution of the event creating the instance.

### 2.4 Timer handling

In Message Sequence Charts, either the setting of a timer and its subsequent timeout due to timer expiration or the setting of a timer and its subsequent timer reset (time supervision) may be specified. The setting of a timer is denoted by a small rectangle placed against the time axis, a timeout is represented by an arrow from the timer set symbol to the axis, and a timer reset is represented by a modified timeout symbol with a dashed arrow. A timer event is labelled by an identifier, the \( \text{timer name} \), that is placed aside the small rectangle. The setting of a timer may be labelled with an identifier for the duration, the \( \text{duration name} \). The duration name is placed between brackets behind the timer name. A timer event is local to the instance it is specified on. It is not allowed to specify a timer set and a subsequent timeout or timer reset on different instances.

In Figure 5 on instance \( i \) the setting of a timer \( T \) with duration \( d \) and its subsequent timer reset are specified, and on instance \( j \) the setting of a timer \( T \) and its subsequent timeout are specified.

![Image](image_url)

**Figure 5:** Message Sequence Chart with timer handling

In the graphical representation the correspondence between a timer set and a timer reset or timeout is given by the connection of the begin of the timer reset or timeout symbol to the rectangle representing the timer set. In the textual representation the correspondence between timer set and timer reset or timeout is given by timer name and timer instance name identification. The setting of a timer with name \( T \) is denoted by "\( \text{set } T; \)" and the corresponding reset by "\( \text{reset } T; \)" and timeout by "\( \text{timeout } T; \)." The Grammar in Table 2 is extended with the rules in Table 3. The nonterminals \( <tn>, <tin> \) and \( <dn> \) represent identifiers. The identifiers \( \text{set}, \text{reset} \) and \( \text{timeout} \) are reserved keywords.

| <inst body> | ::= stop; |
| <event>     | ::= <create> |
| <create>    | ::= create <inst name> [([par list])]; |

Table 2: Extension of the grammar with process creation and termination
The following static requirements are formulated. The timer identifier (i.e. the timer name and timer instance name) must be unique within an instance definition. With every timer set either a corresponding timer reset or a corresponding timeout has to be specified on the same instance. With every timer reset and every timeout there has to be a corresponding timer set specified on the same instance. The timer set must precede its corresponding timer reset or timeout.

2.5 Coregions

So far we have seen that the events specified on an instance are totally ordered in time. To enable the specification of unordered events on an instance the coregion is introduced. A coregion is a dashed part of the time axis for which the events specified within that part are assumed to be unordered in time. Within a coregion only message events may be specified.

In Figure 6 an instance with a coregion is specified which contains an input of message m and an output of a message n. These two events are not ordered in time, but they are executed after the output of message k and before the input of message l. On instance j the events are totally ordered in time.

In the textual notation a coregion is denoted by a list of the message events specified within the coregion started with the reserved keyword concurrent and ended by the reserved keyword endconcurrent. In Table 4 the rules for the extension with coregions are given.

2.6 Conditions

A condition describes a state referring to a (non-empty) subset of instances specified in the Message Sequence Chart. A condition which refers to only one instance is called local, and a condition
which refers to all instances is called *global*. Conditions are used for documentation purposes in the sense of comments or illustrations. In case of a whole set of Message Sequence Charts conditions determine possible continuations of Message Sequence Charts by means of condition identification.

In the graphical representation a condition is represented by a hexagon that is placed on top of the instances it refers to. If a condition crosses an instance axis which is not involved in the condition the instance axis is drawn through the condition. A condition is labelled with a *condition name* that is placed inside the hexagon.

In Figure 7 a Message Sequence Chart with a global condition $C_1$, a local condition $C_2$ and a condition $C_3$ is given.

![Figure 7: Message Sequence Chart with a coregion](image)

In the textual representation the condition has to be defined on every instance it refers to using the reserved keyword *condition* together with the condition name. If the condition refers to several instances then the reserved keyword *shared* together with the *instance list* denotes the set of all instances with which the condition is shared. In case of a global condition we may replace the instance list by the reserved keyword *all*. The Message Sequence Chart in Figure 7 is in the textual representation given by

```plaintext
msc cond;
  instance i;
    condition C2;
    condition C1 shared all;
    condition C3 shared k;
  endinstance;
instance j;
```

Table 4: Extension with coregions

| `<event>` | ::= | `<coregion>` |
| `<coregion>` | ::= | concurrent `<coevents>` endconcurrent; |
| `<coevents>` | ::= | <> | <out> `<coevents>` | <in> `<coevents>` |
In Table 5 the rules for the extension with conditions are given. The nonterminal `<cn>` represents an identifier. The identifiers `condition`, `shared` and `all` are reserved keywords.

```
<event> ::= <condition>
<condition> ::= condition <cn> [shared { <shared inst list> | all } ];
<shared inst list> ::= <inst name> [,<shared inst list>]
```

Table 5: Extension with conditions

### 2.7 Refinement of instances

Since Message Sequence Charts can be rather complex, there is a need for a refinement of one instance by a set of instances defined in another Message Sequence Chart. By means of the keyword `decomposed` placed in the top symbol of the instance a Sub Message Sequence Chart with the same name may be attached to that instance.

Such a Sub Message Sequence Chart is, in fact, a Message Sequence Chart in itself. The Sub Message Sequence Chart represents a decomposition of the instance without affecting its observable behavior. The refinement of a decomposed instance into a Sub Message Sequence Chart may not affect the ordering of the message events defined on the decomposed instance. There is no formal mapping between non-message events specified in the Sub Message Sequence Chart and the events specified on the decomposed instance.

In Figure 8 a Message Sequence Chart and a Sub Message Sequence Chart are given. The decomposed instance `d` is refined by the Sub Message Sequence Chart.

![Figure 8: Refinement](image-url)
In the textual representation these charts are represented by

```plaintext
msc decinst;
  instance i;
  out m to d;
endinstance;
instance d decomposed;
  in m from i;
  out n to env;
endinstance;
endmsc;
```

```plaintext
submsc d;
  instance j;
  in m from env;
endinstance;
instance k;
  in o from j;
  out n to env;
endinstance;
endsubmsc;
```

In the textual representation, an instance that is decomposed is labelled with the reserved keyword `decomposed`. Because of the refinement primitive we are no longer dealing with one single Message Sequence Chart, but with a collection of charts, both Message Sequence Charts and Sub Message Sequence Charts, which are related through the refinement of decomposed instances by Sub Message Sequence Charts. Such a collection of charts is called a Message Sequence Chart document. In Table 6 the rules for the extension with instance refinement are given. The nonterminal `<doc name>` represents an identifier. The identifiers `decomposed, mscdocument, endmscdocument, submsc and endsubmsc` are reserved keywords.

| `<msc doc>` | := mscdocument `<doc name>`; <doc body> endmscdocument; |
| `<doc body>` | := `<` | `<msc` | `<doc body>` | `>` | `<submsc` | `<doc body>` |
| `<inst def>` | := instance `<inst name` decomposed; `<inst body` endinstance; |
| `<submsc>` | := submsc `<msc name`; `<msc body` endsubmsc; |

Table 6: Extension with instance refinement

The following static requirements are formulated. In a Message Sequence Chart document no two charts with the same name may be defined. With every decomposed instance in one of the charts of the Message Sequence Chart document a corresponding Sub Message Sequence Chart with the same name has to be defined. On a decomposed instance no create events may be specified. A decomposed instance may not be created. After replacing all decomposed instances of a chart by their corresponding Sub Message Sequence Charts the resulting chart has to respect all previously mentioned requirements. A decomposed instance may not be refined by the chart it is defined in, directly or via a number of refinements.

3 The process algebra $PA_c$

3.1 Introduction

The process algebra $PA_c$ is an algebraic theory for the description of process behavior [BW90]. Such an algebraic theory is given by a signature defining the processes and a set of equations defining the equality relation on these processes. The signature of $PA_c$ is denoted by $\Sigma_{PA_c}$ and the set of equations is denoted by $E_{PA_c}$. 

11
3.2 The signature of $PA_e$

Before we turn to the signature of $PA_e$, we define the terms associated with a signature $\Sigma$ and a set of variables $V$ and the derivability of an equation with respect to an algebraic theory. A signature $\Sigma$ is a set of constant and function symbols. For every function symbol in the signature its arity is specified.

**Definition 3.2.1** Let $\Sigma$ be a signature and let $V$ be a set of variables. Terms over signature $\Sigma$ with variables from $V$ are defined inductively by

1. $v \in V$ is a term
2. if $c \in \Sigma$ is a constant symbol, then $c$ is a term
3. if $f \in \Sigma$ is an $n$-ary ($n \geq 1$) function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term

The set of all terms over a signature $\Sigma$ with variables from $V$ is denoted by $T(\Sigma, V)$. A term $t \in T(\Sigma, V)$ is called a closed term if $t$ does not contain variables. The set of all closed terms over a signature $\Sigma$ is denoted by $T(\Sigma)$.

Next, we define the derivability of equations with respect to a signature $\Sigma$ and a set of equations $E$. This derivability relation expresses which equations between the process terms associated with the signature $\Sigma$ can be derived from the equations of $E$. If $\sigma : V \rightarrow T(\Sigma, V)$ is a substitution, then $\overline{\sigma} : T(\Sigma, V) \rightarrow T(\Sigma, V)$ denotes the extension of $\sigma$ to terms in $T(\Sigma, V)$ in the obvious way.

**Definition 3.2.2** Let $\Sigma$ be a signature and let $E$ be a set of equations over the signature $\Sigma$ and a set of variables $V$. Then the derivability of an equation $e$ with respect to the algebraic theory $(\Sigma, E)$, notation $(\Sigma, E) \vdash e$, is for all $s, t, u \in T(\Sigma, V)$ defined by

1. if $s = t \in E$, then $(\Sigma, V) \vdash s = t$
2. for all $s \in T(\Sigma, V)$: $(\Sigma, V) \vdash s = s$
3. if $(\Sigma, V) \vdash s = t$, then $(\Sigma, V) \vdash t = s$
4. if $(\Sigma, V) \vdash s = t$ and $(\Sigma, V) \vdash t = u$, then $(\Sigma, V) \vdash s = u$
5. if $(\Sigma, V) \vdash s = t$, then for any substitution $\sigma : V \rightarrow T(\Sigma, V)$: $(\Sigma, V) \vdash \overline{\sigma}(s) = \overline{\sigma}(t)$
6. if $(\Sigma, V) \vdash s = t$ for some $s, t \in T(\Sigma, V)$, then for all contexts\(^2\) $C[]$; $(\Sigma, V) \vdash C[s] = C[t]$

Now we are ready to turn to the signature $\Sigma_{PA_e}$ of $PA_e$. The signature $\Sigma_{PA_e}$ consists of

1. the special constants $\delta$ and $\varepsilon$
2. the set of unspecified constants $A$
3. the unary operator $\checkmark$
4. the binary operators $+,$ $\cdot,$ $\parallel$ and $\parallel$

\(^2\)A context is a term containing a hole, e.g. $[\text{Klo}\text{[02]]}$. 

12
The special constant \( \delta \) denotes the process that has stopped executing actions and cannot proceed. This constant is called *deadlock*. The special constant \( \varepsilon \) denotes the process that is only capable of terminating successfully. It is called the *empty process*. The elements of the set of unspecified constants \( A \) are called *atomic actions*. These are the smallest processes in the description. This set is considered a parameter of the theory. We will specify this set as soon as we consider an application of the theory.

The binary operators \(+\) and \(\cdot\) are called the *alternative* and *sequential composition*. The alternative composition of the processes \( x \) and \( y \) is the process that either executes process \( x \) or \( y \) but not both. The sequential composition of the processes \( x \) and \( y \) is the process that first executes process \( x \), and upon completion thereof starts with the execution of process \( y \).

The binary operator \( \parallel \) is called the *free merge*. The free merge of the processes \( x \) and \( y \) is the process that executes the processes \( x \) and \( y \) in parallel. For a finite set \( D = \{d_1, \ldots, d_n\} \), the notation \( \|d \in D P(d) \) is an abbreviation for \( P(d_1) \parallel \cdots \parallel P(d_n) \). If \( D = \emptyset \) then \( \|d \in D P(d) = \varepsilon \). For the definition of the merge we use two auxiliary operators. The termination operator \( \sqcup \) applied to a process \( x \) signals whether or not the process \( x \) has an option to terminate immediately. The binary operator \( \parallel \) is called the *left merge*. The left merge of the processes \( x \) and \( y \) is the process that first has to execute an atomic action from process \( x \), and upon completion thereof executes the remainder of process \( x \) and process \( y \) in parallel.

The precedence of the operators is as follows: \( \cdot \) binds stronger than all other operators, and \( + \) binds weaker. The other operators have the same binding power. Brackets are associated to the left. We sometimes use \( xy \) to denote \( x \cdot y \).

### 3.3 The equations of \( PA_\varepsilon \)

The set of equations \( E_{PA_\varepsilon} \) of \( PA_\varepsilon \) specifies which processes are considered equal. An equation is of the form \( t_1 = t_2 \), where \( t_1, t_2 \in T(\Sigma_{PA_\varepsilon}, V) \).

For \( a \in A \) and \( x, y, z \in V \), the equations of \( PA_\varepsilon \) are given in Table 7.

Axioms A1–A9 are well known. The axioms TE1–TE4 express that a process \( x \) has an option to terminate immediately if \( \sqrt{x} = \varepsilon \), and that \( \sqrt{x} = \delta \) otherwise. In itself the termination operator is not very interesting, but in defining the free merge we need this operator to express the case in which both processes \( x \) and \( y \) are incapable of executing an atomic action. Axiom TM1 expresses that the free merge of the two processes \( x \) and \( y \) is their interleaving. This is expressed in the three summands. The first two state that \( x \) and \( y \) may start executing. The third summand expresses that if both \( x \) and \( y \) have an option to terminate, their merge has this option too.

**Definition 3.3.1** *Basic \( PA_\varepsilon \) terms* are defined inductively by

1. \( \varepsilon \) and \( \delta \) are basic \( PA_\varepsilon \) terms
2. if \( a \in A \) and \( s \) is a basic \( PA_\varepsilon \) term, then \( a \cdot s \) is a basic \( PA_\varepsilon \) term
3. if \( s \) and \( t \) are basic \( PA_\varepsilon \) terms, then \( s + t \) is a basic \( PA_\varepsilon \) term

A basic \( PA_\varepsilon \) term \( x \) is of the general form

\[
\sum_{k \in K} a_k \cdot x_k + \sum_{l \in L} \varepsilon + \delta
\]

with \( K \) and \( L \) finite index sets, \( a_k \in A \) and \( x_k \) a basic \( PA_\varepsilon \) term (for \( k \in K \)).
Theorem 3.3.1 (Elimination) For every closed $PA_{\ell}$ term $s$, there exists a basic $PA_{\ell}$ term $t$ such that

$$PA_{\ell} \vdash s = t$$

Proof. The definition of basic $PA_{\ell}$ terms used coincides with the definition of basic $BPA_{\ell}$ terms in [BW90]. A proof of this theorem is given there. \qed

Because of Theorem 3.3.1 we have a very simple induction structure. If we want to prove a property of closed $PA_{\ell}$ terms, we only have to prove that the property holds for basic $PA_{\ell}$ terms.

In some proofs we encounter it is easier to use induction on the general form of a basic $PA_{\ell}$ term.

Lemma 3.3.1 For closed $PA_{\ell}$ terms $x, y$ and $z$ we have the following properties:

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

Proof. These properties are taken from [BW90]. \qed

Because of the symmetry and associativity of the alternative composition and the free merge the quantified notations for these operators from the previous section are allowed.
4 Semantics of Message Sequence Charts

4.1 Introduction

In this section the semantics of Message Sequence Charts is presented in a stepwise way. First, we define the atomic actions that represent the smallest events specified within Message Sequence Charts. After that, the semantics of Basic Message Sequence Charts is considered. This semantics is extended to the semantics of the complete language by adding the other primitives one by one. In Section 4.9 an overview of the complete semantics is given.

In defining the semantics of Message Sequence Charts some additional process algebra operators are needed. The extended process algebra is called $PA_{MSC}$. In Section 4.10 we show that $PA_{MSC}$ is a conservative extension of $PA_e$.

In defining the semantic functions a number of auxiliary functions and predicates are used. Since the names of those functions and predicates are chosen to be representative for the intended meaning, we have defined those functions and predicates in Appendix B.

4.2 Specifying the atomic actions

In dealing with Message Sequence Charts we encounter a number of significantly different atomic actions. These are, with their representations in the semantics:

1. the execution of an action $a$ at by instance $i$: $action(i, a)$
2. the sending of a message $m$ with parameter list $p$ by instance $s$ to instance $r$: $out(s, r, m, p)$
3. the sending of a message $m$ with parameter list $p$ by instance $s$ to the environment: $out(s, env, m, p)$
4. the receiving of a message $m$ with parameter list $p$ by instance $r$ from instance $s$: $in(s, r, m, p)$
5. the receiving of a message $m$ with parameter list $p$ by instance $r$ from the environment: $in(env, r, m, p)$
6. the setting of a timer $t$ with duration $d$ by instance $i$: $set(i, t, d)$
7. the setting of a timer $t$ on instance $i$: $set(i, t)$
8. the resetting of a timer $t$ on instance $i$: $reset(i, t)$
9. the timing out of a timer $t$ on instance $i$: $timeout(i, t)$
10. the creation of an instance $j$ with parameter list $p$ by an instance $i$: $create(i, j, p)$
11. the stopping of an instance $i$: $stop(i)$

If no parameter list is specified for a message event or a create event then $p$ is taken to be $\langle\rangle$.

In Table 8 the sets of atomic actions are given. For a nonterminal $X$, we use $L_{\langle\rangle}(X)$ to denote $L(X) \cup \{\langle\rangle\}$.

In order to discriminate between actions performed on different instances we define a function $E$ on atomic actions that associates to each atomic action the set of all instances it refers to. In
Table 8: The atomic actions of $PA_{MSG}$

In this paper we denote, for $X$ an arbitrary set, the power set of $X$ by $P(X)$. In Table 9, for $i,j \in \mathcal{L}(<\text{inst name}>), at \in \mathcal{L}(<ct>), t \in \mathcal{L}(<tn>[,..<tin>])), d \in \mathcal{L}(<dn>), m \in \mathcal{L}(<mn>[,..<min>])), p \in \mathcal{L}(<>(<par list>))$, and $p \in \mathcal{L}(<>(<par list>))$, the function $E: A \rightarrow P(\mathcal{L}(<\text{inst name}>))$ is defined.

Table 9: The function $E$

<table>
<thead>
<tr>
<th>Action</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E(\text{action}(i, at))$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$E(\text{set}(i, t))$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$E(\text{reset}(i, t))$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$E(\text{timeout}(i, t))$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$E(\text{out}(i, j, m, p))$</td>
<td>${i, j}$</td>
</tr>
<tr>
<td>$E(\text{in}(i, j, m, p))$</td>
<td>${i, j}$</td>
</tr>
<tr>
<td>$E(\text{create}(i, env, m, p))$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$E(\text{stop}(i))$</td>
<td>${i}$</td>
</tr>
</tbody>
</table>

4.3 Basic Message Sequence Charts

A Basic Message Sequence Chart specifies a finite number of non-decomposed instances that communicate messages. A message is divided into two parts: a message output and a message input. Within a Basic Message Sequence Chart the correspondence between message outputs and message inputs has to be defined uniquely by means of message identifier identification. Besides the communication of messages a Basic Message Sequence Chart is also allowed to execute local actions. A message input may not be executed before the corresponding message output has been executed.
The general idea of the semantics of a Message Sequence Chart is that it is the free merge of its constituent instances. By this construction we also enable interlevings in which a message output is preceded by its corresponding message input. We introduce an operator \( \lambda_M \) that enables only those interlevings that respect the constraint above. The operator \( \lambda_M \) remembers the message identifier of all message outputs that have been executed in a set \( M \) and only allows a message input if the message identifier thereof is in the set \( M \). After the execution of the message input the message identifier is removed from the set \( M \). The operator \( \lambda_M \) is an instance of the state operator [BB88, BW90].

For all \( M \subseteq L(<\text{msg}>,), x, y \in V, a \in A, i, j \in L(<\text{inst name}>,), m \in L(<\text{msg}>,), p \in L_<<<\text{par list}>>, \) we define the state operator \( \lambda_M \) in Table 10.

| \( \lambda_M(e) = e \) | if \( M = \emptyset \) | LM1 |
| \( \lambda_M(e) = \delta \) | if \( M \neq \emptyset \) | LM2 |
| \( \lambda_M(\delta) = \delta \) | LM3 |
| \( \lambda_M(a \cdot x) = a \cdot \lambda_M(x) \) | if \( a \notin A_\text{a} \cup A_i \) | LM4 |
| \( \lambda_M(\text{out}(i, j, m, p) \cdot x) = \delta \) | if \( m \notin M \) | LM5 |
| \( \lambda_M(\text{out}(i, j, m, p) \cdot x) = \text{out}(i, j, m, p) \cdot \lambda_M(\{m\}(x)) \) | if \( m \in M \setminus \{m\} \) | LM6 |
| \( \lambda_M(\text{in}(i, j, m, p) \cdot x) = \text{in}(i, j, m, p) \cdot \lambda_M(\{m\}(x)) \) | if \( m \in M \) | LM7 |
| \( \lambda_M(\text{in}(i, j, m, p) \cdot x) = \delta \) | if \( m \notin M \) | LM8 |
| \( \lambda_M(x + y) = \lambda_M(x) + \lambda_M(y) \) | LM9 |

Table 10: The state operator \( \lambda_M \)

We will explain some of the axioms defining \( \lambda_M \). If \( \lambda_M \) encounters a non-message event, the event is simply executed and the set \( M \) is not altered (LM4). If a message output with message identifier \( m \) is encountered there are two possibilities. First, we have already executed a message output with message identifier \( m \). In that case we are not allowed to execute any events, so we have a resulting deadlock (LM5). In the other case we may just execute the message output and extend the set \( M \) with the message identifier \( m \) (LM6). If a message input with message identifier \( m \) is encountered and we did not execute a message output with message identifier \( m \) in the past then we cannot execute any more events (LM8). If we did execute a message with message identifier \( m \) then we can just execute the input message and remove the message identifier \( m \) from \( M \) (LM7).

The following examples illustrate the use of the operator \( \lambda_M \):

1. \( \lambda_M(\text{out}(i, j, m, p) \cdot \text{in}(i, j, m, p)) = \text{out}(i, j, m, p) \cdot \lambda_M(\text{in}(i, j, m, p)) \)
2. \( \lambda_M(\text{in}(i, j, m, p) \cdot \text{out}(i, j, m, p)) = \delta \)
3. \( \lambda_M(\text{out}(i, j, m, p) \cdot \text{in}(i, j, m, p)) = \lambda_M(\text{out}(i, j, m, p) \cdot \text{in}(i, j, m, p) + \text{in}(i, j, m, p) \cdot \text{out}(i, j, m, p)) \)
4. \( \lambda_M(\text{in}(i, j, m, p) \cdot \text{out}(i, j, m, p)) \)

The semantic function for Basic Message Sequence Charts, \( S : L(<\text{msg}>) \rightarrow T(\Sigma_{PA_{MSC}}) \), is defined by

\[ S[ch] = \lambda_M \left( \bigl|_{i \in \text{inst}(ch)} S_{\text{inst}[i]} \right) \]
where $\text{Inst}(\text{ch})$ is the set of all instance definitions in Basic Message Sequence Chart $\text{ch}$ and where $S_{\text{inst}}$ is the semantics of one instance in isolation. The function $\text{Inst}$ is defined in Appendix B.

On an instance a number of communication events and actions are defined. The order in which they appear in the textual representation of the Basic Message Sequence Chart is the order in which they are to be executed. The semantic function for instance definitions, $S_{\text{inst}} : \mathcal{L}(\text{inst def}) \rightarrow T(\Sigma_{\text{PA}})$, is defined by

$$S_{\text{inst}}[i] = S_{\text{body}}[\text{InstName}(i)]\begin{pmatrix} \text{InstBody}(i) \end{pmatrix}$$

where the functions $\text{InstName}$ and $\text{InstBody}$ assign to an instance definition respectively its instance name and instance body (see Appendix B for their definitions).

Since in the textual representation the instance on which the events are defined is not specified explicitly, we label the semantic function for the instance body with the instance name of the instance it originates from. For $iid \in \mathcal{L}(\text{inst name})$ the semantic function for instance bodies, $S_{\text{body}}^{iid} : \mathcal{L}(\text{inst body}) \rightarrow T(\Sigma_{\text{PA}})$, is defined by

$$S_{\text{body}}^{iid}[\text{<>}] = \varepsilon$$

$$S_{\text{body}}^{iid}[\text{event} \text{ inst body}] = S_{\text{body}}^{iid}[\text{event}] \cdot S_{\text{body}}^{iid}[\text{inst body}]$$

The semantic function $S_{\text{event}}^{iid}$ gives the semantics of one event in separation. It is merely a translation of the smallest components of Basic Message Sequence Charts into the atomic actions of the process algebra. For $iid \in \mathcal{L}(\text{inst name})$ the semantic function for events, $S_{\text{event}}^{iid} : \mathcal{L}(\text{event}) \rightarrow T(\Sigma_{\text{PA}})$, is defined by

$$S_{\text{event}}^{iid}[\text{action } \langle \text{at} \rangle] = \text{action}(iid, \langle \text{at} \rangle)$$

$$S_{\text{event}}^{iid}[\text{out } \langle \text{mn} \rangle \text{ to } \langle \text{address} \rangle] = \text{out}(iid, \langle \text{address} \rangle, \langle \text{mn} \rangle, \langle \text{min} \rangle, \langle <> \rangle)$$

$$S_{\text{event}}^{iid}[\text{out } \langle \text{mn} \rangle \text{ to } \langle \text{address} \rangle] = \text{out}(iid, \langle \text{address} \rangle, \langle \text{mn} \rangle, \langle \text{min} \rangle, \langle \text{par list} \rangle)$$

$$S_{\text{event}}^{iid}[\text{in } \langle \text{mn} \rangle \text{ from } \langle \text{address} \rangle] = \text{in}(\langle \text{address} \rangle, iid, \langle \text{mn} \rangle, \langle \text{min} \rangle, \langle <> \rangle)$$

$$S_{\text{event}}^{iid}[\text{in } \langle \text{mn} \rangle \text{ from } \langle \text{address} \rangle] = \text{in}(\langle \text{address} \rangle, iid, \langle \text{mn} \rangle, \langle \text{min} \rangle, \langle \text{par list} \rangle)$$

The semantic function for Basic Message Sequence Charts will be illustrated by means of an example. Consider the Basic Message Sequence Chart in Figure 9. It consists of three instances which exchange two messages.

![Figure 9: Example Basic Message Sequence Chart](image)

In the textual representation this Basic Message Sequence Chart is represented by
The interpretation of this Basic Message Sequence Chart is that along instance \( a \) the ordering of the output of messages \( k \) and \( l \) is fixed and furthermore that the output of message \( k \) comes before the input of message \( k \) and, likewise, that the output of message \( l \) comes before the input of message \( l \). These are the only restrictions that apply.

When using the textual syntax, the Basic Message Sequence Chart is represented by describing the behavior of every instance in separation. Strictly speaking the argument of the semantic function \( S_{\text{inst}} \) is an instance definition, but for brevity we write \( S_{\text{inst}}[a] \) if we want to denote the semantics of the instance with name \( a \). After applying the semantic function \( S_{\text{inst}} \) to these instances we obtain

\[
S_{\text{inst}}[a] = \text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)
\]

\[
S_{\text{inst}}[b] = \text{in}(a, b, k, \langle \rangle)
\]

\[
S_{\text{inst}}[c] = \text{in}(a, c, l, \langle \rangle)
\]

The first step in deriving the expression which we aim at is putting the instances \( a \), \( b \) and \( c \) in parallel.

\[
S_{\text{inst}}[a] \parallel S_{\text{inst}}[b] \parallel S_{\text{inst}}[c]
\]

After some calculations, we arrive at the following normalized expression.

\[
(\text{out}(a, b, k, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, c, l, \langle \rangle)) + (\text{in}(a, c, l, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{in}(a, b, k, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{in}(a, c, l, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{in}(a, b, k, \langle \rangle) \cdot \text{in}(a, c, l, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{out}(a, b, k, \langle \rangle) \cdot \text{in}(a, c, l, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{in}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle))
\]

\[
+ (\text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]

\[
+ (\text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, b, k, \langle \rangle) \cdot \text{out}(a, c, l, \langle \rangle)) + (\text{out}(a, c, l, \langle \rangle) \cdot \text{in}(a, b, k, \langle \rangle))
\]
This expression clearly shows execution traces which are not desirable, such as \( \text{in}(a, b, k, <>): out(a, b, k, <>): \text{in}(a, c, l, <>): out(a, c, l, <>). \) These traces are removed by applying the state operator \( \lambda_\sigma \) to this expression. This results in

\[
\begin{align*}
\text{out}(a, b, k, <>): (\text{in}(a, b, k, <>): \text{out}(a, c, l, <>): \text{in}(a, c, l, <>) \\
+ \text{out}(a, c, l, <>): \text{in}(a, c, l, <>): \text{in}(a, b, k, <>))
\end{align*}
\]

### 4.4 Process creation and termination

In this section we extend the semantics of Basic Message Sequence Charts with the semantics of the process creation and process termination primitives.

An instance may be created by another instance by executing a create event. No events before the creation of an instance may refer to the created instance. At the end of an instance body a process termination event may be specified. The execution of a process termination event results in the termination of the instance.

Since the free merge of the semantics of the instances also enables interleavings that do not respect the above constraints we introduce an operator \( \lambda^L \) that only enables the interleavings in which no events are executed that refer to a not yet created instance. The set \( L \) contains the instance names that may be referred to. Events referring to an instance name that is not in \( L \) are blocked. If a create event is executed the name of the created instance is added to \( L \). Note that the environment may always be referred to. The operator \( \lambda^L \) is, just as \( \lambda_M \), an instance of the state operator.

In Section 4.2 we defined a function \( E \) that associates with an atomic action the instances it refers to. We will use this function in defining the operator \( \lambda^L \). For \( L \subseteq L(<\text{inst name}>) \), \( x, y \in V \), \( a \in A \), \( i, j \in L(<\text{inst name}>) \), and \( p \in L<>(<\text{par list}>) \), the state operator \( \lambda^L \) is defined in Table 11.

\[
\begin{align*}
\lambda^L(\varepsilon) &= \varepsilon & \text{LL1} \\
\lambda^L(\delta) &= \delta & \text{LL2} \\
\lambda^L(a \cdot x) &= \delta & \text{if } a \not\in A \land E(a) \not\subseteq L & \text{LL3} \\
\lambda^L(a \cdot x) &= a \cdot \lambda^L(x) & \text{if } a \not\in A \land E(a) \subseteq L & \text{LL4} \\
\lambda^L(\text{create}(i, j, p) \cdot x) &= \delta & \text{if } i \not\in L \land j \in L & \text{LL5} \\
\lambda^L(\text{create}(i, j, p) \cdot x) &= \text{create}(i, j, p) \cdot \lambda^L[j] (x) & \text{if } i \in L \land j \not\in L & \text{LL6} \\
\lambda^L(x + y) &= \lambda^L(x) + \lambda^L(y) & \text{LL7}
\end{align*}
\]

Table 11: The state operator \( \lambda^L \)

The set \( L \) consists of all instance names that may be referred to by the events. If we encounter an event \( a \) that is not a create event then the instances referred to by \( a \) must be in the set \( L \) (LL4). If this is not the case we have a resulting deadlock (LL3). If the event to be executed is a create event then this create may only be executed if the creating instance may be referred to (LL6). After the execution of the create we have to add the name of the created instance to the set \( L \). If the create event may not be executed, the result is a deadlock (LL5).
The following examples illustrate the use of the operator $\lambda^L$:

1. $\lambda^L(\text{action}(j, at)) = \delta$
2. $\lambda^L(\text{create}(i, j, p) \parallel \text{action}(j, at))$
   \[ = \lambda^L(\text{create}(i, j, p) \cdot \text{action}(j, at) + \text{action}(j, at) \cdot \text{create}(i, j, p)) \]
   \[ = \lambda^L(\text{create}(i, j, p) \cdot \text{action}(j, at)) + \lambda^L(\text{action}(j, at) \cdot \text{create}(i, j, p)) \]
   \[ = \text{create}(i, j, p) \cdot \lambda^L(\text{action}(j, at)) + \delta = \text{create}(i, j, p) \cdot \text{action}(j, at) \]

As the initial value for the set $L$ we have to take the set of all names of instances that may be referred to initially. These are all instance names of the Message Sequence Chart except the names of the instances that are to be created. Let $L(ch)$ be an abbreviation for $\{\text{InstName}(i) \mid i \in \text{Inst}(ch)\} \setminus \text{CreatedInsts}(ch)$. Then the initial value for $L$ is $L(ch)$. We use the functions $\text{Inst}$ and $\text{CreatedInsts}$ to determine the set of instances of a chart and the set of created instances of a chart (see Appendix B).

The semantic function for Basic Message Sequence Charts extended with the process creation and process stop primitives, $S : \mathcal{L}(\langle \text{msg} \rangle) \rightarrow T(\Sigma_{PA\text{MSC}})$, is defined by

$$S[ch] = \lambda^L(ch) \left( \lambda_\theta \left( \prod_{i \in \text{Inst}(ch)} S[\text{inst}[i]] \right) \right)$$

The semantics of an instance is obtained in the same way as for Basic Message Sequence Charts. Since a process termination event may only be defined as the last event to be executed by an instance it is treated differently from all other events. The stop event is contained in the instance body part of the syntax. Therefore we add the following clause to the definition of the semantic function for instance bodies, $S[\text{body}] : \mathcal{L}(\langle \text{inst body} \rangle) \rightarrow T(\Sigma_{PA\text{MSC}})$, from the previous section.

$$S[\text{body}]_{\text{stop;}\;} = \text{stop}(iid)$$

The semantic function for events, $S_{\text{event}} : \mathcal{L}(\langle \text{event} \rangle) \rightarrow T(\Sigma_{PA\text{MSC}})$, is extended with

$$S_{\text{event}}[\text{create } \langle \text{inst name} \rangle ; i] = \text{create}(iid, \langle \text{inst name} \rangle, \langle \text{par list} \rangle)$$

$$S_{\text{event}}[\text{create } \langle \text{inst name} \rangle \ (\langle \text{par list} \rangle ); i] = \text{create}(iid, \langle \text{inst name} \rangle, \langle \text{par list} \rangle)$$

We will calculate the semantic function for the Message Sequence Chart in Figure 10. This example shows the creation and termination of instance $q$.

The textual representation of this Message Sequence Chart is the following.

```
msc creation;
  instance p;
  create q;
  in b from q;
endinstance;
instance q;
in a from r;
out b to p;
stop;
endinstance;
instance r;
out a to q;
endinstance;
endmsc;
```
Figure 10: Process creation and stop

Since no events in the Message Sequence Chart have a parameter list, we leave all empty parameter lists out. Using the techniques from the previous section we calculate that

$$
\lambda _\square \left( \| _{i \in \text{Inst}(ch)} S_{\text{inst}[i]} \right)
$$
equals

$$
create(p, q) \cdot out(r, q, a) \cdot in(r, q, a) \cdot out(q, p, b) \cdot \left( stop(q) \cdot in(q, p, b) + in(q, p, b) \cdot stop(q) \right) \\
+ out(r, q, a) \cdot \left( create(p, q) \cdot in(r, q, a) \cdot out(q, p, b) \cdot \left( stop(q) \cdot in(q, p, b) + in(q, p, b) \cdot stop(q) \right) \\
+ in(r, q, a) \cdot \left( create(p, q) \cdot out(q, p, b) \cdot \left( stop(q) \cdot in(q, p, b) + in(q, p, b) \cdot stop(q) \right) \\
+ out(q, p, b) \cdot \left( create(p, q) \cdot \left( stop(q) \cdot in(q, p, b) + in(q, p, b) \cdot stop(q) \right) \\
+ stop(q) \cdot create(p, q) \cdot in(q, p, b) \\
\right) \right)
$$

Note that this expression has an execution trace in which reference is made to an instance before it has been created. It is possible to execute $out(r, q, a)$ before $create(p, q)$. Application of the $\lambda^L$ operator solves this problem.

$$
\lambda_{\langle p,r \rangle} \left( \lambda _\square \left( \| _{i \in \text{Inst}(ch)} S_{\text{inst}[i]} \right) \right)
$$

= 

$$
create(p, q) \cdot \lambda_{\langle p,r,s \rangle}(out(r, q, a) \cdot in(r, q, a) \cdot out(q, p, b) \cdot \left( stop(q) \cdot in(q, p, b) + in(q, p, b) \cdot stop(q) \right) \\
+ \delta \\
= create(p, q) \cdot out(r, q, a) \cdot in(r, q, a) \cdot out(q, p, b) \cdot \left( stop(q) \cdot in(q, p, b) + in(q, p, b) \cdot stop(q) \right)
$$

This expression gives the semantics of the given Message Sequence Chart.
4.5 Refinement of instances

In a Message Sequence Chart an instance may be refined by decomposing it into a Sub Message Sequence Chart. The relation between the decomposed instance and the Sub Message Sequence Chart is provided by the communication events defined on the decomposed instance on the one hand and the communication events with the environment in the Sub Message Sequence Chart on the other hand. This relation is determined by means of message identifier identification. The order of the communications on the decomposed instance must be identical to the order of the corresponding communications with the exterior in the Sub Message Sequence Chart.

For a decomposed instance the semantics is the semantics of the corresponding Sub Message Sequence Chart. As a consequence we have that the semantics of a Message Sequence Chart is, in general, depending on the semantics of the Sub Message Sequence Charts given in the document. To make these Sub Message Sequence Charts available we equip the semantic function $S$ with the Message Sequence Chart document under consideration.

Furthermore, we define for $doc \in L(<msg \ doc>)$ the set $\text{ChNamed}^{doc}$ by

$$\text{ChNamed}^{doc} = \{(\text{ChName}(ch),ch) \mid ch \in \text{Charts}(doc)\}$$

The function $\text{Charts}$ defines the set of all charts in a Message Sequence Chart document (see Appendix B). Since the first component of the pairs is unique, we can look at $\text{ChNamed}^{doc}$ as a function from the Message Sequence Chart names to the Message Sequence Chart definitions in the document $doc$. From now on we will interpret $\text{ChNamed}^{doc}$ as a function.

Another consequence of taking the semantics of a decomposed instance by the semantics of the corresponding Sub Message Sequence Chart is that every reference to the decomposed instance by the non-decomposed instances of the chart must be replaced by one of the instances of the Sub Message Sequence Chart. Consider a non-decomposed instance $i$. With respect to the renaming there are five different cases to be considered. These are:

1. a non-message event is executed;
2. a message is sent to a non-decomposed instance or to the environment;
3. a message is received from a non-decomposed instance or from the environment;
4. a message is sent to a decomposed instance;
5. a message is received from a decomposed instance.

In the first three cases, there is no need for a renaming, since the events do not refer to decomposed instances. In the other two cases the name of a decomposed instance occurs, and we have to replace this name by an appropriate name from the instances of the corresponding Sub Message Sequence Chart.

In the fourth case, the sending of a message $m$ by instance $i$ to a decomposed instance $d$ is considered. By the static requirements we have that there is a message input of message $m$ specified on exactly one of the instances of Sub Message Sequence Chart $d$. The receiver instance name is given by the receiver of the corresponding message input in Sub Message Sequence Chart $d$. In the fifth case, the receiving of a message input from a decomposed instance $d$ is considered. By the static requirements we have that there is a message output of message $m$ specified on exactly one of the instances of Sub Message Sequence Chart $d$. The sender instance name is given by the sender of the corresponding message output in Sub Message Sequence Chart $d$.

Instead of tracing these sender and receiver instances in the textual representation, we perform this operation on the semantics of the Sub Message Sequence Chart corresponding with the decomposed
instance. For $m \in \mathcal{L}(\text{in} \{<\text{min}>, <\text{min}>\})$, we define a function $\text{rec}_m$ that traces the instance by which a message $m$ is received, and a function $\text{sen}_m$ that traces the instance by which a message $m$ is sent.

For $a \in A$, $x, y \in V$, $i \in \mathcal{L}(\text{inst name})$, $j \in \mathcal{L}(\text{address})$, $m, n \in \mathcal{L}(\text{in} \{<\text{min}>, <\text{min}>\})$, and $p \in \mathcal{L}(\text{par list})$, the functions $\text{rec}_m : T(\Sigma_{PA_{MSC}}) \rightarrow \mathcal{L}(\text{address})$ and $\text{sen}_m : T(\Sigma_{PA_{MSC}}) \rightarrow \mathcal{L}(\text{address})$ are defined in Table 12.

<table>
<thead>
<tr>
<th>$\text{rec}_m$</th>
<th>$\text{sen}_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{rec}_m(\varepsilon) = \text{env}$</td>
<td>$\text{sen}_m(\varepsilon) = \text{env}$</td>
</tr>
<tr>
<td>$\text{rec}_m(a \cdot x) = \text{rec}<em>m(x)$ if $a \notin A</em>{tr}$</td>
<td>$\text{sen}_m(a \cdot x) = \text{sen}<em>m(x)$ if $a \notin A</em>{es}$</td>
</tr>
<tr>
<td>$\text{rec}_m(m(j, i, u, p) \cdot x) = i$ if $n = m$</td>
<td>$\text{sen}_m(\text{out}(i, j, u, p) \cdot x) = \text{sen}_m(x)$ if $n = m$</td>
</tr>
<tr>
<td>$\text{rec}_m(m(j, i, u, p) \cdot x) = \text{rec}_m(x)$ if $n \neq m$</td>
<td>$\text{sen}_m(\text{out}(i, j, u, p) \cdot x) = \text{sen}_m(x)$ if $n \neq m$</td>
</tr>
<tr>
<td>$\text{rec}_m(x + y) = \text{rec}_m(x)$ if $\text{rec}_m(x) = \text{rec}_m(y)$</td>
<td>$\text{sen}_m(x + y) = \text{sen}_m(x)$ if $\text{sen}_m(x) = \text{sen}_m(y)$</td>
</tr>
<tr>
<td>$\text{rec}_m(x + y) = \text{env}$ if $\text{rec}_m(x) \neq \text{rec}_m(y)$</td>
<td>$\text{sen}_m(x + y) = \text{env}$ if $\text{sen}_m(x) \neq \text{sen}_m(y)$</td>
</tr>
</tbody>
</table>

Table 12: The functions $\text{rec}_m$ and $\text{sen}_m$

Given a message $m$ being sent by a non-decomposed instance $i$ to a decomposed instance $d$, we compute the receiver name by tracing the receiver instance of message $m$ in the semantics of the Sub Message Sequence Chart corresponding with decomposed instance $d$. So, the receiver is given by

$$\text{rec}_m(S_{\text{doc}}^d[\text{ChName} \text{d} \text{oc}^d(\text{InstName}(d))]$$

Analogous, given a message $m$ being received by a non-decomposed instance $i$ which is sent by a decomposed instance $d$, we compute the correct sender name by tracing the sender instance of message $m$ in the semantics of the Sub Message Sequence Chart corresponding with decomposed instance $d$. The sender is given by

$$\text{sen}_m(S_{\text{doc}}^d[\text{ChName} \text{doc}^d(\text{InstName}(d))]$$

For the renaming of the atomic actions we use the global renaming operator $\rho_f$ [BB88, BW90]. It renames an atomic action $a \in A$ into an atomic action given by $f(a)$. Let $f : A \rightarrow A$ be a function. For $x, y \in V$ and $a \in A$ we define the global renaming operator $\rho_f : T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$ in Table 13.

The semantics of a non-decomposed instance $i$ in a chart $ch$ from a document $doc$ is, with every reference to a decomposed instance name replaced by the instance name from the corresponding Sub Message Sequence Chart, given by

$$\rho_f(S_{\text{inst}[i]})$$

where $F$ is shorthand for the function $F_{\text{doc}, \text{ch}}$. The function $F_{\text{doc}, \text{ch}} : A \rightarrow A$ is, for $a \in A$, $j \in \mathcal{L}(\text{inst name})$, $k \in \mathcal{L}(\text{address})$, $m \in \mathcal{L}(\text{in} \{<\text{min}>, <\text{min}>\})$ and $p \in \mathcal{L}(\text{par list})$, given by

24
\[ p_f(\varepsilon) = \varepsilon \]
\[ p_f(\delta) = \delta \]
\[ p_f(a \cdot x) = f(a) \cdot p_f(x) \]
\[ p_f(x + y) = p_f(x) + p_f(y) \]

Table 13: The renaming operator \( p_f \)

\[
\begin{align*}
F_{doc, ch}(a) &= a & \text{if } a \notin A_s \cup A_t \\
F_{doc, ch}(out(j, k, m, p)) &= out(j, k, m, p) & \text{if } k \notin AllDecInstNames(ch) \\
F_{doc, ch}(out(j, k, m, p)) &= out(j, rec_m(S^{doc}[ChNamed^{doc}(k)]), m, p) & \text{if } k \in AllDecInstNames(ch) \\
F_{doc, ch}(in(k, j, m, p)) &= in(k, j, m, p) & \text{if } k \notin AllDecInstNames(ch) \\
F_{doc, ch}(in(k, j, m, p)) &= in(scm_m(S^{doc}[ChNamed^{doc}(k)]), j, m, p) & \text{if } k \in AllDecInstNames(ch)
\end{align*}
\]

The function \( AllDecInstNames \) determines the names of all decomposed instances in a chart (see Appendix B).

Besides the renaming discussed above, we also have to replace every occurrence of the environment in the Sub Message Sequence Chart, corresponding with a decomposed instance \( d \) of the chart under consideration, by the name of the actual sender or receiver instance. All events in the Sub Message Sequence Chart that do not refer to the environment remain unchanged. By the static requirements we know that every message sent to or received from the environment in the Sub Message Sequence Chart is related to a corresponding event defined on the decomposed instance \( d \). We use this relation in determining the instance names. There are four cases to be considered. These are:

1. A message \( m \) is sent to the environment of the Sub Message Sequence Chart and the corresponding message on the decomposed instance \( d \) is sent to a non-decomposed instance or the environment;

2. A message \( m \) is received from the environment of the Sub Message Sequence Chart and the corresponding message on the decomposed instance \( d \) is received from a non-decomposed instance or the environment;

3. A message \( m \) is sent to the environment of the Sub Message Sequence Chart and the corresponding message on the decomposed instance \( d \) is sent to a decomposed instance;

4. A message \( m \) is received from the environment of the Sub Message Sequence Chart and the corresponding message on the decomposed instance \( d \) is received from a decomposed instance.

In the first two cases we find the instance name by taking the address specification of the corresponding message event on the decomposed instance \( d \). For \( m \in \mathcal{L}(\text{msg}[\text{address}], \text{inst name}) \), \( j \in \mathcal{L}(\text{inst name}) \), and \( p \in \mathcal{L}(\text{inst name}) \), we define a function \( outpm \) that traces the receiver instance of a message output of message \( m \), and a function \( inpnm \) that traces the sender instance of a message input of message \( m \). For \( a \in A \), \( x, y \in V \), \( i \in \mathcal{L}(\text{inst name}) \), \( j \in \mathcal{L}(\text{address}) \), \( m, u \in \mathcal{L}(\text{msg}[\text{address}], \text{inst name}) \) and \( p \in \mathcal{L}(\text{par list}) \), the functions \( outpm : T(\Sigma_{PA_{MSC}}) \to \mathcal{L}(\text{address}) \) and \( inpnm : T(\Sigma_{PA_{MSC}}) \to \mathcal{L}(\text{address}) \) are defined in Table 14.
\[
\begin{align*}
\text{out}_m(\varepsilon) &= \text{env} \\
\text{out}_m(\delta) &= \text{env} \\
\text{out}_m(a \cdot x) &= \text{out}_m(x) & \text{if } a \notin A_v \cup A_{cv} \\
\text{out}_m(\text{out}(i, j, n, p, x)) &= j & \text{if } n = m \\
\text{out}_m(\text{out}(i, j, n, p, x)) &= \text{out}_m(x) & \text{if } n \neq m \\
\text{out}_m(x + y) &= \text{out}_m(x) & \text{if } \text{out}_m(x) = \text{out}_m(y) \\
\text{out}_m(x + y) &= \text{env} & \text{if } \text{out}_m(x) \neq \text{out}_m(y)
\end{align*}
\]

\[
\begin{align*}
\text{in}_m(\varepsilon) &= \text{env} \\
\text{in}_m(\delta) &= \text{env} \\
\text{in}_m(a \cdot x) &= \text{in}_m(x) & \text{if } a \notin A_t \cup A_{ct} \\
\text{in}_m(\text{in}(j, i, a, p, x)) &= j & \text{if } n = m \\
\text{in}_m(\text{in}(j, i, a, p, x)) &= \text{in}_m(x) & \text{if } n \neq m \\
\text{in}_m(x + y) &= \text{in}_m(x) & \text{if } \text{in}_m(x) = \text{in}_m(y) \\
\text{in}_m(x + y) &= \text{env} & \text{if } \text{in}_m(x) \neq \text{in}_m(y)
\end{align*}
\]

Table 14: The functions \text{out}_m and \text{in}_m

In the first case the receiver instance name is given by

\[\text{out}_m(S_{\text{inst}}[d])\]

and, in the second case, the sender name is given by

\[\text{in}_m(S_{\text{inst}}[d])\]

For the third and fourth case things get more complicated. Consider a message \(m\) being sent by an instance \(k\) of the Sub Message Sequence Chart to the environment. Suppose that the decomposed instance \(d\) sends the message \(m\) to a decomposed instance. The name of the decomposed instance that receives the message \(m\) from decomposed instance \(d\) is then given by \(\text{out}_m(S_{\text{inst}}[d])\). The receiver name of message \(m\) is not given by the name of the decomposed instance it is sent to, but by one of the instances of the Sub Message Sequence Chart that refines the decomposed instance \(\text{out}_m(S_{\text{inst}}[d])\). Therefore, we trace the instance name in the Sub Message Sequence Chart corresponding to the decomposed instance \(\text{out}_m(S_{\text{inst}}[d])\). The receiver instance name is therefore given by

\[\text{rec}_m(S^{\text{doc}}[\text{ChName}^{\text{doc}}(\text{out}_m(S_{\text{inst}}[d]))])\]

Consider a message \(m\) being received by an instance \(k\) of the Sub Message Sequence Chart from the environment. Suppose that the decomposed instance \(d\) receives the message from a decomposed instance. The name of the decomposed instance that sends the message \(m\) to the decomposed instance \(d\) is given by \(\text{in}_m(S_{\text{inst}}[d])\). The instance name of message \(m\) is not given by the name of the decomposed instance it is received from, but by one of the instances of the Sub Message Sequence Chart that refines the decomposed instance \(\text{in}_m(S_{\text{inst}}[d])\). Therefore, we trace the instance name in the Sub Message Sequence Chart corresponding with the decomposed instance \(\text{in}_m(S_{\text{inst}}[d])\). The sender instance name is therefore given by

\[\text{sen}_m(S^{\text{doc}}[\text{ChName}^{\text{doc}}(\text{in}_m(S_{\text{inst}}[d]))])\]

As a result we have that the semantics of a decomposed instance \(d\) in a chart \(ch\) from a document \(doc\) is, with all occurrences of the environment as address specification for a message event replaced
by the instance name, given by

\[ p_G(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(d))) \]

where \( G \) is shorthand for \( G_{\text{doc},ch,d} \). The function \( G_{\text{doc},ch,d} : A \to A \) is, for \( a \in A, j \in \mathcal{L}(<\text{name}>), m \in \mathcal{L}(<\text{min}>[,<\text{min}>]) \) and \( p \in \mathcal{L}<\langle \text{par list} \rangle > \), given by

\[
G_{\text{doc},ch,d}(a) = a
\]

\[
G_{\text{doc},ch,d}(\text{out}(j, env, m, p)) = \text{out}(j, \text{out}_m(S_{\text{inst}}[d]), m, p)
\]

if \( a \not\in A_e \cup A_s \)

\[
G_{\text{doc},ch,d}(\text{out}(j, env, m, p)) = \text{out}(j, \text{rec}_m(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(d))), m, p)
\]

if \( \text{out}_m(S_{\text{inst}}[d]) \not\in \text{AllDecInstNames}(ch) \)

\[
G_{\text{doc},ch,d}(\text{in}(env, j, m, p)) = \text{in}(\text{in}_m(S_{\text{inst}}[d]), j, m, p)
\]

if \( \text{in}_m(S_{\text{inst}}[d]) \not\in \text{AllDecInstNames}(ch) \)

\[
G_{\text{doc},ch,d}(\text{in}(env, j, m, p)) = \text{in}(\text{in}_m(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(d))), j, m, p)
\]

if \( \text{in}_m(S_{\text{inst}}[d]) \not\in \text{AllDecInstNames}(ch) \)

We obtain the semantics of the complete chart by placing the semantics of the instances, with the renamings discussed above, in parallel and applying the state operators \( \lambda_M \) and \( \lambda^L \) as in the previous sections. The function \( \text{DecInst} \) gives the instance definitions of all decomposed instances of the chart it is applied to (see Appendix B). This gives the following recursive definition of \( S_{\text{doc}}^{\text{ChNamed}} \). The recursion is finite by the static requirement that expresses that an instance may not be refined by the chart it is defined in, directly or via a number of refinements.

For \( \text{doc} \in \mathcal{L}(<\text{msc doc}> \setminus <\text{submsc}>) \) the semantic function, \( S_{\text{doc}}^{\text{ChNamed}} : \mathcal{L}(<\text{submsc}> \setminus <\text{submsc}>) \to T(\Sigma_P A_{MSC}) \), is for all \( \text{ch} \in \text{Charts}(\text{doc}) \) defined by

\[
S_{\text{doc}}\{\text{ch}\} = \lambda^L(\text{ch}) \left( \left( \big\| \begin{array}{c} \text{if } a \not\in A_e \cup A_s \big\| \\
\text{if } k \not\in \text{AllDecInstNames}(\text{ch}) \big\| \\
\text{if } k \not\in A_{\text{ch}} \big\| \right) \right)
\]

where \( F : A \to A \) is, for \( a \in A, j \in \mathcal{L}(<\text{name}>), k \in \mathcal{L}(<\text{address}>), m \in \mathcal{L}(<\text{min}>[,<\text{min}>]) \) and \( p \in \mathcal{L}<\langle \text{par list} \rangle > \), given by

\[
F(a) = a
\]

\[
F(\text{out}(j, k, m, p)) = \text{out}(j, k, m, p)
\]

if \( a \not\in A_e \cup A_s \)

\[
F(\text{out}(j, k, m, p)) = \text{out}(j, \text{rec}_m(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(k)))), m, p)
\]

if \( k \not\in \text{AllDecInstNames}(\text{ch}) \)

\[
F(\text{in}(k, j, m, p)) = \text{in}(k, j, m, p)
\]

if \( k \not\in A_{\text{ch}} \)

\[
F(\text{in}(k, j, m, p)) = \text{in}(\text{in}_m(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(k))), j, m, p)
\]

if \( k \not\in \text{AllDecInstNames}(\text{ch}) \)

and where \( G : A \to A \) is, for \( a \in A, j \in \mathcal{L}(<\text{name}>), m \in \mathcal{L}(<\text{min}>[,<\text{min}>]) \) and \( p \in \mathcal{L}<\langle \text{par list} \rangle > \), given by

\[
G(a) = a
\]

\[
G(\text{out}(j, env, m, p)) = \text{out}(j, \text{out}_m(S_{\text{inst}}[d]), m, p)
\]

if \( a \not\in A_e \cup A_s \)

\[
G(\text{out}(j, env, m, p)) = \text{out}(j, \text{rec}_m(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(d))), m, p)
\]

if \( \text{out}_m(S_{\text{inst}}[d]) \not\in \text{AllDecInstNames}(\text{ch}) \)

\[
G(\text{out}(j, env, m, p)) = \text{out}(j, \text{rec}_m(S_{\text{doc}}^{\text{ChNamed}}(\text{Name}(d))), m, p)
\]

if \( \text{out}_m(S_{\text{inst}}[d]) \not\in \text{AllDecInstNames}(\text{ch}) \)
\[
G(\text{in}(\text{env}, j, m, p)) = \text{in}(\text{inp}_m(S_{\text{inst}}[d]), j, m, p)
\]

\[
G(\text{in}(\text{env}, j, m, p)) = \text{in}(\text{scn}_m(S_{\text{doc}}[\text{ChNamed}^{\text{doc}}(\text{inp}_m(S_{\text{inst}}[d]))]), i, m, p)
\]

if \(\text{inp}_m(S_{\text{inst}}[d]) \notin \text{AllDecInstNames}(ch)\)

if \(\text{inp}_m(S_{\text{inst}}[d]) \in \text{AllDecInstNames}(ch)\)

The function \(\text{AllDecInstNames}\) determines the names of all decomposed instances in a chart (see Appendix B). The other semantic functions remain unchanged.

In Figure 11 a Message Sequence Chart with a decomposed instance \(d\) and a Sub Message Sequence Chart that refines the decomposed instance are given.

![Figure 11: Refinement of decomposed instances](image)

In the textual representation these charts are represented by

```plaintext
msc decinst;
  instance i;
    out m to d;
  endinstance;
  instance d decomposed;
    in m from i;
    out n to env;
  endinstance;
endmsc;

submsc d;
  instance j;
    in m from env;
    out o to k;
  endinstance;
  instance k;
    in o from j;
    out n to env;
  endinstance;
endsubmsc;
```

First, we compute the semantics of the Sub Message Sequence Chart. Since the Sub Message Sequence Chart only contains non-decomposed instances, we can determine the semantics with the techniques from the previous sections. Moreover, since the Sub Message Sequence Charts contains no create events, we can discard the state operator \(\lambda^b\). The semantics of the Sub Message Sequence Chart is given by the following expression

\[
\text{in}(\text{env}, j, m, <>) \cdot \text{out}(j, k, o, <>) \cdot \text{in}(j, k, o, <>) \cdot \text{out}(k, \text{env}, n, <>)
\]

The semantics of instance \(i\) is given by \(\text{out}(i, d, m, <>)\). Then, the semantics of Message Sequence Chart \(\text{decinst}\) is given by the following expression

\[
\lambda_{\text{decinst}}(\rho_\phi(\text{out}(i, d, m, <>)) ||
\rho_\phi(\text{in}(\text{env}, j, m, <>) \cdot \text{out}(j, k, o, <>) \cdot \text{in}(j, k, o, <>) \cdot \text{out}(k, \text{env}, n, <>)))
\]

where the renaming functions \(F\) and \(G\) are given by
\[ F(o_{lt}(i,d,m,<>)) = \text{out}(i,j,m,<>) \]
and
\[ G(i_{n}(\text{env},j,m,<>)) = i_{n}(i,j,m,<>) \]
\[ G(\text{out}(j,k,o,<>)) = \text{out}(j,k,o,<>) \]
\[ G(\text{in}(j,k,o,<>)) = \text{in}(j,k,o,<>) \]
\[ G(\text{out}(k,\text{env},n,<>)) = \text{out}(k,\text{env},n,<>) \]

So, the semantics is given by the following expression
\[
\lambda^{MSC}(\lambda^{\mathcal{E}}(\text{out}(i,j,m,<>)) \parallel (\text{in}(i,j,m,<>) \cdot \text{out}(j,k,o,<>) \cdot \text{in}(j,k,o,<>) \cdot \text{out}(k,\text{env},n,<>)))
\]
which equals
\[
\text{out}(i,j,m,<>) \cdot \text{in}(i,j,m,<>) \cdot \text{out}(j,k,o,<>) \cdot \text{in}(j,k,o,<>) \cdot \text{out}(k,\text{env},n,<>)
\]

In Figure 12 we present a Message Sequence Chart without decomposed instances. This Message Sequence Chart is semantically equivalent to the Message Sequence Chart in Figure 11.

![MSC equivalent](image)

Figure 12: Semantically equivalent Message Sequence Chart

### 4.6 Timer handling

In Message Sequence Charts either the setting of a timer and a subsequent timeout due to timer expiration or the setting of a timer and a subsequent timer reset may be specified. In the semantics they are treated in the same way as (local) actions. Because a Message Sequence Chart only expresses the relative order of events, the duration is regarded just as a label without special meaning. We will include the timer events in the semantics by extending the semantic function for events with

\[
S^{\text{event}}_{\text{set}}[\text{set } <\text{tn}>[],<\text{tin}>;] = \text{set}(i_{id},<\text{tn}>[],<\text{tin}>[,<\text{dn}>])
\]

\[
S^{\text{event}}_{\text{set}}[\text{set } <\text{tn}>[],<\text{tin}>;] = \text{set}(i_{id},<\text{tn}>[],<\text{tin}>[])
\]

\[
S^{\text{event}}_{\text{reset}}[\text{reset } <\text{tn}>[],<\text{tin}>;] = \text{reset}(i_{id},<\text{tn}>[],<\text{tin}>[])
\]

\[
S^{\text{event}}_{\text{reset}}[\text{timeout } <\text{tn}>[],<\text{tin}>;] = \text{timeout}(i_{id},<\text{tn}>[],<\text{tin}>[])
\]

We will illustrate the semantics of Message Sequence Charts with timer handling with the chart in Figure 13. For instance \( i \) the setting of a timer \( T \) with duration \( d \) and the subsequent resetting of the timer is specified.
In the textual representation this chart is represented by

```plaintext
msc timer;
  instance i;
    set T(d);
    out m to j;
    reset T;
  endinstance;
  instance j;
    in m from i;
  endinstance;
endmsc;
```

Then we have for the semantics of the instances

\[
S_{\text{inst}[i]} = \text{set}(i, T, d) \cdot \text{out}(i, j, m, \langle\rangle) \cdot \text{reset}(i, T)
\]
\[
S_{\text{inst}[j]} = \text{in}(i, j, m, \langle\rangle)
\]

Since the chart has no created instances and no decomposed instances, we calculate the semantics as follows:

\[
\lambda_\varphi((\text{set}(i, T, d) \cdot \text{out}(i, j, m, \langle\rangle) \cdot \text{reset}(i, T)) || (\text{in}(i, j, m, \langle\rangle)))
\]
which equals

\[
\text{set}(i, T, d) \cdot \text{out}(i, j, m, \langle\rangle) \cdot (\text{reset}(i, T) \cdot \text{in}(i, j, m, \langle\rangle) + \text{in}(i, j, m, \langle\rangle) \cdot \text{reset}(i, T))
\]

### 4.7 Coregions

So far the events defined on an instance are totally ordered in time. In order to specify unordered events on an instance the coregion is introduced. Within a coregion only message events may be specified. The events that are defined within a coregion can be executed in any order. So the semantics of the coregion is the free merge of all the events defined within the coregion.

We extend the semantic function for events with

\[
S_{\text{event[coregion]}} = \bigoplus_{e \in \text{Corevents[coregion]}} S_{\text{event[e]}}
\]
where the function \( \text{CoEvents} \) associates to a coregion the set of all message events defined in the coregion (see Appendix B).

Consider the Message Sequence Chart in Figure 14. It consists of two instances which exchange the messages \( m \) and \( n \). The sending of the messages \( m \) and \( n \) is unordered, both orderings are possible. The consumption of the messages by instance \( i2 \) must take place in the specified order. The restriction that a message output must be executed before its corresponding message input remains valid.

![Message Sequence Chart with a coregion](image)

**Figure 14:** Message Sequence Chart with a coregion

In the textual representation this chart is given by either one of the two descriptions below.

```plaintext
msc coregion;
    instance i1;
        concurrent
            out m to i2;
            out n to i2;
        endconcurrent;
    endinstance;
    instance i2;
        in m from i1;
        in n from i1;
    endinstance;
endmsc;
```

```plaintext
msc coregion;
    instance i1;
        concurrent
            out n to i2;
            out m to i2;
        endconcurrent;
    endinstance;
    instance i2;
        in m from i1;
        in n from i1;
    endinstance;
endmsc;
```

On instance \( i1 \) only a coregion is specified, so the semantics of the instance equals the semantics of the coregion. The semantics of instance \( i1 \) is therefore given by

\[
S_{\text{inst}}[i1] = \text{out}(i1, i2, m, \triangleright) || \text{out}(i1, i2, n, \triangleright)
\]

and the semantics of instance \( i2 \) is given by

\[
S_{\text{inst}}[i2] = \text{in}(i1, i2, m, \triangleright) \cdot \text{in}(i1, i2, n, \triangleright)
\]

Because the chart does not contain decomposed instances and created instances, we may discard the renaming operators \( \rho_F \) and \( \rho_G \) and the state operator \( \lambda^L \). The semantics of the complete chart is given by the following expression

\[
\lambda_o((\text{out}(i1, i2, m, \triangleright) || \text{out}(i1, i2, n, \triangleright)) || (\text{in}(i1, i2, m, \triangleright) \cdot \text{in}(i1, i2, n, \triangleright)))
\]

which equals
4.8 Conditions

A condition is a construct that is used to impose restrictions on the composition of Message Sequence Charts. Therefore conditions play an important role in the static semantics. With a condition no dynamic behavior is associated. Thus we come to the following extension of the semantic function for events.

\[ S_{\text{event}}^{\text{cond}}[\text{condition}] = \varepsilon \]

Note that the semantics of a chart containing conditions is simply the semantics of the chart with the conditions deleted from it.

4.9 Overview of the complete semantics

In this section we give an overview of the semantics of the complete language. The process algebra operators that are used can be found in the previous sections and the auxiliary functions are given in Appendix B.

For \( doc \in \mathcal{L}(\text{msg doc}) \) the semantic function, \( S^{doc} : \mathcal{L}(\text{msg} \mid \text{submsg}) \rightarrow T(\Sigma_{PA_{MSC}}) \), is for all \( ch \in \text{Charts}(doc) \) defined by

\[
S^{doc}[ch] = \lambda L(ch) \left( \lambda \varnothing \left( \left( \| i \in \text{Inst}(ch) \right) \rightarrow \right) \left( \| d \in \text{DecInst}(ch) \rightarrow \right) \left( S^{doc}[\text{ChNamed}^{doc}(\text{InstName}(d))] \right) \right)
\]

where \( F : A \rightarrow A \) is, for \( a \in A, j \in \mathcal{L}(\text{inst name}), k \in \mathcal{L}(\text{address}), m \in \mathcal{L}(\text{mn}[,<\text{min}>]) \) and \( p \in \mathcal{L}_{\text{par}}(\text{par list}) \), given by

\[
F(a) = a
\]
\[
F(\text{out}(j, k, m, p)) = \text{out}(j, k, m, p)
\]
\[
F(\text{out}(j, k, m, p)) = \text{out}(j, \text{rec}(S^{doc}[\text{ChNamed}^{doc}(k)]), m, p)
\]
\[
F(\text{in}(k, j, m, p)) = \text{in}(k, j, m, p)
\]
\[
F(\text{in}(k, j, m, p)) = \text{in}(\text{seq}(S^{doc}[\text{ChNamed}^{doc}(k)]), j, m, p)
\]

and where \( G : A \rightarrow A \) is, for \( a \in A, j \in \mathcal{L}(\text{inst name}), m \in \mathcal{L}(\text{mn}[,<\text{min}>]) \) and \( p \in \mathcal{L}_{\text{par}}(\text{par list}) \), given by

\[
G(a) = a
\]
\[
G(\text{out}(j, env, m, p)) = \text{out}(j, \text{out}(S_{\text{inst}}[d]), m, p)
\]

and where \( A_{er} \subseteq A \).
\[ G(\text{out}(j, \text{env}, m, p)) = \text{out}(j, \text{recv}(S^\text{doc} [\text{ChNamed}^\text{doc}(\text{outp}_m(S_{\text{inst}}[d]))], m, p) \]
\[ \quad \text{if } \text{outp}_m(S_{\text{inst}}[d]) \in \text{AllDecInstNames}(ch) \]
\[ G(\text{in}(\text{env}, j, m, p)) = \text{in}(\text{inp}_m(S_{\text{inst}}[d]), j, m, p) \]
\[ \quad \text{if } \text{inp}_m(S_{\text{inst}}[d]) \in \text{AllDecInstNames}(ch) \]
\[ G(\text{in}(\text{env}, j, m, p)) = \text{in}(\text{send}_m(S^\text{doc} [\text{ChNamed}^\text{doc}(\text{inp}_m(S_{\text{inst}}[d]))], i, m, p) \]
\[ \quad \text{if } \text{inp}_m(S_{\text{inst}}[d]) \in \text{AllDecInstNames}(ch) \]

On an instance a number of communication events and actions are defined. The order in which they appear in the textual representation of the Message Sequence Chart is the order in which they are to be executed. The semantic function for non-decomposed instances, \( S_{\text{inst}} : \mathcal{L}(\text{inst def}) \rightarrow T(\Sigma_{PA_{MSC}}) \), is defined by

\[ S_{\text{inst}}[i] = S_{\text{body}}^{\text{InstName}(i)}[\text{InstBody}(i)] \]

Since in the textual description the instance on which the events are defined is not specified explicitly, we label the semantic function for the instance body with the name of the instance it originates from. For \( iid \in \mathcal{L}(\text{inst name}) \) the semantic function for instance bodies, \( S_{\text{body}}^{\text{iid}} : \mathcal{L}(\text{inst body}) \rightarrow T(\Sigma_{PA_{MSC}}) \), is defined by

\[ S_{\text{body}}^{\text{iid}}[\text{stop}] = \text{stop}(iid) \]

\[ S_{\text{body}}^{\text{iid}}[\text{event} < \text{inst body}>] = S_{\text{event}}^{\text{iid}}[\text{event}] \cdot S_{\text{body}}^{\text{iid}}[\text{inst body}] \]

The semantic function for events gives the semantics of one event in isolation. It is merely a translation of the smallest components of Message Sequence Charts into the atomic actions of the process algebra. For \( iid \in \mathcal{L}(\text{inst name}) \) the semantic function for events, \( S_{\text{event}}^{\text{iid}} : \mathcal{L}(\text{event}) \rightarrow T(\Sigma_{PA_{MSC}}) \), is defined by

\[ S_{\text{event}}^{\text{iid}}[\text{action at}] = \text{action}(iid, at) \]

\[ S_{\text{event}}^{\text{iid}}[\text{out} < \text{mn}, <\min超越> \text{to address}>] = \text{out}(iid, \text{address}, <\text{mn}, <\min>, <>) \]

\[ S_{\text{event}}^{\text{iid}}[\text{out} < \text{mn}, <\min超越> \text{par list} \text{to address}>] = \text{out}(iid, \text{address}, <\text{mn}, <\min>, \text{par list}) \]

\[ S_{\text{event}}^{\text{iid}}[\text{in} < \text{mn}, <\min超越> \text{from address}>] = \text{in}(\text{address}, iid, <\text{mn}, <\min>, <>) \]

\[ S_{\text{event}}^{\text{iid}}[\text{in} < \text{mn}, <\min超越> \text{par list} \text{from address}>] = \text{in}(\text{address}, iid, <\text{mn}, <\min>, \text{par list}) \]

\[ S_{\text{event}}^{\text{iid}}[\text{create inst name}>] = \text{create}(iid, <\text{inst name}, <>) \]

\[ S_{\text{event}}^{\text{iid}}[\text{create inst name} \text{par list}>] = \text{create}(iid, <\text{inst name}, \text{par list}) \]

\[ S_{\text{event}}^{\text{iid}}[\text{set tna}, <\tin>, <\dn> >] = \text{set}(iid, <\text{tna}, <\tin>, <\dn>) \]

\[ S_{\text{event}}^{\text{iid}}[\text{set tna}, <\tin]> = \text{set}(iid, <\text{tna}, <\tin>) \]

\[ S_{\text{event}}^{\text{iid}}[\text{reset tna}, <\tin> >] = \text{reset}(iid, <\text{tna}, <\tin>) \]
4.10 Process algebra \(PA_{MSC}\)

In the previous sections we have specified the set of atomic actions \(A\) and extended the process algebra \(PA_{\epsilon}\) with the state operators \(\lambda_M\) and \(\lambda^L\) and the global renaming operators \(\rho_F\) and \(\rho_G\). The resulting process algebra will be called \(PA_{MSC}\).

In this section we consider the process algebra \(PA_{MSC}\). We show that the induction structure for \(PA_{MSC}\) equals the induction structure for \(PA_{\epsilon}\). Furthermore, we show that \(PA_{MSC}\) is a conservative extension of \(PA_{\epsilon}\). This means that we have not introduced any new processes (from a semantic point of view) and that we have not added any new identities between existing processes.

**Theorem 4.10.1 (Elimination)** For every closed \(PA_{MSC}\) term \(s\), there exists a basic \(PA_{\epsilon}\) term \(t\) such that

\[
PA_{MSC} \vdash s = t
\]

**Proof.** We can associate a term rewriting system \([\text{Klo92}]\) with the axioms from \(PA_{MSC}\) by replacing the \(=\) by \(\equiv\) in all axioms except \(A1\) and \(A2\). Using term-rewrite techniques \([\text{AB91}, \text{Der79}]\) it can be proven that this term-rewriting system is strongly normalizing (i.e. there are no infinite rewritings possible). Furthermore it can be proven that a normal form of a closed \(PA_{MSC}\) term is a closed \(PA_{\epsilon}\) term.

Let \(s\) be a closed \(PA_{MSC}\) term. Since the term-rewrite system associated with \(PA_{MSC}\) is strongly normalizing, there must exist a normal form \(t\) such that there is a reduction from \(s\) to \(t\) in the term-rewrite system. The proof of the reduction from \(s\) to \(t\) in the term-rewrite system gives us a proof of \(PA_{MSC} \vdash s = t\). We also have that \(t\) is a basic \(PA_{\epsilon}\) term.

**Theorem 4.10.2 (Conservative Extension)** For all closed \(PA_{\epsilon}\) terms \(s\) and \(t\) such that \(PA_{MSC} \vdash s = t\) we have

\[
PA_{\epsilon} \vdash s = t
\]

**Proof.** If we consider the term-rewrite system associated with \(PA_{MSC}\), we see that no rewrite rule introduces one of the operators \(\lambda_M\), \(\lambda^L\) or \(\rho_F\) in the right-hand side if no such operator was already present in the left-hand side of the (instantiation of the) rewrite rule. Since the term-rewrite system associated with \(PA_{MSC}\) is strongly normalizing and since a normal form of a closed \(PA_{MSC}\) term must be a basic \(PA_{\epsilon}\) term, we have reductions \(s \rightarrow s'\) and \(t \rightarrow t'\) with \(s'\) and \(t'\) basic \(PA_{\epsilon}\) terms. Both \(s\) and \(t\) do not contain any of the operators \(\lambda_M\), \(\lambda^L\) or \(\rho_F\), so the reductions \(s \rightarrow s'\) and \(t \rightarrow t'\) are instances of the rewrite rules of the term-rewrite system associated with \(PA_{\epsilon}\). Therefore, we have \(PA_{\epsilon} \vdash s = s'\) and \(PA_{\epsilon} \vdash t = t'\). Since the term-rewrite system associated with \(PA_{MSC}\) is confluent modulo \(A1\) and \(A2\) we have that \(s'\) and \(t'\) are identical except for the order of summands. Therefore, we also have \(PA_{\epsilon} \vdash s = t\).
Conclusions

In this paper we developed a formal algebraic semantics of Message Sequence Charts. We used the textual representation of Message Sequence Charts for the definition of the semantics. We started with the semantics of Basic Message Sequence Charts, the core language of Message Sequence Charts. Since this fragment of the complete language only incorporates those primitives that can be found in most languages comparable to Message Sequence Charts, it has become possible to relate Message Sequence Charts to those graphical or textual specification languages.

We extended the semantics of Basic Message Sequence Charts to the semantics of the complete language of Message Sequence Charts by adding the other primitives one by one. This approach results in a definition of the semantics that is more accessible than a definition of the semantics with respect to the complete language at one go.

For most primitives the extension of the semantics is straightforward. Only the semantics of the instance refinement primitive is not easily captured. We decided to compute the refinement from the semantics, whereas another choice would have been to compute the refinement syntactically. In both methods we need to "compute" the renaming on atomic actions as imposed by the instance refinement principle. The only difference is the level on which the computation takes place: on syntax or on semantics. In both cases we have to search for the right names of the sender and receiver of messages. The use of process algebra in the formal semantics has, as can be expected, influenced the definition of the semantics. This is best illustrated by the state operators we used to remove some traces from the interleaving of the instances.

The algebraic approach from process algebra enables the use of term rewriting systems for the rapid prototyping of specifications [MW93]. The definition of a formal semantics of Message Sequence Charts based on process algebra has turned out to be a natural and successful method.

References


A Concrete textual syntax

A.1 The concrete textual syntax from Z.120

<lexical unit> ::= <word> | <character string>
    | <special> | <composite special>
    | <note> | <keyword>
    | <semicolon>

<keyword> ::= action | all
    | block | comment
    | concurrent | condition
    | create | decomposed
    | endconcurrent | endinstance
    | endmsc | endmscdocument
    | endsubmsc | endtext
    | env | from
    | inst | instance
    | msc | mscdocument
    | in | out
    | process | referenced
    | related to | reset
    | service | set
    | shared | stop
    | submsc | system
    | text | timeout
    | to

<text> ::= {<alphanumeric> | <other character> | <special>
    | <full stop> | <underline> | <space> | <apostrophe>
    | <other character>}
<special> ::= + | - | % | ! | | > | * | ( | ) | " | , | = | :
<semicolon> ::= <end>
<note> ::= /* <text> */
<end> ::= [ <comment> ] <semicolon>
<comment> ::= <note> | comment <character string>
<text definition> ::= text <note> endtext <end>
<message sequence chart document> ::= mscdocument
   <document head>
   <document body>
endmscdocument
<end>

<document head> ::= <msc document name>
   [related to <sdl reference>]
<end>

<sdl reference> ::= <sdl document identifier>
<sdl document identifier> ::= [<qualifier>] <name>
<qualifier> ::= <path item> {/<path item>}
<path item> ::= <scope unit class> <name>
<scope unit class> ::= system | block | process
<document body> ::= { <message sequence chart> | <submsc>
   | <msc diagram> | <submsc diagram> }

<message sequence chart> ::= msc <msc head>
   <msc body>
endmsc <end>

<msc head> ::= <message sequence chart name> <end>
   [<msc interface>]
<msc interface> ::= inst <instance list> <end>
<instance list> ::= <instance name> [:<instance kind>]
   [,<instance list>]
<instance kind> ::= [<kind denominator>] <kind name>
<kind denominator> ::= system | block | process | service
<msc body> ::= { <instance definition>
   | <text definition>
   }

<instance definition> ::= instance <instance head>
   <instance body>
endinstance <end>

<instance head> ::= <instance name>
   [:[:] <instance kind>]
   [decomposed] <end>
<instance body> ::= <instance event list> [stop]
<instance event list> ::= { <message input>
   | <create>
   | <timer statement>
   | <coregion>
   | <action>
   | <condition>
   }*

<message input> ::= in <msg identification>
   from <address> <end>
<message output> ::= out <msg identification>
   to <address> <end>
<msg identification> ::= <message name>
   [,<message instance name>]
   [([<parameter list>]]
<parameter list> ::= [<parameter name>
   [,<parameter list>]]
Because of the long names for the nonterminals, the concrete textual syntax is not suited for manipulation as for example in the definition of the semantic functions in Section 4 and the auxiliary functions in Appendix B. Therefore, we will introduce abbreviations for the nonterminals in the above concrete textual grammar. The abbreviations are given below.

<table>
<thead>
<tr>
<th>nonterminal</th>
<th>abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;message sequence chart document&gt;</td>
<td>&lt;msc doc&gt;</td>
</tr>
<tr>
<td>&lt;document head&gt;</td>
<td>&lt;doc head&gt;</td>
</tr>
<tr>
<td>&lt;document body&gt;</td>
<td>&lt;doc body&gt;</td>
</tr>
<tr>
<td>&lt;msc document name&gt;</td>
<td>&lt;doc name&gt;</td>
</tr>
<tr>
<td>&lt;sdl reference&gt;</td>
<td>&lt;sdl ref&gt;</td>
</tr>
<tr>
<td>&lt;sdl document identifier&gt;</td>
<td>&lt;sdl docid&gt;</td>
</tr>
<tr>
<td>&lt;document body&gt;</td>
<td>&lt;doc body&gt;</td>
</tr>
<tr>
<td>&lt;message sequence chart&gt;</td>
<td>&lt;msg&gt;</td>
</tr>
<tr>
<td>&lt;message sequence chart name&gt;</td>
<td>&lt;msg name&gt;</td>
</tr>
<tr>
<td>&lt;instance list&gt;</td>
<td>&lt;inst list&gt;</td>
</tr>
<tr>
<td>&lt;instance name&gt;</td>
<td>&lt;inst name&gt;</td>
</tr>
<tr>
<td>&lt;instance kind&gt;</td>
<td>&lt;inst kind&gt;</td>
</tr>
<tr>
<td>&lt;instance definition&gt;</td>
<td>&lt;inst def&gt;</td>
</tr>
<tr>
<td>&lt;text definition&gt;</td>
<td>&lt;text def&gt;</td>
</tr>
<tr>
<td>&lt;instance head&gt;</td>
<td>&lt;inst head&gt;</td>
</tr>
<tr>
<td>&lt;instance body&gt;</td>
<td>&lt;inst body&gt;</td>
</tr>
</tbody>
</table>
The result of introducing these abbreviations is:

[01] <msc doc> ::= mscdocument <doc head>
    <doc body>
    endmscdocument <end>
[02] <doc head> ::= <doc name> [ related to <sdl ref> ] <end>
[03] <sdl ref> ::= <sdl docid>
[04] <sdl docid> ::= [ <qualifier> ] <name>
[05] <qualifier> ::= <path item> { / <path item> }
[06] <path item> ::= <scope unit class> <name>
[07] <scope unit class> ::= system | block | process
[08] <doc body> ::= { <msc>
    | <submsc>
    | <msc diagram>
    | <submsc diagram>
  }*
[09] <msc> ::= msc <msc head>
    <msc body>
    endmsc <end>
[10] <msc head> ::= <msc name> <end> [ <msc interface> ]
[11] <msc interface> ::= inst <inst list> <end>
[12] <inst list> ::= <inst name> [ : <inst kind> ] [,<inst list> ]
[13] <inst kind> ::= [ <type> ] <kind name>
[14] <type> ::= system | block | process | service
[15] <msc body> ::= { <inst def> | <text def> }*
[16] <inst def> ::= instance <inst head>
    <inst body>
    endinstance <end>
[17] <inst head> ::= <inst name> [ [ : ] <inst kind> ]
    [ decomposed ] <end>
[18] <inst body> ::= <inst event list> [ <stop> ]
[19] <inst event list> ::= { <in>
    | <out>
    | <create>
    | <timer stat>
The concrete textual syntax for the semantics

The original grammar for Message Sequence Charts from the previous section is not suited for the inductive definition of functions on nonterminals. Therefore, we give an equivalent grammar (i.e. a grammar that generates the same language with respect to the nonterminal <msc doc>) which does not contain the *.

Since some syntactic parts of a Message Sequence Chart are of no importance to the semantics, we leave those out. These are the references to SDL, the Message Sequence Chart interface, the instance kinds and the text definitions. We also leave out the nonterminals <msc diagram> and <submsc diagram> since we don’t intend to define a formal semantics on the graphical representation. Moreover, we replace some nonterminals by their right-hand sides.

The resulting grammar is then:

```
[01]  <msc doc> ::= mscdocument <doc name>; 
        <doc body> 
        endmscdocument; 
[02]  <doc body> ::= <> | <msc> <doc body> | <submsc> <doc body> 
[03]  <msc> ::= msc <msc name>; <msc body> endmsc; 
[04]  <msc body> ::= <> | <inst def> <msc body> 
[05]  <inst def> ::= instance <inst name> [decomposed];
        <inst body> 
        endinstance; 
[06]  <inst body> ::= <> [stop:] | <event> <inst body> 
[07]  <event> ::= <out> | <in> | <action> | <create> | <set> | 
```
B Auxiliary functions for the semantics of Message Sequence Charts

In this appendix we define the auxiliary functions used in defining the semantics of Message Sequence Charts in Section 4.

\[
\begin{align*}
\text{DocBody} & : \mathcal{L}(\text{msc doc}) \to \mathcal{L}(\text{doc body}) \\
\text{DocBody}(\text{msc document } \langle\text{doc name}\rangle; \langle\text{doc body}\rangle \text{ endmsc document;} ) & = \langle\text{doc body}\rangle \\
\text{ChName} & : \mathcal{L}(\text{msc} \mid \text{submsc}) \to \mathcal{L}(\text{msc name}) \\
\text{ChName}(\text{msc } \langle\text{msc name}\rangle; \langle\text{msc body}\rangle \text{ endmsc;} ) & = \langle\text{msc name}\rangle \\
\text{ChName}(\text{submsc } \langle\text{msc name}\rangle; \langle\text{msc body}\rangle \text{ endsubmsc;} ) & = \langle\text{msc name}\rangle \\
\text{ChBody} & : \mathcal{L}(\text{msc} \mid \text{submsc}) \to \mathcal{L}(\text{msc body}) \\
\text{ChBody}(\text{msc } \langle\text{msc name}\rangle; \langle\text{msc body}\rangle \text{ endmsc;} ) & = \langle\text{msc body}\rangle \\
\text{ChBody}(\text{submsc } \langle\text{msc name}\rangle; \langle\text{msc body}\rangle \text{ endsubmsc;} ) & = \langle\text{msc body}\rangle \\
\text{InstName} & : \mathcal{L}(\text{inst def}) \to \mathcal{L}(\text{inst name}) \\
\text{InstName}(\text{instance } \langle\text{inst name}\rangle [\text{decomposed}]; \langle\text{inst body}\rangle \text{ endinstance;} ) & = \langle\text{inst name}\rangle \\
\text{InstBody} & : \mathcal{L}(\text{inst def}) \to \mathcal{L}(\text{inst body}) \\
\text{InstBody}(\text{instance } \langle\text{inst name}\rangle [\text{decomposed}]; \langle\text{inst body}\rangle \text{ endinstance;} ) & = \langle\text{inst body}\rangle \\
\text{Charts} & : \mathcal{L}(\text{doc body}) \to \mathcal{H}(\mathcal{L}(\text{msc} \mid \text{submsc}))
\end{align*}
\]
Charts : \( L(<\text{msc doc}) \rightarrow IP(L(<\text{msc}|<\text{submsc}))) \)

\[
\begin{align*}
\text{Charts}(\langle \rangle) &= \emptyset \\
\text{Charts}(\langle \text{msc} \rangle \langle \text{doc body} \rangle) &= \{ \langle \text{msc} \rangle \} \cup \text{Charts}(\langle \text{doc body} \rangle) \\
\text{Charts}(\langle \text{submsc} \rangle \langle \text{doc body} \rangle) &= \{ \langle \text{submsc} \rangle \} \cup \text{Charts}(\langle \text{doc body} \rangle) \\
\text{Charts}(\langle \text{doc} \rangle) &= \text{Charts}(\langle \text{DocBody}(\langle \text{doc} \rangle) \rangle)
\end{align*}
\]

IsDecomposed : \( L(<\text{inst def}) \rightarrow \mathbb{B} \)

\[
\begin{align*}
\text{IsDecomposed}(\text{instance } \langle \text{inst name} \rangle; \langle \text{inst body} \rangle \text{ endinstance;} ) &= \text{false} \\
\text{IsDecomposed}(\text{instance } \langle \text{inst name} \rangle \text{ decomposed}; \langle \text{inst body} \rangle \text{ endinstance;} ) &= \text{true}
\end{align*}
\]

Inst : \( L(<\text{msc body}) \rightarrow IP(L(<\text{inst def}))) \)

\[
\begin{align*}
\text{Inst}(\langle \rangle) &= \emptyset \\
\text{Inst}(\langle \text{inst def} \rangle \langle \text{msc body} \rangle) &= \{ \langle \text{inst def} \rangle \} \cup \text{Inst}(\langle \text{msc body} \rangle) \\
& \quad \text{if } \neg \text{IsDecomposed}(\langle \text{inst def} \rangle) \\
\text{Inst}(\langle \text{inst def} \rangle \langle \text{msc body} \rangle) &= \text{Inst}(\langle \text{msc body} \rangle) \\
& \quad \text{if } \text{IsDecomposed}(\langle \text{inst def} \rangle) \\
\text{Inst}(ch) &= \text{Inst}(\langle \text{ChBody}(ch) \rangle)
\end{align*}
\]

DecInst : \( L(<\text{msc body}) \rightarrow IP(L(<\text{inst def}))) \)

\[
\begin{align*}
\text{DecInst}(\langle \rangle) &= \emptyset \\
\text{DecInst}(\langle \text{inst def} \rangle \langle \text{msc body} \rangle) &= \{ \langle \text{inst def} \rangle \} \cup \text{DecInst}(\langle \text{msc body} \rangle) \\
& \quad \text{if } \neg \text{IsDecomposed}(\langle \text{inst def} \rangle) \\
\text{DecInst}(\langle \text{inst def} \rangle \langle \text{msc body} \rangle) &= \text{DecInst}(\langle \text{msc body} \rangle) \\
& \quad \text{if } \text{IsDecomposed}(\langle \text{inst def} \rangle) \\
\text{DecInst}(ch) &= \text{DecInst}(\langle \text{ChBody}(ch) \rangle)
\end{align*}
\]

AllDecInstNames : \( L(<\text{msc}) \rightarrow IP(L(<\text{inst name}))) \)

\[
\begin{align*}
\text{AllDecInstNames}(ch) &= \{ \text{InstName}(i) \mid i \in \text{DecInst}(ch) \}
\end{align*}
\]

Events : \( L(<\text{coevents}) \rightarrow IP(L(<\text{in}|<\text{out}))) \)

\[
\begin{align*}
\text{Events}(\langle \rangle) &= \emptyset \\
\text{Events}(\langle \text{in} \rangle \langle \text{coevents} \rangle) &= \{ \langle \text{in} \rangle \} \cup \text{Events}(\langle \text{coevents} \rangle) \\
\text{Events}(\langle \text{out} \rangle \langle \text{coevents} \rangle) &= \{ \langle \text{out} \rangle \} \cup \text{Events}(\langle \text{coevents} \rangle) \\
\text{CoEvents}(\langle \text{concurrent} \langle \text{coevents} \rangle \text{ endconcurrent;} \rangle) &= \text{Events}(\langle \text{coevents} \rangle)
\end{align*}
\]

CreateName : \( L(<\text{create}) \rightarrow L(<\text{inst name}>) \)

\[
\begin{align*}
\text{CreateName}(\text{create } \langle \text{inst name} \rangle \langle \langle \text{par list}\rangle \rangle)) &= \langle \text{inst name} \rangle
\end{align*}
\]

CreatedInsts : \( L(<\text{event}) \rightarrow IP(L(<\text{inst name}))) \)

\[
\begin{align*}
\text{CreatedInsts}(\langle \text{inst body} \rangle) &= \text{CreatedInsts}(\langle \text{inst body} \rangle) \\
\text{CreatedInsts}(\langle \text{inst def} \rangle) &= \text{CreatedInsts}(\langle \text{inst def} \rangle) \\
\text{CreatedInsts}(\langle \text{msc} \rangle \langle \text{submsc} \rangle) &= \text{CreatedInsts}(\langle \text{msc} \rangle \langle \text{submsc} \rangle)
\end{align*}
\]

42
\[
\text{CreatedInsts(<in>) = } \emptyset \\
\text{CreatedInsts(<out>) = } \emptyset \\
\text{CreatedInsts(<create>) = } \{\text{CreateName(<create>)}\} \\
\text{CreatedInsts(<set>) = } \emptyset \\
\text{CreatedInsts(<reset>) = } \emptyset \\
\text{CreatedInsts(<timeout>) = } \emptyset \\
\text{CreatedInsts(<coregion>) = } \emptyset \\
\text{CreatedInsts(<action>) = } \emptyset \\
\text{CreatedInsts(<condition>) = } \emptyset \\
\text{CreatedInsts(<event> [stop;]) = } \emptyset \\
\text{CreatedInsts(<event> <inst body>) = } \text{CreatedInsts(<event>)} \\
\qquad \quad \cup \text{CreatedInsts(<inst body>)} \\
\text{CreatedInsts(i) = } \text{CreatedInsts(InstBody(i))} \\
\text{CreatedInsts(ch) = } \bigcup_{i \in \text{Inst(ch)}} \text{CreatedInsts}(i)
\]
In this series appeared:

91/01 D. Alstein

91/02 R.P. Nederpelt, H.C.M. de Swart
Implication. A survey of the different logical analyses "if...then...", p. 26.

91/03 J.P. Katoen, L.A.M. Schoenmakers
Parallel Programs for the Recognition of P-invariant Segments, p. 16.

Performance Analysis of VLSI Programs, p. 31.

91/05 D. de Reus
An Implementation Model for GOOD, p. 18.

91/06 K.M. van Hee
SPECIFICATIEMETHODEN, een overzicht, p. 20.

91/07 E. Poll
CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.

91/08 H. Schepers
Terminology and Paradigms for Fault Tolerance, p. 25.

91/09 W.M.P.v.d.Aalst
Interval Timed Petri Nets and their analysis, p.53.

POLYNOMIAL RELATORS, p. 52.

91/11 R.C. Backhouse, P.J. de Bruin, G. Malcolm, E. Voermans, J. van der Woude
Relational Catamorphism, p. 31.

91/12 E. van der Sluis

91/13 F. Rietman
A note on Extensionality, p. 21.

91/14 P. Lemmens
The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15 A.T.M. Aerts, K.M. van Hee

91/16 A.J.J.M. Marcelis
An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17 A.T.M. Aerts
    P.M.E. de Bra
    K.M. van Hee

91/18 Rik van Geldrop

91/19 Erik Poll

91/20 A.E. Eiben
    R.Y. Schuwer

91/21 J. Coenen
    W.-P. de Roever
    J. Zwiers

91/22 G. Wolf

91/23 K.M. van Hee
    L.J. Somers
    M. Voorhoeve

91/24 A.T.M. Aerts
    D. de Reus

91/25 P. Zhou
    J. Hooman
    R. Kuiper

91/26 P. de Bra
    O.J. Houben
    J. Paredaens

91/27 F. de Boer
    C. Palamidessi

91/28 F. de Boer

91/29 H. Ten Eikelder
    R. van Geldrop

91/30 J.C.M. Baeten
    F.W. Vaandrager

91/31 H. ten Eikelder

91/32 P. Struik

91/33 W. v.d. Aalst

91/34 J. Coenen

Transforming Functional Database Schemes to Relational Representations, p. 21.

Transformational Query Solving, p. 35.

Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.


Assertional Data Reification Proofs: Survey and Perspective, p. 18.


Z and high level Petri nets, p. 16.

Formal semantics for BRM with examples, p. 25.

A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.

The GOOD based hypertext reference model, p. 12.

Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.

A compositional proof system for dynamic process creation, p. 24.

Correctness of Acceptor Schemes for Regular Languages, p. 31.

An Algebra for Process Creation, p. 29.

Some algorithms to decide the equivalence of recursive types, p. 26.


The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.

Specifying fault tolerant programs in deontic logic, p. 15.
<table>
<thead>
<tr>
<th>Year/No</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p.45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p.22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p.36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Seljée</td>
<td>Integrity checking in deductive databases: an exposition, p.32.</td>
</tr>
<tr>
<td>92/17</td>
<td>W.M.P. van der Aalst</td>
<td>Interval timed coloured Petri nets and their analysis, p. 20.</td>
</tr>
<tr>
<td>92/18</td>
<td>R.Nederpelt, F. Kamareddine</td>
<td>A unified approach to Type Theory through a refined lambda-calculus, p. 30.</td>
</tr>
<tr>
<td>92/20</td>
<td>F.Kamareddine</td>
<td>Are Types for Natural Language? P. 32.</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>92/21</td>
<td>F.Kamareddine</td>
<td>Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.</td>
</tr>
<tr>
<td>92/22</td>
<td>R. Nederpelt, F.Kamareddine</td>
<td>A useful lambda notation, p. 17.</td>
</tr>
<tr>
<td>92/23</td>
<td>F.Kamareddine, E.Klein</td>
<td>Nominalization, Predication and Type Containment, p. 40.</td>
</tr>
<tr>
<td>92/24</td>
<td>M.Codish, D.Dams, Eyal Yardeni</td>
<td>Bottom-up Abstract Interpretation of Logic Programs, p. 33.</td>
</tr>
<tr>
<td>92/25</td>
<td>E.Poll</td>
<td>A Programming Logic for Föa, p. 15.</td>
</tr>
<tr>
<td>93/01</td>
<td>R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
</tr>
<tr>
<td>93/02</td>
<td>T. Verhoeff</td>
<td>A continuous version of the Prisoner's Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03</td>
<td>T. Verhoeff</td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/04</td>
<td>E.H.L. Aarts, J.H.M. Korst, P.J. Zwietering</td>
<td>Deterministic and randomized local search, p. 78.</td>
</tr>
<tr>
<td>93/05</td>
<td>J.C.M. Baeten, C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates, p. 18.</td>
</tr>
<tr>
<td>93/06</td>
<td>J.P. Vellkamp</td>
<td>On the unavoidability of metastable behaviour, p. 29</td>
</tr>
<tr>
<td>93/07</td>
<td>P.D. Moerland</td>
<td>Exercises in Multiprogramming, p. 97</td>
</tr>
<tr>
<td>93/08</td>
<td>J. Verhoosel</td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
</tr>
<tr>
<td>93/10</td>
<td>K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.</td>
</tr>
<tr>
<td>93/13</td>
<td>K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach</td>
</tr>
<tr>
<td>Paper Reference</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>93/16</td>
<td>H. Schepers, J. Hooman</td>
<td>A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27</td>
</tr>
<tr>
<td>93/17</td>
<td>D. Alstein, P. van der Stok</td>
<td>Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.</td>
</tr>
<tr>
<td>93/18</td>
<td>C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.</td>
</tr>
<tr>
<td>93/19</td>
<td>G-J. Houben</td>
<td>The Design of an Online Help Facility for ExSpect, p. 21.</td>
</tr>
<tr>
<td>93/22</td>
<td>E. Poll</td>
<td>A Typechecker for Bijective Pure Type Systems, p. 28.</td>
</tr>
<tr>
<td>93/23</td>
<td>E. de Kogel</td>
<td>Relational Algebra and Equational Proofs, p. 23.</td>
</tr>
<tr>
<td>93/24</td>
<td>E. Poll and Paula Severi</td>
<td>Pure Type Systems with Definitions, p. 38.</td>
</tr>
<tr>
<td>93/26</td>
<td>W.M.P. van der Aalst</td>
<td>Multi-dimensional Petri nets, p. 25.</td>
</tr>
<tr>
<td>93/27</td>
<td>T. Kloks and D. Kratsch</td>
<td>Finding all minimal separators of a graph, p. 11.</td>
</tr>
<tr>
<td>93/28</td>
<td>F. Kamareddine and R. Nederpelt</td>
<td>A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.</td>
</tr>
<tr>
<td>93/29</td>
<td>R. Post and P. De Bra</td>
<td>GOLD, a Graph Oriented Language for Databases, p. 42.</td>
</tr>
<tr>
<td>93/30</td>
<td>J. Deogun, T. Kloks, D. Kratsch, H. Müller</td>
<td>On Vertex Ranking for Permutation and Other Graphs, p. 11.</td>
</tr>
<tr>
<td>93/31</td>
<td>W. Körver</td>
<td>Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.</td>
</tr>
<tr>
<td>Page</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</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. Rambags</td>
<td>Automatic Verification of Regular Protocols in P/T Nets, p. 23.</td>
</tr>
<tr>
<td>93/43</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata construction algorithms, p. 87.</td>
</tr>
<tr>
<td>93/44</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata minimization algorithms, p. 23.</td>
</tr>
<tr>
<td>93/48</td>
<td>R. Gerth</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
<tr>
<td>Paper Number</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>--------------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>94/01</td>
<td>P. America, M. van der Kammen, R.P. Nederpelt, O.S. van Roosmalen, H.C.M. de Swart</td>
<td>The object-oriented paradigm, p. 28.</td>
</tr>
<tr>
<td>94/02</td>
<td>F. Kamareddinc, R.P. Nederpelt</td>
<td>Canonical typing and Π-conversion, p. 51.</td>
</tr>
<tr>
<td>94/04</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Graph Isomorphism Models for Non-Interleaving Process Algebra, p. 18.</td>
</tr>
<tr>
<td>94/06</td>
<td>T. Basten, T. Kunz, J. Black, M. Coffin, 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, J.A. Bergstra</td>
<td>Process Algebra with Partial Choice, p. 16.</td>
</tr>
<tr>
<td>94/10</td>
<td>T. verhoef</td>
<td>The testing Paradigm Applied to Network Structure. p. 31.</td>
</tr>
<tr>
<td>94/13</td>
<td>R. Seljée</td>
<td>A New Method for Integrity Constraint checking in Deductive Databases, p. 34.</td>
</tr>
<tr>
<td>94/14</td>
<td>W. Peremans</td>
<td>Ups and Downs of Type Theory, p. 9.</td>
</tr>
<tr>
<td>94/16</td>
<td>R.C. Backhouse, H. Doorens</td>
<td>Mathematical Induction Made Calculational, p. 36.</td>
</tr>
<tr>
<td>94/18</td>
<td>F. Kamareddine</td>
<td>Refining Reduction in the Lambda Calculus, p. 15.</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>R. Nederpelt</td>
<td></td>
</tr>
<tr>
<td>94/19</td>
<td>B.W. Watson</td>
<td>The performance of single-keyword and multiple-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>keyword pattern matching algorithms, p. 46.</td>
</tr>
<tr>
<td>94/20</td>
<td>R. Bloo</td>
<td>Beyond $\beta$-Reduction in Church's $\lambda \to$, p. 22.</td>
</tr>
<tr>
<td></td>
<td>F. Kamareddine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R. Nederpelt</td>
<td></td>
</tr>
<tr>
<td>94/22</td>
<td>B.W. Watson</td>
<td>The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.</td>
</tr>
</tbody>
</table>