Contents

Summary

Scope / Objective

Message Sequence Chart is a graphical and textual language for the description and specification of the interactions between system components. The purpose of the formal definition of the semantics is to provide for an unambiguous interpretation of Message Sequence Charts.

Coverage

The document presents a formal semantics of Message Sequence Charts using techniques from process algebra. The semantic constructions are introduced incrementally. This means that first the semantics of Basic Message Sequence Charts is given, and that subsequently additional features are added until the complete language is covered. Examples are added which explain the use of the semantic functions.

Application

The formalization of the semantics of Message Sequence Charts serves several purposes. For users it will help in order to obtain a clear understanding of Message Sequence Charts and in order to further a harmonization of the use. Tool builders can use the semantics for derivation of prototypes directly from the definitions provided or they can base their computer applications on these definitions. Validation and comparison of tools may be based on the formal semantics. Finally the developers of the Message Sequence Chart language can benefit because the semantics may show overlap or lack of features and may guide in unification of features.

Status / Stability

The semantics described here is a formalization of the semantics informally explained in the main text of the Recommendation. This interpretation of Message Sequence Charts is fairly stable and it is expected that alternative interpretations will only be implemented by the addition of extra features to the language. This annex describes the semantics of a single Message Sequence Chart and possible refinements only. Since there is no means within the Message Sequence Chart language to explicitly define the relation between the Message Sequence Charts in a Message Sequence Chart document, there is no semantics for a complete document. The development of structural concepts for Message Sequence Charts is still subject of research.

Associated work

Recommendation Z.120: Message Sequence Charts (MSC)

Annex B to Recommendation Z.120

Algebraic Semantics of Message Sequence Charts

(This annex forms an integral part of the Recommendation.)

B.1 Introduction

Message Sequence Chart is a graphical and textual language for the description and specification of the interactions between system components. 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.

This document contains a formal semantics of the Message Sequence Chart language based on the informal explanation of the semantics in the main text of the Recommendation. The primary reason for formalizing the semantics is to provide for an unambiguous interpretation of Message Sequence Charts. A formal semantics may be useful for users, tool builders and developers of the Message Sequence Chart language.

The semantics is defined in a stepwise way. First the semantics of the core language (Basic Message Sequence Charts) is treated, and subsequently other features are added such as process creation and termination, timer handling and refinement.

This document only defines a semantics for a single Message Sequence Chart together with its corresponding Sub Message Sequence Charts. A semantics cannot be defined for a complete Message Sequence Chart document, because the relation between the Message Sequence Charts in a document is not explicitly stated.

The formal semantics presented in this document is based on the algebraic theory of process description ACP (Algebra of Communicating Processes) [3, 5]. ACP is a theory in many ways related to the algebraic process theories CCS (Calculus of Communicating Systems) [8] and CSP (Communicating Sequential Processes) [6]. The process algebra ACP is a useful framework for the description of the formal semantics of Message Sequence Charts since all features incorporated in Message Sequence Charts are related to topics already studied in process algebra, such as the state operator, the process creation operator and the global renaming operator. Since Message Sequence Charts may be 'empty', ACP is extended with the empty process ε [2, 3]. Furthermore, the synchronous communication function can be deleted from ACP because Message Sequence Charts have asynchronous communication only. The resulting theory is called PA_ε.

This document is structured in the following way. Section B.2 contains an overview of the Message Sequence Chart language and informally lists all static requirements which are relevant for the definition of the semantics. First the core language (Basic Message Sequence Charts) is introduced. The other features are added one by one.

In Section B.3 the algebraic theory PA_ε is defined which is the formal framework for the definition of the semantics.

The semantic functions are defined in Section B.4. The main function maps every Message Sequence Chart into an expression in the theory PA_ε. First the semantics of a Basic Message Sequence Chart is defined. Subsequently this construction is extended with the semantics of all

other features. For easy reference an overview of the complete semantics is given in Section B.4.10.

B.2 Message Sequence Charts

B.2.1 Introduction

This section contains an introduction to 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 the core language of Message Sequence Charts is introduced. This core language 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 B.4, are given. The static requirements are not formalized. After the introduction of Basic Message Sequence Charts the other primitives incorporated in the language of Message Sequence Charts are introduced. These primitives are process creation and process termination, timer handling, coregions, conditions, and refinement.

The syntax of Message Sequence Charts as defined in Section B.5 is used for the definition of the semantics. This syntax is different from the syntax in the main text of the Recommendation, but it is better suited for the definition of the semantics. The languages generated by both syntaxes are equal with respect to those constructs which have a semantic meaning.

B.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 on an instance 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 the *action text* placed in it. 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. The parameter list is denoted between brackets after the message name.

Example B.2.2.1 Consider the messages $m1$, $m2$, $m3$ and $m4$ in Figure B.1. Message $m0$ is sent to the environment. The behavior of the environment is not specified. For instance $i2$ also a local action a is defined.

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 B.1 this implies that message $m3$ is received by $i4$

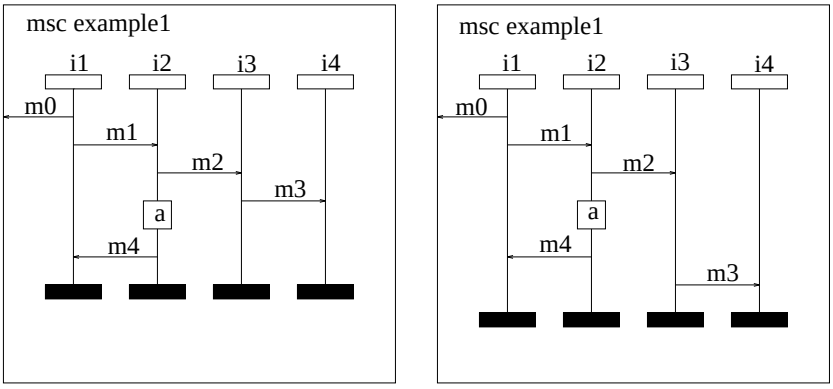

Figure B.1: Example Basic Message Sequence Charts

only after it has been sent by $i3$, and, consequently, after the consumption of $m2$ by $i3$. Thus the events concerning $m1$ and $m3$ are ordered in time, while for the events of $m4$ and $m3$ no order is specified apart from the requirement that the output of a message occurs before its input. 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 B.1 defines the same Basic Message Sequence Chart (from a semantic point of view), but in an alternative drawing.

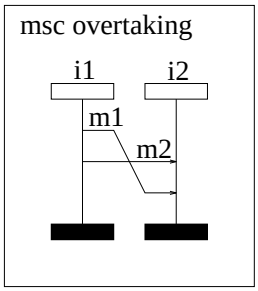

Figure B.2: Basic Message Sequence Chart with overtaking

Because of the asynchronous communication, it would even be possible to first send $m3$, then send and receive $m4$, and finally receive $m3$. Another consequence of this mode of communication is that overtaking of messages is allowed, as expressed in Figure B.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 document it is used 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 m1 to i2;`" and a message input by "`in m1 from i1;`".

The Basic Message Sequence Charts of Figure B.1 have the following textual representation.

```
msc example1;
   instance i1;
      out m0 to env;
      out m1 to i2;
```

```
      in m4 from i2;
   endinstance;
   instance i2;
      in m1 from i1;
      out m2 to i3;
      action a;
      out m4 to i1;
   endinstance;
   instance i3;
      in m2 from i2;
      out m3 to i4;
   endinstance;
   instance i4;
      in m3 from i3;
   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 B.1. The nonterminals `<at>`, `<inst name>`, `<min>`, `<mn>`, `<msc name>`, 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`. The language generated

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

```
<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 <msgid> to <address>;
<msgid>       ::=  <mid> [(<par list>)]
<mid>         ::=  <mn> [,<min>]
<par list>    ::=  <par name> [,<par list>]
<address>     ::=  <inst name> | env
<in>          ::=  in <msgid> from <address>;
<action>      ::=  action <at>;
```

by a nonterminal `X` in the grammar of Table B.1 will be denoted by $\mathcal{L}(\texttt{X})$.

The static requirements for Basic Message Sequence Charts are formulated informally 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.

Example B.2.2.2 Consider the first diagram in Figure B.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 diagram therefore violates the static requirements. In this example the message output is depending on its corresponding message input in a direct way.

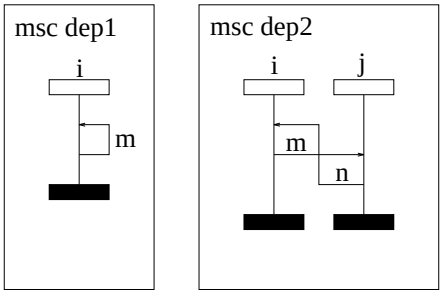

Figure B.3: Two diagrams that violate the static requirements

As an example of the indirect causal dependency between a message output and a message input the second diagram in Figure B.3 is considered. Amongst others, there are 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 diagram specifies that the input of message m precedes the output of message m. So the diagram violates the static requirements, and is therefore not a Basic Message Sequence Chart.

B.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. 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.

Example B.2.3.1 In Figure B.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.

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;`". The grammar for Basic Message Sequence Charts

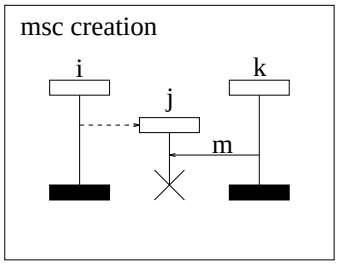

Figure B.4: Message Sequence Chart with process creation and termination

Table B.2: Extension of the grammar with process creation and termination

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

in Table B.1 is extended with the rules in Table B.2. The identifiers `create` and `stop` are reserved keywords.

The textual representation of the Message Sequence Chart in Figure B.4 is as follows.

```
msc creation;
   instance i;
      create j;
   endinstance;
   instance j;
      in m from k;
      stop;
   endinstance;
   instance k;
      out m to j;
   endinstance;
endmsc;
```

With respect to Basic Message Sequence Charts extended with process creation and termination the following static requirements are added. 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.

B.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 instance 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 *timer name*, that is placed aside the small rectangle. The setting of a timer may be labelled with an identifier for the duration, the *duration name*. The duration name is placed

between brackets after 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.

Example B.2.4.1 In Figure B.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.

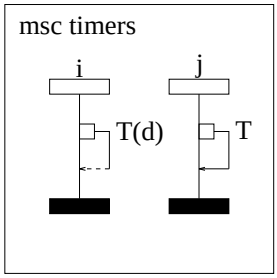

Figure B.5: Message Sequence Chart with timer handling

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 "`set T;`" and the corresponding reset by "`reset T;`" and timeout by "`timeout T;`". The grammar in Table B.2 is extended with the rules in Table B.3. The nonterminals `<dn>`, `<tin>` and `<tn>` represent identifiers. The identifiers `reset`, `set` and `timeout` are reserved keywords.

Table B.3: Extension with timer-handling

```
<event>     ::=  <set> | <reset> | <timeout>
<set>       ::=  set <tid> [(<dn>)];
<tid>       ::=  <tn> [,<tin>]
<reset>     ::=  reset <tid>;
<timeout>   ::=  timeout <tid>;
```

The Message Sequence Chart in Figure B.5 is represented as follows.

```
msc timer;
   instance i;
      set T(d);
      reset T;
   endinstance;
   instance j;
      set T;
      timeout T;
   endinstance;
endmsc;
```

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.

B.2.5 Coregions

So far the events specified on an instance were 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 instance 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.

Example B.2.5.1 In Figure B.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.

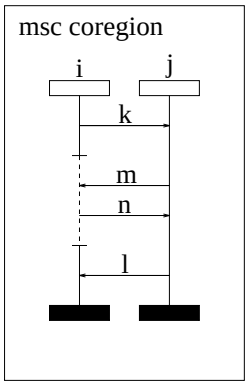

Figure B.6: Message Sequence Chart with a coregion

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 B.4 the rules for the extension with coregions are given.

Table B.4: Extension with coregions

```
<event>      ::=  <coregion>
<coregion>   ::=  concurrent <coevents> endconcurrent;
<coevents>   ::=  <> | <out> <coevents> | <in> <coevents>
```

The textual representation of the Message Sequence Chart in Figure B.6 is as follows.

```
msc coregion;
   instance i;
```

```
      out k to j;
      concurrent
         in m from j;
         out n to j;
      endconcurrent;
      in l from j;
   endinstance;
   instance j;
      in k from i;
      out m to i;
      in n from i;
      out l to i;
   endinstance;
endmsc;
```

B.2.6 Conditions

A condition describes a state referring to a (non-empty) subset of instances specified in the Message Sequence Chart. 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.

Example B.2.6.1 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 B.7 a Message Sequence Chart with three conditions is given. Conditions $C1$ refers to all instances, $C2$ refers to i and $C3$ refers to i and k.

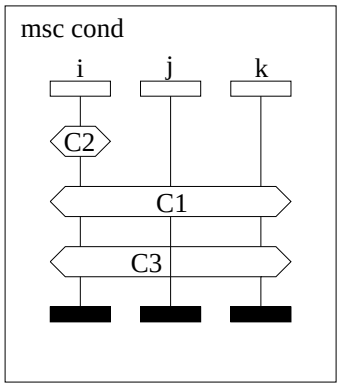

Figure B.7: Message Sequence Chart with conditions

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. If the condition refers to all instances the instance list may be replaced by the reserved keyword `all`.

In Table B.5 the rules for the extension with conditions are given. The nonterminal `<cn>` represents an identifier. The identifiers `all`, `condition` and `shared` are reserved keywords.

Table B.5: Extension with conditions

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

The Message Sequence Chart in Figure B.7 is in the textual representation given by

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

B.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.

Example B.2.7.1 In Figure B.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.

In the textual representation these charts are represented by

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

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

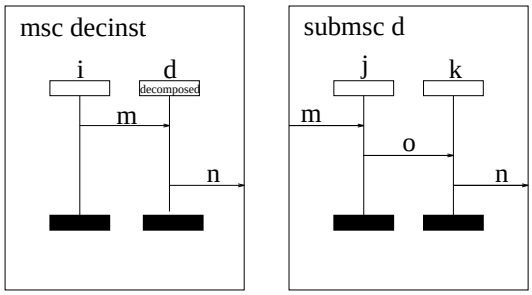

Figure B.8: Refinement

```
   endinstance;                          out n to env;
endmsc;                                endinstance;
                                    endsubmsc;
```

In the textual representation, an instance that is decomposed is labelled with the reserved keyword **decomposed**. Because of the refinement primitive a complete description consists of a collection of charts. Such a collection of charts is called a *Message Sequence Chart document.* A Message Sequence Chart document may contain more than one Message Sequence Chart and their corresponding refinements. In Table B.6 the rules for the extension with instance refinement are given. The nonterminal **<doc name>** represents an identifier. The identifiers **decomposed**, **endmscdocument**, **endsubmsc**, **mscdocument** and **submsc** are reserved keywords.

Table B.6: Extension with instance refinement

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

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.

B.3 The process algebra PA_ε

B.3.1 Introduction

The process algebra PA_ε is an algebraic theory for the description of process behavior [2, 3]. 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_ε is denoted by Σ_{PA_ε} and the set of equations is denoted by E_{PA_ε}.

B.3.2 Signature and equations

A signature is a set of constant and function symbols. For every function symbol in the signature its arity, i.e. the number of arguments, is specified. With the symbols from a signature and with variables from some set V, terms can be constructed.

Definition B.3.2.1 *Let Σ be a signature and let V be a set of variables. Terms over signature Σ 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 Σ 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 Σ is denoted by $T(\Sigma)$.

An algebraic theory (Σ, E) consists of a signature Σ and a set of equations E. An equation is an expression of the form $s = t$ with s and t terms. Next, the derivability of equations with respect to a signature Σ and a set of equations E is defined. The derivability relation expresses which equations between the process terms associated with the signature Σ can be derived from the equations of E. The notions of a context and substitution of variables are used in the definition of derivability.

Definition B.3.2.2 *A substitution of variables is a function $\sigma : V \rightarrow T(\Sigma, V)$ which assigns a term to each variable. The extension $\overline{\sigma} : T(\Sigma, V) \rightarrow T(\Sigma, V)$ is defined as follows*

1) $\overline{\sigma}(v) = \sigma(v)$ if v is a variable

2) $\overline{\sigma}(c) = c$ if c is a constant symbol

3) $\overline{\sigma}(f(t_1, \cdots, t_n)) = f(\overline{\sigma}(t_1), \cdots, \overline{\sigma}(t_n))$ if f is an n-ary function symbol and $t_1, \cdots, t_n$ are terms

A context can be regarded as a term with a 'hole' in it. Contexts are defined formally in the following definition.

Definition B.3.2.3 *Contexts are defined inductively by*

1) [] is a context

2) if f is an n-ary function symbol, $t_1, \cdots, t_{n-1}$ are terms and C is a context, then $f(t_1, \cdots, t_{i-1}, C, t_i, \cdots, t_{n-1})$ is a context ($1 \leq i \leq n$)

If C is a context and t is a term, then $C[t]$ denotes the insertion of term t in the hole of the context.

Definition B.3.2.4 *If C is a context and t is a term, then the term $C[t]$ is defined inductively by*

1) $[][t] = t$

2) $f(t_1, \cdots, t_{i-1}, C, t_i \cdots, t_{n-1})[t] = f(t_1, \cdots, t_{i-1}, C[t], t_i, \cdots, t_{n-1})$ *for f an n-ary function symbol and $t_1, \cdots, t_{n-1}$ terms*

Definition B.3.2.5 *Let Σ be a signature and let E be a set of equations over the signature Σ and a set of variables V. Then the derivability of an equation $t_1 = t_2$ with respect to the algebraic theory (Σ, E), notation $(\Sigma, E) \vdash t_1 = t_2$, is for all $s, t, u \in T(\Sigma, V)$ defined by*

1) if $s = t \in E$, then $(\Sigma, E) \vdash s = t$

2) for all $s \in T(\Sigma, V) : (\Sigma, E) \vdash s = s$

3) if $(\Sigma, E) \vdash s = t$, then $(\Sigma, E) \vdash t = s$

4) if $(\Sigma, E) \vdash s = t$ and $(\Sigma, E) \vdash t = u$, then $(\Sigma, E) \vdash s = u$

5) if $(\Sigma, E) \vdash s = t$, then for any substitution $\sigma : V \to T(\Sigma, V) : (\Sigma, E) \vdash \overline{\sigma}(s) = \overline{\sigma}(t)$

6) if $(\Sigma, E) \vdash s = t$ for some $s, t \in T(\Sigma, V)$, then for all contexts $C : (\Sigma, E) \vdash C[s] = C[t]$

If $(\Sigma, E) \vdash s = t$ then s and t are called *derivably equal.*

Next a model for (Σ, E) is defined which satisfies exactly those equations between closed terms that are derivable from the equational specification, and no others.

Definition B.3.2.6 *The initial algebra $I(\Sigma, E)$ of an equational specification (Σ, E) is the model that results from identifying all closed terms from the set $T(\Sigma)$ which are derivably equal.*

The initial algebra $I(\Sigma, E)$ is considered the model for the equational specification (Σ, E). Especially, $I(\Sigma_{PA_\varepsilon}, E_{PA_\varepsilon})$ is the semantics of the process algebra PA_ε which will be discussed in the remainder of this section.

B.3.3 The signature of PA_ε

The signature Σ_{PA_ε} of PA_ε specifies the building blocks for the description of processes.

Definition B.3.3.1 *The signature Σ_{PA_ε} consists of*

1) the special constants δ and ε

2) the set of unspecified constants A

3) the unary operator $\surd$

4) the binary operators $+$, $\cdot$, $\|$ and $\between$

The special constant δ denotes the process that has stopped executing actions and cannot proceed. This constant is called *deadlock*. The special constant ε 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. This set will be specified as soon as an application of the theory is considered.

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 $\|$ 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 the definition of the merge two auxiliary operators are used. The *termination operator* $\surd$ applied to a process x signals whether or not the process x has an option to terminate immediately. The binary operator $\lfloor\!\lfloor$ 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.

B.3.4 The equations of PA_ε

The set of equations E_{PA_ε} of PA_ε specifies which processes are considered equal.

Definition B.3.4.1 *For $a \in A$ and $x, y, z \in V$, the equations of PA_ε are given in Table B.7.*

The axioms are explained below. Axioms A1, A2 and A3 state that the alternative composition is commutative, associative and idempotent. Axiom A4 defines the right distributivity of the sequential composition over alternative composition. Associativity of the sequential composition is given in axiom A5. Axiom A6 states that deadlock is the unit of alternative composition, so if there is an alternative, a process will not deadlock. The fact that after a deadlock has occurred nothing else can happen is reflected in axiom A7. The empty process is the unit of sequential composition (A8 – A9). The axioms TE1–TE4 express that a process x has an option to terminate immediately if $\surd(x) = \varepsilon$, and that $\surd(x) = \delta$ otherwise. In itself the termination operator is not very interesting, but in defining the free merge this operator is needed 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 either x or 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.

The following properties are taken from [3].

Lemma B.3.4.2 For closed PA_ε terms x, y and z and $a \in A$

1) $x \| y = y \| x$

2) $x \| \varepsilon = \varepsilon \| x = x$

3) $(x \| y) \| z = x \| (y \| z)$

Table B.7: Axioms of PA_ε

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5
$x + \delta = x$	A6
$\delta \cdot x = \delta$	A7
$x \cdot \varepsilon = x$	A8
$\varepsilon \cdot x = x$	A9
$x \parallel y = x \between y + y \between x + \surd(x) \cdot \surd(y)$	TM1
$\varepsilon \between x = \delta$	TM2
$\delta \between x = \delta$	TM3
$a \cdot x \between y = a \cdot (x \parallel y)$	TM4
$(x + y) \between z = x \between z + y \between z$	TM5
$\surd(\varepsilon) = \varepsilon$	TE1
$\surd(\delta) = \delta$	TE2
$\surd(a \cdot x) = \delta$	TE3
$\surd(x + y) = \surd(x) + \surd(y)$	TE4

4) $a \between x = a \cdot x$

Because of the symmetry and associativity of the free merge, a quantified notation for this operator is allowed.

Definition B.3.4.3 *Let D be a finite index set and let $P(d)$ be an expression over d. The quantified free merge operator, $\parallel_D$, is defined inductively by*

$$\parallel_{d \in \varnothing} P(d) = \varepsilon$$
$$\parallel_{d \in D \cup \{d_1\}} P(d) = P(d_1) \parallel \left(\parallel_{d \in D \setminus \{d_1\}} P(d) \right)$$

Example B.3.4.4 The free merge is illustrated by the following two examples.

1) $b \parallel c = b \between c + c \between b + \surd(x) \cdot \surd(y)$
$= b \cdot c + c \cdot b + \delta$
$= b \cdot c + c \cdot b$

2) $a \cdot b \parallel c = a \cdot b \between c + c \between a \cdot b + \surd(a \cdot b) \cdot \surd(c)$
$= a \cdot (b \parallel c) + c \cdot a \cdot b + \delta$
$= a \cdot (b \cdot c + c \cdot b) + c \cdot a \cdot b$

B.4 Semantics of Message Sequence Charts

B.4.1 Introduction

In this section the semantics of Message Sequence Charts is presented in a stepwise way. First, the general form of the semantic functions is discussed. After that, the atomic actions that represent

the smallest events specified within Message Sequence Charts are defined. Then, the semantics of Basic Message Sequence Charts is considered [7]. This semantics is extended to the semantics of the complete language by adding the other primitives one by one. Section B.4.10 contains an overview of the complete semantics.

In defining the semantics of Message Sequence Charts some additional process algebra operators are needed. The extended process algebra is called PA_{MSC}. Some auxiliary functions and predicates are used. Since the names of those functions and predicates are chosen to be representative for the intended meaning, those functions and predicates are only defined in the overview of the semantics (Section B.4.10.4).

B.4.2 The semantic function

The semantic function translates every Message Sequence Chart into a process algebra expression. Its general appearance is

$$S_{\texttt{<X>}}(p)[\![a]\!]$$

with `<X>` a nonterminal from the grammar, p some parameter and a the actual argument representing (part of) a Message Sequence Chart.

For example, $S_{\texttt{<event>}}(iid)[\![\texttt{action}\ at\texttt{;}]\!]$ (See Definition B.4.4.6) determines the semantics of the event `action` at`;` executed by instance iid.

B.4.3 Specifying the atomic actions

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

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

If no parameter list is specified for a message event, a create event or a start event, then it is taken to be <>.

Definition B.4.3.1 *In Table B.8 the sets of atomic actions A_a, A_o, A_{oe}, A_i, A_{ie}, A_c, A_{st}, A_s, A_t and their union A are defined. For a nonterminal* X, $\mathcal{L}_{<>}$(X) *is used to denote* $\mathcal{L}$(X) $\cup \{<>\}$.

Table B.8: The atomic actions of PA_{MSC}

A_a	=	$\{action(i,at) \mid i \in \mathcal{L}(\texttt{<inst name>}), at \in \mathcal{L}(\texttt{<at>})\}$
A_o	=	$\{out(i,j,m,p) \mid i,j \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{<>}(\texttt{<par list>})\}$
A_{oe}	=	$\{out(i,env,m,p) \mid i \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{<>}(\texttt{<par list>})\}$
A_i	=	$\{in(i,j,m,p) \mid i,j \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{<>}(\texttt{<par list>})\}$
A_{ie}	=	$\{in(env,i,m,p) \mid i \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{<>}(\texttt{<par list>})\}$
A_c	=	$\{create(i,j,p) \mid i,j \in \mathcal{L}(\texttt{<inst name>}), p \in \mathcal{L}_{<>}(\texttt{<par list>})\}$
A_{st}	=	$\{start(i,p) \mid i \in \mathcal{L}(\texttt{<inst name>}), p \in \mathcal{L}_{<>}(\texttt{<par list>})\}$
A_s	=	$\{stop(i) \mid i \in \mathcal{L}(\texttt{<inst name>})\}$
A_t	=	$\{set(i,t,d) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>}), d \in \mathcal{L}(\texttt{<dn>})\}$
		$\cup \{set(i,t) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>})\}$
		$\cup \{reset(i,t) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>})\}$
		$\cup \{timeout(i,t) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>})\}$
A	=	$A_a \cup A_o \cup A_{oe} \cup A_i \cup A_{ie} \cup A_c \cup A_{st} \cup A_s \cup A_t$

In order to discriminate between actions which are defined on different instances, a function D is defined which associates to an atomic action the name of the instance it is defined on.

Definition B.4.3.2 *The function* $D : A \rightarrow \mathcal{L}$(<inst name>) *is for all* $i,j \in \mathcal{L}$(<inst name>), $at \in \mathcal{L}$(<at>), $t \in \mathcal{L}$(<tid>), $d \in \mathcal{L}$(<dn>), $m \in \mathcal{L}$(<mid>) *and* $p \in \mathcal{L}_{<>}$(<par list>) *defined in Table B.9.*

Table B.9: The function D

$D(action(i,at))$	=	i	$D(start(i,p))$	=	i
$D(out(i,j,m,p))$	=	i	$D(stop(i))$	=	i
$D(out(i,env,m,p))$	=	i	$D(set(i,t,d))$	=	i
$D(in(i,j,m,p))$	=	j	$D(set(i,t))$	=	i
$D(in(env,j,m,p))$	=	j	$D(reset(i,t))$	=	i
$D(create(i,j,p))$	=	i	$D(timeout(i,t))$	=	i

B.4.4 Basic Message Sequence Charts

A Basic Message Sequence Chart specifies a finite number of non-decomposed instances that communicate by exchanging 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 also interleavings in which a message output is preceded by its corresponding message input are enabled. An operator λ_M is introduced that enables only those interleavings that respect the constraint above. This operator is called the *I/O-filter*. The I/O-filter λ_M remembers the message identifier of all message outputs that have been executed and for which no corresponding message input was executed yet, in a set M. It 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 I/O-filter λ_M is an instance of the state operator [1, 3]. The I/O-filter considers messages between instances only. For communication events exchanged with the environment there is no corresponding communication event.

Definition B.4.4.1 *Let* $M \subseteq \mathcal{L}$(<mid>). *For all* $x, y \in V$, $a \in A$, $i, j \in \mathcal{L}$(<inst name>), $m \in \mathcal{L}$(<mid>) *and* $p \in \mathcal{L}_{<>}$(<par list>), *the I/O-filter* λ_M *is defined in Table B.10.*

Table B.10: The I/O-filter λ_M

$\lambda_M(\varepsilon) = \varepsilon$	if $M = \varnothing$	LM1
$\lambda_M(\varepsilon) = \delta$	if $M \neq \varnothing$	LM2
$\lambda_M(\delta) = \delta$		LM3
$\lambda_M(a \cdot x) = a \cdot \lambda_M(x)$	if $a \notin A_o \cup A_i$	LM4
$\lambda_M(out(i,j,m,p) \cdot x) = \delta$	if $m \in M$	LM5
$\lambda_M(out(i,j,m,p) \cdot x) = out(i,j,m,p) \cdot \lambda_{M \cup \{m\}}(x)$	if $m \notin M$	LM6
$\lambda_M(in(i,j,m,p) \cdot x) = in(i,j,m,p) \cdot \lambda_{M \setminus \{m\}}(x)$	if $m \in M$	LM7
$\lambda_M(in(i,j,m,p) \cdot x) = \delta$	if $m \notin M$	LM8
$\lambda_M(x + y) = \lambda_M(x) + \lambda_M(y)$		LM9

Some of the axioms defining λ_M are explained below. If λ_M encounters a non-message event or a message referring to the environment, 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, a message output with message identifier m is already executed. In that case it is not allowed to execute any events, so a deadlock results (LM5). If the Message Sequence Chart satisfies the static requirements this situation cannot occur. In the other case the message output may just be executed and the set M is extended with the message identifier m (LM6). If a message input with message identifier m is encountered and a message output with message identifier m was not executed in the past then it is not allowed to execute any more events (LM8). If a message output with message identifier m was already executed then the input message can be executed and the message identifier m is removed from M (LM7).

Example B.4.4.2 The following examples illustrate the use of the I/O-filter λ_M:

$$
\begin{array}{lll}
1) \quad \lambda_{\varnothing}(out(i,j,m,p)\cdot in(i,j,m,p)) & = & out(i,j,m,p)\cdot\lambda_{\{m\}}(in(i,j,m,p)) \\
 & = & out(i,j,m,p)\cdot in(i,j,m,p)\cdot\lambda_{\varnothing}(\varepsilon) \\
 & = & out(i,j,m,p)\cdot in(i,j,m,p) \\
2) \quad \lambda_{\varnothing}(in(i,j,m,p)\cdot out(i,j,m,p)) & = & \delta \\
3) \quad \lambda_{\varnothing}(out(i,j,m,p)\,\|\,in(i,j,m,p)) & = & \lambda_{\varnothing}(out(i,j,m,p)\cdot in(i,j,m,p) \\
 & & +in(i,j,m,p)\cdot out(i,j,m,p) \\
 & & +\surd(out(i,j,m,p))\cdot\surd(in(i,j,m,p))) \\
 & = & \lambda_{\varnothing}(out(i,j,m,p)\cdot in(i,j,m,p)) \\
 & & +\lambda_{\varnothing}(in(i,j,m,p)\cdot out(i,j,m,p))+\delta \\
 & = & out(i,j,m,p)\cdot in(i,j,m,p)+\delta+\delta \\
 & = & out(i,j,m,p)\cdot in(i,j,m,p)
\end{array}
$$

The semantics of a Basic Message Sequence Chart is obtained by placing all instances in parallel and applying the I/O-filter with the set M initialized by the empty set ($\lambda_{\varnothing}$).

Definition B.4.4.3 *The semantic function for Basic Message Sequence Charts,* $S_{\texttt{<msc>}}:\mathcal{L}(\texttt{<msc>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $ch \in \mathcal{L}(\texttt{<msc>})$ *defined by*

$$S_{\texttt{<msc>}}[\![ch]\!] \;=\; \lambda_{\varnothing}\left(\,\|_{\,i\in AllInst(ch)}\; S_{\texttt{<inst def>}}[\![i]\!]\right)$$

where $AllInst(ch)$ *is the set of all instance definitions in Basic Message Sequence Chart ch (see Section B.4.10.4 for its definition) and where* $S_{\texttt{<inst def>}}$ *is the semantic function for one instance in isolation.*

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. Since in the textual representation the instance on which the events are defined is not specified explicitly, the semantic functions for instance bodies and events are labelled with the name of the instance it originates from.

Definition B.4.4.4 *The semantic function for instance definitions,* $S_{\texttt{<inst def>}}:\mathcal{L}(\texttt{<inst def>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $i \in \mathcal{L}(\texttt{<inst def>})$ *defined by*

$$S_{\texttt{<inst def>}}[\![i]\!] \;=\; S_{\texttt{<inst body>}}(InstName(i))[\![InstBody(i)]\!]$$

where the functions InstName and InstBody assign to an instance definition its instance name and instance body (see Section B.4.10.4 for both definitions), and where $S_{\texttt{<inst body>}}$ *is the semantic function for instance bodies.*

Definition B.4.4.5 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for instance bodies,* $S_{\texttt{<inst body>}}(iid) : \mathcal{L}(\texttt{<inst body>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all event* $\in \mathcal{L}(\texttt{<event>})$ *and ibody* $\in \mathcal{L}(\texttt{<inst body>})$ *defined inductively by*

$$
\begin{array}{lcl}
S_{\texttt{<inst body>}}(iid)[\![\texttt{<>}]\!] & = & \varepsilon \\
S_{\texttt{<inst body>}}(iid)[\![event\ ibody]\!] & = & S_{\texttt{<event>}}(iid)[\![event]\!]\ \cdot\ S_{\texttt{<inst body>}}(iid)[\![ibody]\!]
\end{array}
$$

where $S_{\texttt{<event>}}$ *is the semantic function for events.*

The semantic function $S_{\texttt{<event>}}(iid)$ 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.

Definition B.4.4.6 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for events,* $S_{\texttt{<event>}}(iid) : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $at \in \mathcal{L}(\texttt{<at>})$, $m \in \mathcal{L}(\texttt{<mid>})$, $iid' \in \mathcal{L}(\texttt{<address>})$ *and* $p \in \mathcal{L}(\texttt{<par list>})$ *defined by*

$$
\begin{array}{lcl}
S_{\texttt{<event>}}(iid)[\![\texttt{action } at\texttt{;}]\!] & = & action(iid, at) \\
S_{\texttt{<event>}}(iid)[\![\texttt{out } m \texttt{ to } iid'\texttt{;}]\!] & = & out(iid, iid', m, \texttt{<>}) \\
S_{\texttt{<event>}}(iid)[\![\texttt{out } m\texttt{(}p\texttt{) to } iid'\texttt{;}]\!] & = & out(iid, iid', m, p) \\
S_{\texttt{<event>}}(iid)[\![\texttt{in } m \texttt{ from } iid'\texttt{;}]\!] & = & in(iid', iid, m, \texttt{<>}) \\
S_{\texttt{<event>}}(iid)[\![\texttt{in } m\texttt{(}p\texttt{) from } iid'\texttt{;}]\!] & = & in(iid', iid, m, p)
\end{array}
$$

Example B.4.4.7 The semantic function for Basic Message Sequence Charts will be illustrated by means of an example. Consider the Basic Message Sequence Chart in Figure B.9. It consists of three instances which exchange two messages.

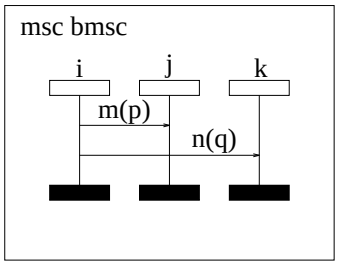

Figure B.9: Example Basic Message Sequence Chart

In the textual representation this Basic Message Sequence Chart is represented by

```
msc bmsc;
  instance i;
    out m(p) to j;
    out n(q) to k;
  endinstance;
  instance j;
    in m(p) from i;
  endinstance;
  instance k;
    in n(q) from i;
  endinstance;
endmsc;
```

The interpretation of this Basic Message Sequence Chart is that along instance i the ordering of the output of messages m and n is fixed and furthermore that the output of message m comes before the input of message m and, likewise, that the output of message n comes before the input of message n. 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 $S_{\texttt{<inst def>}}$ is an instance definition, but for brevity the semantics of an instance with name i is denoted by $S_{\texttt{<inst def>}}[\![i]\!]$. Applying the semantic function $S_{\texttt{<inst def>}}$ to these instances results in

$$S_{\texttt{<inst def>}}[\![i]\!] = out(i, j, m, p) \cdot out(i, k, n, q)$$

$S_{\texttt{<inst def>}}[\![j]\!] = in(i,j,m,p)$
$S_{\texttt{<inst def>}}[\![k]\!] = in(i,k,n,q)$

The first step in deriving the expression for the semantics of the Basic Message Sequence Chart is putting the instances i, j and k in parallel.

$$S_{\texttt{<inst def>}}[\![i]\!] \,||\, S_{\texttt{<inst def>}}[\![j]\!] \,||\, S_{\texttt{<inst def>}}[\![k]\!]$$

After some calculations, the following normalized expression results.

$$\begin{aligned}
&out(i,j,m,p) \cdot (in(i,j,m,p) \cdot (out(i,k,n,q) \cdot in(i,k,n,q) \\
&\qquad\qquad + in(i,k,n,q) \cdot out(i,k,n,q) \\
&\qquad\qquad) \\
&\qquad + out(i,k,n,q) \cdot (in(i,j,m,p) \cdot in(i,k,n,q) \\
&\qquad\qquad + in(i,k,n,q) \cdot in(i,j,m,p) \\
&\qquad\qquad) \\
&\qquad + in(i,k,n,q) \cdot (in(i,j,m,p) \cdot out(i,k,n,q) \\
&\qquad\qquad + out(i,k,n,q) \cdot in(i,j,m,p) \\
&\qquad\qquad) \\
&\qquad) \\
&+ in(i,j,m,p) \cdot (out(i,j,m,p) \cdot (in(i,k,n,q) \cdot out(i,k,n,q) \\
&\qquad\qquad + out(i,k,n,q) \cdot in(i,k,n,q) \\
&\qquad\qquad) \\
&\qquad + in(i,k,n,q) \cdot out(i,j,m,p) \cdot out(i,k,n,q) \\
&\qquad) \\
&+ in(i,k,n,q) \cdot (out(i,j,m,p) \cdot (in(i,j,m,p) \cdot out(i,k,n,q) \\
&\qquad\qquad + out(i,k,n,q) \cdot in(i,j,m,p) \\
&\qquad\qquad) \\
&\qquad + in(i,j,m,p) \cdot out(i,j,m,p) \cdot out(i,k,n,q) \\
&\qquad)
\end{aligned}$$

This expression clearly shows execution traces which are not desirable, such as $in(i,j,m,p) \cdot out(i,j,m,p) \cdot in(i,k,n,q) \cdot out(i,k,n,q)$. These traces are removed by applying the I/O-filter $\lambda_{\varnothing}$ to this expression. This results in

$$\begin{aligned}
&out(i,j,m,p) \cdot (in(i,j,m,p) \cdot out(i,k,n,q) \cdot in(i,k,n,q) \\
&\qquad + out(i,k,n,q) \cdot (in(i,j,m,p) \cdot in(i,k,n,q) \\
&\qquad\qquad + in(i,k,n,q) \cdot in(i,j,m,p) \\
&\qquad\qquad) \\
&\qquad)
\end{aligned}$$

B.4.5 Process creation and termination

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

The creation of an instance j by an instance i with parameter list p is reflected in the semantics by the atomic actions $create(i,j,p)$ and $start(j,p)$. The create atom is used to indicate the moment in time when instance i executes the create event, and the start atom is used to indicate the moment in time when instance j becomes active. The termination of an instance j is represented by the atomic action $stop(j)$.

Firstly, the process creation operator E_φ [3, 4], which will be used in defining the semantics of the process creation primitive in Message Sequence Charts, will be defined. The index φ is a function which maps an instance name into a process algebra expression. This function will define the created process.

Definition B.4.5.1 *Let* $\varphi : \mathcal{L}(\texttt{<inst name>}) \rightarrow T(\Sigma_{PA_{MSC}})$. *For* $a \in A$, $x \in V$, $i,j \in \mathcal{L}(\texttt{<inst name>})$ *and* $p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})$ *the process creation operator* $E_\varphi : T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$ *is defined in Table B.11.*

Table B.11: The process creation operator E_φ

$E_\varphi(\varepsilon) = \varepsilon$		CR1
$E_\varphi(\delta) = \delta$		CR2
$E_\varphi(a \cdot x) = a \cdot E_\varphi(x)$	if $a \notin A_c$	CR3
$E_\varphi(create(i,j,p) \cdot x) = create(i,j,p) \cdot E_\varphi(x \parallel start(j,p) \cdot \varphi(j))$		CR4
$E_\varphi(x + y) = E_\varphi(x) + E_\varphi(y)$		CR5

The crucial axiom defining E_φ (CR4) is explained below. If E_φ encounters a $create(i,j,p)$ action, an instance with name j has to be created. The description of the behavior of instance j is given by $\varphi(j)$. The behavior of instance j is preceded with the atomic action $start(j,p)$. This instance behavior is placed in parallel with x, the remaining behavior of the other instances. Since there may be more instances which are to be created the process creation operator is applied again.

Example B.4.5.2 The following computation illustrates the use of the process creation operator E_φ. Let $\varphi : \mathcal{L}(\texttt{<inst name>}) \rightarrow T(\Sigma_{PA_{MSC}})$ be a function such that $\varphi(j) = action(j,a)$. Then

$$\begin{aligned} E_\varphi(create(i,j,p)) &= create(i,j,p) \cdot E_\varphi(\varepsilon \parallel start(j,p) \cdot action(j,a)) \\ &= create(i,j,p) \cdot start(j,p) \cdot action(j,a) \end{aligned}$$

Next, the function φ will be instantiated by the function $sel(mscbody)$. This function determines the semantics of the instance definition associated to an instance name in the context of the Message Sequence Chart body *mscbody*. If no such instance definition is present in the Message Sequence Chart body, the empty process is taken.

Definition B.4.5.3 *Let* $mscbody \in \mathcal{L}(\texttt{<msc body>})$. *The function* $sel(mscbody) : \mathcal{L}(\texttt{<inst def>}) \rightarrow T(\Sigma_{PA_{MSC}})$ *is for all* $iid \in \mathcal{L}(\texttt{<inst name>})$ *and* $idef \in \mathcal{L}(\texttt{<inst def>})$ *defined inductively by*

$$\begin{aligned} sel(\texttt{<>})(iid) &= \varepsilon \\ sel(idef\ mscbody)(iid) &= S_{\texttt{<inst def>}}[\![idef]\!] && \text{if } InstName(idef) = iid \\ sel(idef\ mscbody)(iid) &= sel(mscbody)(iid) && \text{if } InstName(idef) \neq iid \end{aligned}$$

By the static requirements the process creation operator is only applied to instance names which correspond with an instance definition from the Message Sequence Chart body.

Definition B.4.5.4 *The semantic function for Basic Message Sequence Charts extended with the process creation and termination primitives,* $S_{\texttt{<msc>}} : \mathcal{L}(\texttt{<msc>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $ch \in \mathcal{L}(\texttt{<msc>})$ *defined by*

$$S_{\texttt{<msc>}}[\![ch]\!] = \lambda_\varnothing \left(E_{sel(ChBody(ch))} \left(\parallel_{i \in AllInst(ch) \setminus CreatedInsts(ch)} S_{\texttt{<inst def>}}[\![i]\!] \right) \right)$$

where the function ChBody associates to a chart its body, the function AllInst associates to a chart all its instance definitions, and the function CreatedInsts determines the set of created instances in the chart it is applied to (see Section B.4.10.4 for their definitions).

Since the process creation operator takes care of the dynamically created instances, the instances which should be considered merged in the first place are only those which are statically created.

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 a clause is added to the definition of the semantic function for instance bodies from Definition B.4.4.5.

Definition B.4.5.5 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for instance bodies,* $S_{\texttt{<inst body>}}(iid) : \mathcal{L}(\texttt{<inst body>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is extended with*

$$S_{\texttt{<inst body>}}(iid)[\![\texttt{stop;}]\!] \;=\; stop(iid)$$

The semantics of the create event is straightforward.

Definition B.4.5.6 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for events,* $S_{\texttt{<event>}}(iid) : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is extended with the semantics of the process creation event. For* $iid' \in \mathcal{L}(\texttt{<inst name>})$ *and* $p \in \mathcal{L}(\texttt{<par list>})$ *the definition is extended with*

$$\begin{array}{lcl} S_{\texttt{<event>}}(iid)[\![\texttt{create } iid'\texttt{;}]\!] & = & create(iid, iid', \texttt{<>}) \\ S_{\texttt{<event>}}(iid)[\![\texttt{create } iid'\texttt{(}p\texttt{);}]\!] & = & create(iid, iid', p) \end{array}$$

Example B.4.5.7 The semantic function for the Message Sequence Chart in Figure B.10 is calculated to illustrate the semantics of the creation and termination of an instance.

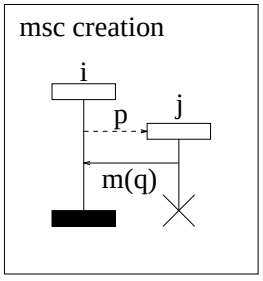

Figure B.10: Process creation and stop

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

```
msc creation;
  instance i;
    create j(p);
    in m(q) from j;
  endinstance;
  instance j;
    out m(q) to i;
    stop;
  endinstance;
endmsc;
```

The complete Message Sequence Chart is denoted by ch. First the semantics of every instance is calculated. This results in

$$S_{\texttt{<inst def>}}[\![i]\!] = create(i,j,p) \cdot in(j,i,m,q)$$
$$S_{\texttt{<inst def>}}[\![j]\!] = out(j,i,m,q) \cdot stop(j)$$

Combining these according to the definition of the semantic function results in

$$\lambda_{\varnothing}\left(E_{sel(ChBody(ch))}\left(\|_{k \in AllInst(msc) \setminus CreatedInsts(msc)} S_{\texttt{<inst def>}}[\![k]\!]\right)\right)$$

where

$$\begin{array}{lll} sel(ChBody(ch))(i) = & S_{\texttt{<inst def>}}[\![i]\!] & \\ sel(ChBody(ch))(j) = & S_{\texttt{<inst def>}}[\![j]\!] & \\ sel(ChBody(ch))(k) = & \varepsilon & \text{for all } k \in \mathcal{L}(\texttt{<inst name>}) \\ & & \text{such that } k \neq i \text{ and } k \neq j \end{array}$$

The semantics is calculated as follows

$$\begin{array}{ll} & \lambda_{\varnothing}\left(E_{sel(ChBody(ch))}\left(\|_{k \in AllInst(msc) \setminus CreatedInsts(msc)} S_{\texttt{<inst def>}}[\![k]\!]\right)\right) \\ = & \lambda_{\varnothing}(E_{sel(ChBody(ch))}(S_{\texttt{<inst def>}}[\![i]\!])) \\ = & \lambda_{\varnothing}(E_{sel(ChBody(ch))}(create(i,j,p) \cdot in(j,i,m,q))) \\ = & \lambda_{\varnothing}(create(i,j,p) \cdot E_{sel(ChBody(ch))}(in(j,i,m,q) \,\|\, start(j,p) \cdot sel(ChBody(ch))(j))) \\ = & \lambda_{\varnothing}(create(i,j,p) \cdot E_{sel(ChBody(ch))}(in(j,i,m,q) \,\|\, start(j,p) \cdot out(j,i,m,q) \cdot stop(j))) \\ = & create(i,j,p) \cdot start(j,p) \cdot out(j,i,m,q) \cdot \\ & \qquad (in(j,i,m,q) \cdot stop(j) + stop(j) \cdot in(j,i,m,q)) \end{array}$$

B.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. The timer events are included in the semantics by extending the semantic function for events.

Definition B.4.6.1 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for events,* $S_{\texttt{<event>}}(iid) : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $t \in \mathcal{L}(\texttt{<tid>})$ *and* $d \in \mathcal{L}(\texttt{<dn>})$ *extended with*

$$\begin{array}{lcl} S_{\texttt{<event>}}(iid)[\![\texttt{set } t\texttt{(}d\texttt{);}]\!] & = & set(iid,t,d) \\ S_{\texttt{<event>}}(iid)[\![\texttt{set } t\texttt{;}]\!] & = & set(iid,t) \\ S_{\texttt{<event>}}(iid)[\![\texttt{reset } t\texttt{;}]\!] & = & reset(iid,t) \\ S_{\texttt{<event>}}(iid)[\![\texttt{timeout } t\texttt{;}]\!] & = & timeout(iid,t) \end{array}$$

Example B.4.6.2 The semantics of Message Sequence Charts with timer handling is illustrated with the chart in Figure B.11. 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

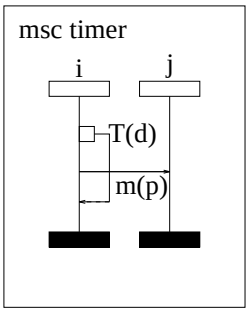

Figure B.11: Timer handling

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

Then the semantics of the instances are given by

$$\begin{aligned} S_{\texttt{<inst def>}}[\![i]\!] &= set(i,T,d)\cdot out(i,j,m,p)\cdot reset(i,T) \\ S_{\texttt{<inst def>}}[\![j]\!] &= in(i,j,m,p) \end{aligned}$$

Since the chart has no created instances, the semantics is calculated as follows:

$$\lambda_{\varnothing}((set(i,T,d)\cdot out(i,j,m,p)\cdot reset(i,T))\,\|\,(in(i,j,m,p)))$$

which equals

$$set(i,T,d)\cdot out(i,j,m,p)\cdot(reset(i,T)\cdot in(i,j,m,p)+in(i,j,m,p)\cdot reset(i,T))$$

B.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.

Definition B.4.7.1 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for events,* $S_{\texttt{<event>}}(iid) : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all coregion* $\in \mathcal{L}(\texttt{<coregion>})$ *extended with*

$$S_{\texttt{<event>}}(iid)[\![coregion]\!] = \|_{\,e\in CoEvents(\,coregion)}\, S_{\texttt{<event>}}(iid)[\![e]\!]$$

where the function CoEvents associates to a coregion the set of all message events defined in the coregion (see Section B.4.10.4 for its definition).

Example B.4.7.2 Consider the Message Sequence Chart in Figure B.12. 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.

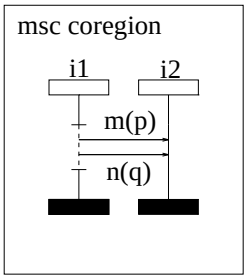

Figure B.12: Coregion

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

```
msc coregion;
   instance i1;
      concurrent
         out m(p) to i2;
         out n(q) to i2;
      endconcurrent;
   endinstance;
   instance i2;
      in m(p) from i1;
      in n(q) from i1;
   endinstance;
endmsc;
```

```
msc coregion;
   instance i1;
      concurrent
         out n(q) to i2;
         out m(p) to i2;
      endconcurrent;
   endinstance;
   instance i2;
      in m(p) from i1;
      in n(q) 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

$$S_{\texttt{<inst def>}}[\![i1]\!] \;=\; out(i1, i2, m, p) \,||\, out(i1, i2, n, q)$$

and the semantics of instance $i2$ is given by

$$S_{\texttt{<inst def>}}[\![i2]\!] \;=\; in(i1, i2, m, p) \cdot in(i1, i2, n, q)$$

The semantics of the complete chart is given by the following expression

$$\lambda_{\varnothing}((out(i1, i2, m, p) \,||\, out(i1, i2, n, q)) \,||\, (in(i1, i2, m, p) \cdot in(i1, i2, n, q)))$$

which equals

$$\begin{aligned} &out(i1, i2, m, p) \cdot (out(i1, i2, n, q) \cdot in(i1, i2, m, p) \cdot in(i1, i2, n, q) \\ &\qquad\qquad + in(i1, i2, m, p) \cdot out(i1, i2, n, q) \cdot in(i1, i2, n, q) \\ &\qquad\qquad) \\ &+ out(i1, i2, n, q) \cdot out(i1, i2, m, p) \cdot in(i1, i2, m, p) \cdot in(i1, i2, n, q) \end{aligned}$$

B.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 rôle in the static semantics. With a condition no dynamic behavior is associated. Thus the following extension of the semantic function for events is given.

Definition B.4.8.1 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function,* $S_{\texttt{<event>}}(iid) : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *for events, is for all condition* $\in \mathcal{L}(\texttt{<condition>})$ *extended with*

$$S_{\texttt{<event>}}(iid)[\![condition]\!] = \varepsilon$$

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

B.4.9 Refinement of instances

B.4.9.1 Introduction

In a Message Sequence Chart an instance may be refined by decomposing it into a Sub Message Sequence Chart with the same name. 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 exterior of the Sub Message Sequence Chart on the other hand. This relation is determined by means of message identifier identification.

The general idea of the instance refinement principle in Message Sequence Charts is the following: a decomposed instance is replaced by the corresponding Sub Message Sequence Chart and the dangling communication events are connected via message identifier identification. Consider for example the two charts in Figure B.13. The result of replacing decomposed instance *d* in Message Sequence Chart *ex* by the Sub Message Sequence Chart *d* is given in Figure B.14. The desired connections between dangling communications are marked with a dotted line.

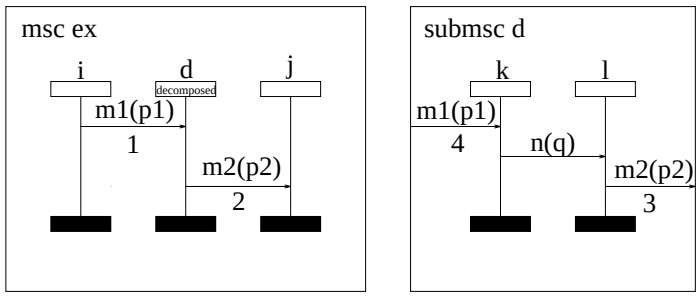

Figure B.13: Instance refinement

The above sketched intuition will be formalized in the next two sections. In Section B.4.9.2 an operation *refine* for the refinement of instances by Sub Message Sequence Charts is considered. In Section B.4.9.3 the semantics of a Message Sequence Chart with decomposed instances is defined using this *refine* operation.

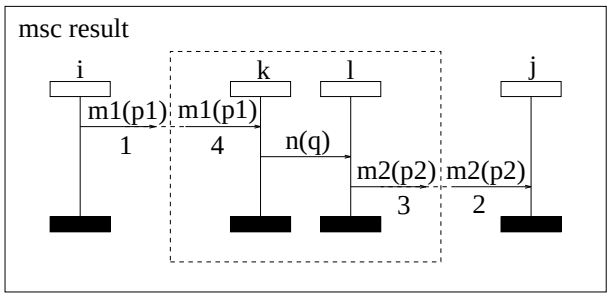

Figure B.14: Replacing the decomposed instance by a Sub Message Sequence Chart

B.4.9.2 Refinement of an instance

In this section the refinement of one instance by a Sub Message Sequence Chart is considered. Suppose that a Message Sequence Chart ch with an instance d is given. The semantics of this Message Sequence Chart is denoted by S. Suppose furthermore that instance d is refined by a Sub Message Sequence Chart D (with name d). The semantics of this Sub Message Sequence Chart is denoted by S'. The refinement of the instance d by the Sub Message Sequence Chart D is carried out in several phases.

As a first step the decomposed instance d is replaced by the Sub Message Sequence Chart D. This is achieved by deleting all atomic actions which are defined on instance d, and placing the resulting process in parallel with the semantics of the Sub Message Sequence Chart.

Next, the connections between the dangling communication events of the Message Sequence Chart and the Sub Message Sequence Chart must be provided. First, all references to decomposed instance d in the Message Sequence Chart must be replaced by the name of one of the instances of the Sub Message Sequence Chart. The only atomic actions which reference decomposed instances are the message inputs from and the message outputs to the decomposed instance.

1) Suppose that a message $m1$ is sent by an instance, say i, to d ($out(i, d, m1, p1)$). This situation corresponds with the arrow labelled with 1 in Figure B.14. Then on decomposed instance d the corresponding message input has to be specified. By the static requirements there also has to be an input of message $m1$ specified on one of the instances, say k, of the Sub Message Sequence Chart. Then instance k is the actual receiver of message $m1$. So, the atomic action $out(i, d, m1, p1)$ must be replaced by $out(i, k, m1, p1)$.

2) Suppose that a message $m2$ is received by an instance, say j, from d ($in(d, j, m2, p2)$). This situation corresponds with the arrow labelled 2 in Figure B.14. Then on decomposed instance d the corresponding message output has to be specified. By the static requirements there also has to be an output of message $m2$ specified on one of the instances, say l, of the Sub Message Sequence Chart. Then instance l is the actual sender of message $m2$. So, the atomic action $in(d, j, m2, p2)$ must be replaced by $in(l, j, m2, p2)$.

Besides the replacement discussed above, also every occurrence of the environment in the Sub Message Sequence Chart must be replaced 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.

3) Suppose that a message $m2$ is sent by an instance l of the Sub Message Sequence Chart to the exterior ($out(l, env, m2, p2)$). This situation corresponds with the arrow labelled 3 in Figure B.14. By the static requirements there has to be an output of a message $m2$ specified

on decomposed instance d. This message is either received by the environment or by another instance, say j, of the Message Sequence Chart. So, the atomic action $out(l, env, m2, p2)$ must be replaced by $out(l, j, m2, p2)$ in the case that the message is received by an instance j of the Message Sequence Chart. In the case that the message is received by the environment of the Message Sequence Chart no replacement is needed.

4) Suppose that a message $m1$ is received by an instance k of the Sub Message Sequence Chart from the exterior ($in(env, k, m1, p1)$). This situation corresponds with the arrow labelled 4 in Figure B.14. By the static requirements there has to be an input of message $m1$ specified on decomposed instance d. This message is either sent by the environment or by another instance, say i, of the Message Sequence Chart. So, the atomic action $in(env, k, m1, p1)$ must be replaced by $in(i, k, m1, p1)$ in the case that the message is sent by an instance i of the Message Sequence Chart. In the case that the message is sent to the environment of the Message Sequence Chart no replacement is needed.

This construction is formalized below.

First, the replacement of the decomposed instance by the Sub Message Sequence Chart is discussed. This can be achieved by

- removing the events which are defined on the decomposed instance;
- merging the result of the previous step with the instances from the Sub Message Sequence Chart.

In order to remove the events which are defined on the decomposed instance d, those must be recognized as such. Every atomic action in the semantics has as one of its parameters the instance it is defined on. The function D (see Definition B.4.3.2) is used in determining whether an atomic action is defined on instance d. A set $A(d)$ which consists of all atomic actions that are defined on an instance with name d is defined as follows.

Definition B.4.9.2.1 *Let $d \in \mathcal{L}$(`<inst name>`). The set $A(d)$ is defined by*

$$A(d) \;=\; \{a \in A \mid D(a) = d\}$$

For the removal of the atomic actions from a set $I \subseteq A$ from a process the special renaming operator ε_I [2] is used. This operator renames all occurrences of the atomic actions from I in the process it is applied to into the empty process ε.

Definition B.4.9.2.2 *Let $I \subseteq A$. For $x, y \in V$ and $a \in A$ the renaming operator $\varepsilon_I : T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$ is defined in Table B.12.*

For the removal of the actions which are defined on the decomposed instance d from the semantics S of a Message Sequence Chart this operator with the set I instantiated by $A(d)$ is applied to S:

$$\varepsilon_{A(d)}(S)$$

Placing this in parallel with the semantics S' of the Sub Message Sequence Chart results in:

$$\lambda_{\varnothing}(\varepsilon_{A(d)}(S) \parallel S')$$

Since the processes are placed in parallel the I/O-filter is applied.

Table B.12: Axioms for the renaming operator ε_I.

$\varepsilon_I(\varepsilon) = \varepsilon$	
$\varepsilon_I(\delta) = \delta$	
$\varepsilon_I(a \cdot x) = a \cdot \varepsilon_I(x)$	if $a \notin I$
$\varepsilon_I(a \cdot x) = \varepsilon_I(x)$	if $a \in I$
$\varepsilon_I(x + y) = \varepsilon_I(x) + \varepsilon_I(y)$	

Next, the dangling communication events of the Message Sequence Chart and the Sub Message Sequence Chart must be connected. The dangling communications of the Message Sequence Charts are characterized by their reference to decomposed instance d, and the dangling communications of the Sub Message Sequence Chart by their reference to the environment.

All occurrences of the decomposed instance name d in the expression $\varepsilon_{A(d)}(S)$ and all occurrences of the environment in the expression S' must be replaced by the correct sender and receiver names. To compute these sender and receiver names the functions rec_m and sen_m are defined. The function rec_m determines the name of the instance on which a message m is received and the function sen_m determines the name of the instance by which a message m is sent.

Definition B.4.9.2.3 *Let* $m \in \mathcal{L}$(<mid>). *For* $a \in A$, $x, y \in V$, $i \in \mathcal{L}$(<inst name>), $j \in \mathcal{L}$(<address>), $n \in \mathcal{L}$(<mid>) *and* $p \in \mathcal{L}_{<>}$(<par list>), *the functions* $rec_m : T(\Sigma_{PA_{MSC}}) \rightarrow \mathcal{L}$(<address>) *and* $sen_m : T(\Sigma_{PA_{MSC}}) \rightarrow \mathcal{L}$(<address>) *are defined in Table B.13.*

Table B.13: The functions rec_m and sen_m

$rec_m(\varepsilon) = env$	
$rec_m(\delta) = env$	
$rec_m(a \cdot x) = rec_m(x)$	if $a \notin A_i$
$rec_m(in(j, i, n, p) \cdot x) = i$	if $n = m$
$rec_m(in(j, i, n, p) \cdot x) = rec_m(x)$	if $n \neq m$
$rec_m(x + y) = rec_m(x)$	if $rec_m(x) = rec_m(y)$
$rec_m(x + y) = env$	if $rec_m(x) \neq rec_m(y)$
$sen_m(\varepsilon) = env$	
$sen_m(\delta) = env$	
$sen_m(a \cdot x) = sen_m(x)$	if $a \notin A_o$
$sen_m(out(i, j, n, p) \cdot x) = i$	if $n = m$
$sen_m(out(i, j, n, p) \cdot x) = sen_m(x)$	if $n \neq m$
$sen_m(x + y) = sen_m(x)$	if $sen_m(x) = sen_m(y)$
$sen_m(x + y) = env$	if $sen_m(x) \neq sen_m(y)$

The function rec_m traces the process it is applied to for the first occurrence of an input of message m. It returns the name of the instance this input is defined on. If no such input is encountered, the message must have been sent to the environment, so the value env is returned. It is required that the different traces of the process return the same value. This is expressed by the last two axioms. In the context of Message Sequence Charts this is indeed the case. The function sen_m essentially

behaves like rec_m but it looks for the name of the instance on which an output of message m is defined.

The four situations, indicated in Figures B.13 and B.14, in which a replacement of an atomic action by another one is required, are formalized below. First, the dangling communication events of the Message Sequence Chart are considered.

1) Given a message $m1$ being sent by instance i to decomposed instance d, the actual receiver name is computed by tracing the receiver instance of message $m1$ in the semantics S' of the Sub Message Sequence Chart. So, the receiver instance is given by $rec_{m1}(S')$ and the atomic action $out(i, d, m1, p1)$ must be replaced by the atomic action $out(i, rec_{m1}(S'), m1, p1)$.

2) Analogous, given a message $m2$ being received by instance j which is sent by decomposed instance d, the actual sender name is computed by tracing the sender instance of message $m2$ in the semantics S' of the Sub Message Sequence Chart. So, the sender instance is given by $sen_{m2}(S')$ and the atomic action $in(d, j, m2, p2)$ must be replaced by the atomic action $in(sen_{m2}(S'), j, m2, p2)$.

For the replacement of these atomic actions the function $\rho_{ext}(d, s)$ is used. This function, the external renaming operator, renames all communication actions which refer to instance name d into the same communication action with the correct instance name in place of d. The correct instance name is traced in the process s. The external renaming operator is an instantiation of the global renaming operator ρ_f [1].

Definition B.4.9.2.4 *Let* $d \in \mathcal{L}$(<inst name>) *and* $s \in T(\Sigma_{PA_{MSC}})$. *The function* $\rho_{ext}(d, s) : A \rightarrow A$ *is for all* $a \in A$, $i \in \mathcal{L}$(<inst name>), $m \in \mathcal{L}$(<mid>) *and* $p \in \mathcal{L}_{<>}$(<par list>) *defined in Table B.14. Furthermore, this function is extended to processes in the obvious way. For* $x, y \in V$, *the external renaming operator* $\rho_{ext}(d, s) : T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$ *is defined in Table B.14.*

Table B.14: The external renaming operator $\rho_{ext}(d, s)$

$\rho_{ext}(d,s)(out(i,d,m,p)) = out(i, rec_m(s), m, p)$ $\rho_{ext}(d,s)(in(d,i,m,p)) = in(sen_m(s), i, m, p)$ $\rho_{ext}(d,s)(a) = a$ otherwise
$\rho_{ext}(d,s)(\varepsilon) = \varepsilon$ $\rho_{ext}(d,s)(\delta) = \delta$ $\rho_{ext}(d,s)(a \cdot x) = \rho_{ext}(d,s)(a) \cdot \rho_{ext}(d,s)(x)$ $\rho_{ext}(d,s)(x + y) = \rho_{ext}(d,s)(x) + \rho_{ext}(d,s)(y)$

The first three axioms state that the renaming operator ρ_{ext} changes the receiver (or sender) of a message into the corresponding receiver (or sender) of that message in the term s. The last four axioms ensure that this renaming is carried out for the complete process expression.

Next, the replacement of the dangling communication events of the Sub Message Sequence Chart is discussed.

3) Given a message $m2$ being sent by an instance l of the Sub Message Sequence Chart to the exterior, the actual receiver instance is computed by tracing the input of message

$m2$ in the semantics S of the Message Sequence Chart. The receiver instance is given by $rec_{m2}(S)$ and the atomic action $out(l, env, m2, p2)$ must be replaced by the atomic action $out(l, rec_{m2}(S), m2, p2)$.

4) Analogously, given a message $m1$ being received by an instance k of the Sub Message Sequence Chart from the exterior, the actual sender instance is computed by tracing the output of the message $m1$ in the semantics S of the Message Sequence Chart. The sender instance is given by $sen_{m1}(S)$ and the atomic action $in(env, k, m1, p1)$ must be replaced by the atomic action $in(sen_{m1}(S), k, m1, p1)$.

For the replacement of these atomic actions the function $\rho_{int}(S)$ is used. This function, the *internal renaming operator*, renames all communication actions which refer to the environment into the same communication action with the correct instance name in place of the environment. This correct instance name is traced in the process S. The internal renaming operator is also an instantiation of the global renaming operator.

Definition B.4.9.2.5 *Let $s \in T(\Sigma_{PA_{MSC}})$. The function $\rho_{int}(s) : A \to A$ is for all $a \in A$, $i \in \mathcal{L}$(<inst name>), $m \in \mathcal{L}$(<mid>) and $p \in \mathcal{L}_{<>}$(<par list>) defined in Table B.15. Furthermore, this function is extended to processes in the obvious way. For $x, y \in V$, the internal renaming operator $\rho_{int}(s) : T(\Sigma_{PA_{MSC}}) \to T(\Sigma_{PA_{MSC}})$ is defined in Table B.15.*

Table B.15: The internal renaming operator $\rho_{int}(s)$

$\rho_{int}(s)(out(i, env, m, p)) = out(i, rec_m(s), m, p)$ $\rho_{int}(s)(in(env, i, m, p)) = in(sen_m(s), i, m, p)$ $\rho_{int}(s)(a) = a$ otherwise
$\rho_{int}(s)(\varepsilon) = \varepsilon$ $\rho_{int}(s)(\delta) = \delta$ $\rho_{int}(s)(a \cdot x) = \rho_{int}(s)(a) \cdot \rho_{int}(s)(x)$ $\rho_{int}(s)(x + y) = \rho_{int}(s)(x) + \rho_{int}(s)(y)$

The first three axioms state that the renaming operator ρ_{int} changes the sender (or receiver) of a message into the corresponding sender (or receiver) of that message in the term s. The last four axioms ensure that this renaming is carried out for the complete process expression.

Now, the renamings which are introduced must be applied to the processes $\varepsilon_{A(d)}(S)$ and S' to provide for the connection of the dangling communications. This results in the processes $\rho_{ext}(d, S')(\varepsilon_{A(d)}(S))$ and $\rho_{int}(S')(S)$, which must be placed in parallel.

If the semantics S of a Message Sequence Chart and the semantics S' of a Sub Message Sequence Chart which refines an instance d of the Message Sequence Chart are given, the semantics of the Message Sequence Chart with decomposed instance d is obtained from the following expression:

$$\lambda_{\varnothing}(\rho_{ext}(d, S')(\varepsilon_{A(d)}(S)) \,\|\, \rho_{int}(S)(S'))$$

Next, an operator *refine* is introduced which expresses the refinement of an instance on a semantic level. The expression $refine(d, x, y)$ denotes the result of refining instance d within process y by process x.

Definition B.4.9.2.6 *The function refine* : $\mathcal{L}(\texttt{<inst name>}) \times T(\Sigma_{PA_{MSC}}) \times T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $d \in \mathcal{L}(\texttt{<inst name>})$ *and* $x, y \in T(\Sigma_{PA_{MSC}})$ *defined by*

$$refine(d, x, y) \;=\; \lambda_{\varnothing}(\rho_{ext}(d, x)(\varepsilon_{A(d)}(y)) \,||\, \rho_{int}(y)(x))$$

B.4.9.3 Semantics of Message Sequence Charts with decomposed instances

The definition of the semantic function for a Message Sequence Chart with decomposed instances will use the *refine* operation from the previous section. The semantics of a Message Sequence Chart is depending on the semantics of the Sub Message Sequence Charts corresponding with the decomposed instances. The semantics of a Sub Message Sequence Chart is determined in the same way as the semantics of a Message Sequence Chart is determined. The semantic function $S_{\texttt{<msc>}}$ (see Definition B.4.5.4) is extended to Sub Message Sequence Charts by changing the domain.

Definition B.4.9.3.1 *The semantic function* $S_{\texttt{<msc>}} : \mathcal{L}(\texttt{<chart>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $ch \in \mathcal{L}(\texttt{<chart>})$ *defined by*

$$S_{\texttt{<msc>}}[\![ch]\!] \;=\; \lambda_{\varnothing}\left(E_{sel(ChBody(ch))}\left(\,||_{\,i \in AllInst(ch) \setminus CreatedInsts(ch)}\, S_{\texttt{<inst def>}}[\![i]\!]\right)\right)$$

where the function ChBody associates to a chart its body, the function AllInst associates to a chart all its instance definitions, and the function CreatedInsts determines the set of created instances in the chart it is applied to (see Section B.4.10.4 for their definitions).

A Message Sequence Chart and its corresponding Sub Message Sequence Charts are contained in a Message Sequence Chart document. For a Message Sequence Chart document *doc* the function $ChNamed(doc)$ determines the chart associated to a given chart name.

Definition B.4.9.3.2 *Let* $doc \in \mathcal{L}(\texttt{<msc doc>})$. *The function* $ChNamed(doc) : \mathcal{L}(\texttt{<msc name>}) \rightarrow \mathcal{L}(\texttt{<chart>})$ *is for all* $mscname \in \mathcal{L}(\texttt{<msc name>})$ *defined by*

$$ChNamed(doc)(mscname) \;=\; ch \qquad \text{if } ChName(ch) = mscname \wedge ch \in Charts(doc)$$

and is undefined otherwise. The function ChName associates to a chart its name (See Section B.4.10.4 for its definition).

Given the semantics of a chart *ch* with all decomposed instances considered non-decomposed ($S_{\texttt{<msc>}}[\![ch]\!]$, Definition B.4.9.3.1), the semantics of this chart is obtained by refining the decomposed instances one by one. This is achieved by repeatedly applying the function *refine*.

Suppose that *ch* has decomposed instances with names $d_1, \cdots, d_n$. Suppose that the semantics of the Sub Message Sequence Chart corresponding with decomposed instance d_i is given by D_i. Then the semantics of the chart *ch* is given by

$$refine(d_1, S(doc)[\![D_1]\!], refine(d_2, S(doc)[\![D_2]\!], \cdots, refine(d_n, S(doc)[\![D_n]\!], S_{\texttt{<msc>}}[\![ch]\!]) \cdots))$$

This is formalized in the following definitions. Given a chart *ch*, the function $refinements(doc)(ch)$ determines the decomposed instance names together with their corresponding Sub Message Sequence Charts.

Definition B.4.9.3.3 *Let* $doc \in \mathcal{L}(\texttt{<msc doc>})$. *The function* $refinements(doc) : \mathcal{L}(\texttt{<chart>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst name>}) \times \mathcal{L}(\texttt{<submsc>}))$ *is for all* $ch \in \mathcal{L}(\texttt{<chart>})$ *defined by*

$$refinements(doc)(ch) \;=\; \{(d, ChNamed(doc)(d)) \mid d \in DecInstNames(ch)\}$$

where the function DecInstNames associates to a chart the names of all its decomposed instances (See Section B.4.10.4 for its definition).

The semantics $S(doc)[\![ch]\!]$ of a chart ch in a Message Sequence Chart document doc is then defined inductively as follows.

Definition B.4.9.3.4 *Let* $doc \in \mathcal{L}(\texttt{<msc doc>})$. *The semantic functions* $S(doc) : \mathcal{L}(\texttt{<chart>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *and* $S'(doc) : I\!P(\mathcal{L}(\texttt{<inst name>}) \times \mathcal{L}(\texttt{<submsc>})) \rightarrow T(\Sigma_{PA_{MSC}})$ *are, for all* $ch \in \mathcal{L}(\texttt{<chart>})$, $d \in \mathcal{L}(\texttt{<inst name>})$, $D \in \mathcal{L}(\texttt{<submsc>})$, *and* $R \subseteq I\!P(\mathcal{L}(\texttt{<inst name>}) \times \mathcal{L}(\texttt{<submsc>}))$ *defined by*

$$S(doc)[\![ch]\!] = S'(doc)(refinements(doc)(ch))[\![ch]\!]$$

$$S'(doc)(\varnothing)[\![ch]\!] = S_{\texttt{<msc>}}[\![ch]\!]$$

$$S'(doc)(R \cup \{(d, D)\})[\![ch]\!] = refine(d, S(doc)[\![D]\!], S'(doc)(R \backslash \{(d, D)\})[\![ch]\!])$$

In order to obtain the semantics of a chart ch first the refinements are determined. The auxiliary function $S'(doc)$ actually computes the semantics. If there is no refinement, then the semantics is computed analogously to the semantics of a chart without decomposed instances, i.e. $S_{\texttt{<msc>}}$. If there is a refinement of a decomposed instance d by a Sub Message Sequence Chart D, then decomposed instance d is replaced by the semantics $S(doc)[\![D]\!]$ of the Sub Message Sequence Chart D by applying the function *refine*. This is repeated until there are no refinements left.

Example B.4.9.3.5 In Figure B.15 a Message Sequence Chart with a decomposed instance d and a Sub Message Sequence Chart that refines the decomposed instance are given. The Message Sequence Chart document in which these two charts are defined is denoted by doc, the Message Sequence Chart *decinst* is denoted by C, and the Sub Message Sequence Chart d is denoted by D.

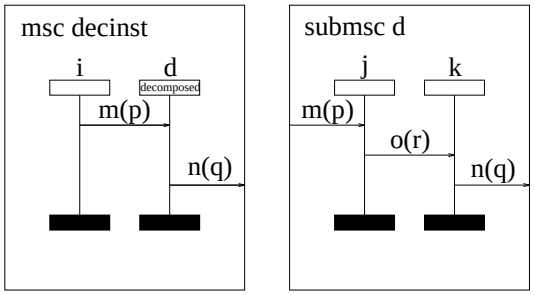

Figure B.15: Refinement of decomposed instances

In the textual representation these charts are represented by

```
msc decinst;                    submsc d;
   instance i;                     instance j;
      out m(p) to d;                  in m(p) from env;
   endinstance;                       out o(r) to k;
   instance d decomposed;          endinstance;
      in m(p) from i;              instance k;
```

```
        out n(q) to env;                in o(r) from j;
      endinstance;                      out n(q) to env;
endmsc;                               endinstance;
                                    endsubmsc;
```

The semantics of this Message Sequence Chart is given by

$$\begin{aligned} S(doc)[\![C]\!] &= S'(doc)(refinements(doc)(C))[\![C]\!] \\ &= S'(doc)(\{(d, D)\})[\![C]\!] \\ &= refine(d, S(doc)[\![D]\!], S'(doc)(\varnothing)[\![C]\!]) \\ &= refine(d, S(doc)[\![D]\!], S_{\texttt{<msc>}}[\![C]\!]) \\ &= \lambda_{\varnothing}(\rho_{ext}(d, S(doc)[\![D]\!])(\varepsilon_{A(d)}(S_{\texttt{<msc>}}[\![C]\!])) \,||\, \rho_{int}(S_{\texttt{<msc>}}[\![C]\!])(S(doc)[\![D]\!])) \end{aligned}$$

where $S_{\texttt{<msc>}}[\![C]\!]$ (abbreviated by T) is the semantics of Message Sequence Chart *decinst* with all decomposed instances considered non-decomposed, and where $S(doc)[\![D]\!]$ (abbreviated by U) is the semantics of Sub Message Sequence Chart D. These are given by

$$T = S_{\texttt{<msc>}}[\![C]\!] = out(i, d, m, p) \cdot in(i, d, m, p) \cdot out(d, env, n, q)$$

and

$$U = S(doc)[\![D]\!] = in(env, j, m, p) \cdot out(j, k, o, r) \cdot in(j, k, o, r) \cdot out(k, env, n, q)$$

Next, the expression $\rho_{ext}(d, U)(\varepsilon_{A(d)}(T))$ is computed.

$$\begin{aligned} \rho_{ext}(d, U)(\varepsilon_{A(d)}(T)) &= \rho_{ext}(d, U)(out(i, d, m, p)) \\ &= out(i, rec_m(U), m, p) \\ &= out(i, j, m, p) \end{aligned}$$

The expression $\rho_{int}(T)(U)$ is computed.

$$\begin{aligned} \rho_{int}(T)(U) &= \rho_{int}(T)(in(env, j, m, p) \cdot out(j, k, o, r) \cdot in(j, k, o, r) \cdot out(k, env, n, q)) \\ &= \rho_{int}(T)(in(env, j, m, p)) \cdot \rho_{int}(T)(out(j, k, o, r)) \\ &\quad \cdot \rho_{int}(T)(in(j, k, o, r)) \cdot \rho_{int}(T)(out(k, env, n, q)) \\ &= in(sen_m(T), j, m, p) \cdot out(j, k, o, r) \cdot in(j, k, o, r) \cdot out(k, rec_n(T), n, q) \\ &= in(i, j, m, p) \cdot out(j, k, o, r) \cdot in(j, k, o, r) \cdot out(k, env, n, q) \end{aligned}$$

So, the semantics of the Message Sequence Chart is given by

$$\lambda_{\varnothing}(out(i, j, m, p) \,||\, in(i, j, m, p) \cdot out(j, k, o, r) \cdot in(j, k, o, r) \cdot out(k, env, n, q))$$

which equals

$$out(i, j, m, p) \cdot in(i, j, m, p) \cdot out(j, k, o, r) \cdot in(j, k, o, r) \cdot out(k, env, n, q)$$

In Figure B.16 a Message Sequence Chart without decomposed instances is presented. This Message Sequence Chart is semantically equivalent to the Message Sequence Charts in Figure B.15.

B.4.10 Overview of the complete semantics

The semantics of the complete language is summarized below. A detailed explanation can be found in the Sections B.4.1 – B.4.9. The process algebra PA_{ε} is given in Section B.3. The semantics of a chart *ch* in a Message Sequence Chart document *doc* is defined by $S(doc)[\![ch]\!]$.

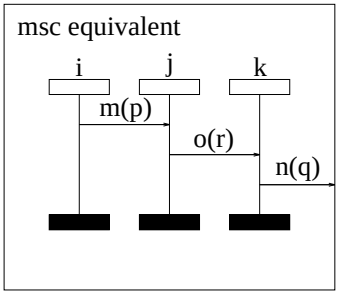

Figure B.16: Semantically equivalent Message Sequence Chart

B.4.10.1 Semantic function

Definition B.4.10.1.1 *Let* $doc \in \mathcal{L}(\texttt{<msc doc>})$. *The semantic functions* $S(doc) : \mathcal{L}(\texttt{<chart>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *and* $S'(doc) : I\!P(\mathcal{L}(\texttt{<inst name>}) \times \mathcal{L}(\texttt{<submsc>})) \rightarrow T(\Sigma_{PA_{MSC}})$ *are, for all* $ch \in \mathcal{L}(\texttt{<chart>})$, $d \in \mathcal{L}(\texttt{<inst name>})$, $D \in \mathcal{L}(\texttt{<submsc>})$, *and* $R \subseteq I\!P(\mathcal{L}(\texttt{<inst name>}) \times \mathcal{L}(\texttt{<submsc>}))$ *defined by*

$$S(doc)[\![ch]\!] \;=\; S'(doc)(refinements(doc)(ch))[\![ch]\!]$$

$$S'(doc)(\varnothing)[\![ch]\!] \;=\; S_{\texttt{<msc>}}[\![ch]\!]$$

$$S'(doc)(R \cup \{(d,D)\})[\![ch]\!] \;=\; refine(d, S(doc)[\![D]\!], S'(doc)(R \backslash \{(d,D)\})[\![ch]\!])$$

Definition B.4.10.1.2 *The semantic function for single charts,* $S_{\texttt{<msc>}} : \mathcal{L}(\texttt{<chart>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $ch \in \mathcal{L}(\texttt{<chart>})$ *defined by*

$$S_{\texttt{<msc>}}[\![ch]\!] \;=\; \lambda_{\varnothing}\left(E_{sel(ChBody(ch))}\left(\|_{i \in AllInst(ch) \backslash CreatedInsts(ch)} S_{\texttt{<inst def>}}[\![i]\!]\right)\right)$$

Definition B.4.10.1.3 *The semantic function for instance definitions,* $S_{\texttt{<inst def>}} : \mathcal{L}(\texttt{<inst def>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all* $i \in \mathcal{L}(\texttt{<inst def>})$ *defined by*

$$S_{\texttt{<inst def>}}[\![i]\!] \;=\; S_{\texttt{<inst body>}}(InstName(i))[\![InstBody(i)]\!]$$

Definition B.4.10.1.4 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function for instance bodies,* $S_{\texttt{<inst body>}}(iid) : \mathcal{L}(\texttt{<inst body>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *is for all event* $\in \mathcal{L}(\texttt{<event>})$ *and ibody* $\in \mathcal{L}(\texttt{<inst body>})$ *defined inductively by*

$$\begin{array}{lcl} S_{\texttt{<inst body>}}(iid)[\![\texttt{<>}]\!] & = & \varepsilon \\ S_{\texttt{<inst body>}}(iid)[\![\texttt{stop;}]\!] & = & stop(iid) \\ S_{\texttt{<inst body>}}(iid)[\![event\ ibody]\!] & = & S_{\texttt{<event>}}(iid)[\![event]\!] \cdot S_{\texttt{<inst body>}}(iid)[\![ibody]\!] \end{array}$$

Definition B.4.10.1.5 *Let* $iid \in \mathcal{L}(\texttt{<inst name>})$. *The semantic function,* $S_{\texttt{<event>}}(iid) : \mathcal{L}(\texttt{<event>}) \rightarrow T(\Sigma_{PA_{MSC}})$, *for events, is for all at* $\in \mathcal{L}(\texttt{<at>})$, $iid' \in \mathcal{L}(\texttt{<address>})$, $m \in \mathcal{L}(\texttt{<mid>})$, $iid'' \in \mathcal{L}(\texttt{<inst name>})$, $p \in \mathcal{L}(\texttt{<par list>})$, $t \in \mathcal{L}(\texttt{<tid>})$, *coregion* $\in \mathcal{L}(\texttt{<coregion>})$, $d \in \mathcal{L}(\texttt{<dn>})$, *and condition* $\in \mathcal{L}(\texttt{<condition>})$ *defined by*

$$
\begin{array}{lcl}
S_{\texttt{<event>}}(iid)[\![\texttt{action } at\texttt{;}]\!] & = & action(iid, at) \\
S_{\texttt{<event>}}(iid)[\![\texttt{out } m \texttt{ to } iid'\texttt{;}]\!] & = & out(iid, iid', m, \texttt{<>}) \\
S_{\texttt{<event>}}(iid)[\![\texttt{out } m\texttt{(}p\texttt{) to } iid'\texttt{;}]\!] & = & out(iid, iid', m, p) \\
S_{\texttt{<event>}}(iid)[\![\texttt{in } m \texttt{ from } iid'\texttt{;}]\!] & = & in(iid', iid, m, \texttt{<>}) \\
S_{\texttt{<event>}}(iid)[\![\texttt{in } m\texttt{(}p\texttt{) from } iid'\texttt{;}]\!] & = & in(iid', iid, m, p) \\
S_{\texttt{<event>}}(iid)[\![\texttt{create } iid''\texttt{;}]\!] & = & create(iid, iid'', \texttt{<>}) \\
S_{\texttt{<event>}}(iid)[\![\texttt{create } iid''\texttt{(}p\texttt{);}]\!] & = & create(iid, iid'', p) \\
S_{\texttt{<event>}}(iid)[\![\texttt{set } t\texttt{(}d\texttt{);}]\!] & = & set(iid, t, d) \\
S_{\texttt{<event>}}(iid)[\![\texttt{set } t\texttt{;}]\!] & = & set(iid, t) \\
S_{\texttt{<event>}}(iid)[\![\texttt{reset } t\texttt{;}]\!] & = & reset(iid, t) \\
S_{\texttt{<event>}}(iid)[\![\texttt{timeout } t\texttt{;}]\!] & = & timeout(iid, t) \\
S_{\texttt{<event>}}(iid)[\![coregion]\!] & = & \|_{e \in CoEvents(coregion)} S_{\texttt{<event>}}(iid)[\![e]\!] \\
S_{\texttt{<event>}}(iid)[\![condition]\!] & = & \varepsilon
\end{array}
$$

B.4.10.2 Additional process algebra operators

Definition B.4.10.2.1 *In Table B.16 the sets of atomic actions A_a, A_o, A_{oe}, A_i, A_{ie}, A_c, A_{st}, A_s, A_t and their union A are defined. For a nonterminal* X, $\mathcal{L}_{\texttt{<>}}(\texttt{X})$ *is used to denote* $\mathcal{L}(\texttt{X}) \cup \{\texttt{<>}\}$.

Table B.16: The atomic actions of PA_{MSC}

$$
\begin{array}{lcl}
A_a & = & \{action(i, at) \mid i \in \mathcal{L}(\texttt{<inst name>}), at \in \mathcal{L}(\texttt{<at>})\} \\
A_o & = & \{out(i, j, m, p) \mid i, j \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})\} \\
A_{oe} & = & \{out(i, env, m, p) \mid i \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})\} \\
A_i & = & \{in(i, j, m, p) \mid i, j \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})\} \\
A_{ie} & = & \{in(env, i, m, p) \mid i \in \mathcal{L}(\texttt{<inst name>}), m \in \mathcal{L}(\texttt{<mid>}), p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})\} \\
A_c & = & \{create(i, j, p) \mid i, j \in \mathcal{L}(\texttt{<inst name>}), p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})\} \\
A_{st} & = & \{start(i, p) \mid i \in \mathcal{L}(\texttt{<inst name>}), p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})\} \\
A_s & = & \{stop(i) \mid i \in \mathcal{L}(\texttt{<inst name>})\} \\
A_t & = & \{set(i, t, d) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>}), d \in \mathcal{L}(\texttt{<dn>})\} \\
 & & \cup \{set(i, t) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>})\} \\
 & & \cup \{reset(i, t) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>})\} \\
 & & \cup \{timeout(i, t) \mid i \in \mathcal{L}(\texttt{<inst name>}), t \in \mathcal{L}(\texttt{<tid>})\} \\
\hline
A & = & A_a \cup A_o \cup A_{oe} \cup A_i \cup A_{ie} \cup A_c \cup A_{st} \cup A_s \cup A_t
\end{array}
$$

Definition B.4.10.2.2 *Let* $M \subseteq \mathcal{L}(\texttt{<mid>})$. *For all* $x, y \in V$, $a \in A$, $i, j \in \mathcal{L}(\texttt{<inst name>})$, $m \in \mathcal{L}(\texttt{<mid>})$ *and* $p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})$, *the I/O-filter* λ_M *is defined in Table B.17.*

Definition B.4.10.2.3 *Let* $\varphi : \mathcal{L}(\texttt{<inst name>}) \rightarrow T(\Sigma_{PA_{MSC}})$. *For* $a \in A$, $x \in V$, $i, j \in \mathcal{L}(\texttt{<inst name>})$ *and* $p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})$ *the process creation operator* $E_\varphi : T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$ *is defined in Table B.18.*

Definition B.4.10.2.4 *Let* $I \subseteq A$. *For* $x, y \in V$ *and* $a \in A$ *the renaming operator* $\varepsilon_I : T(\Sigma_{PA_{MSC}}) \rightarrow T(\Sigma_{PA_{MSC}})$ *is defined in Table B.19.*

Table B.17: The I/O-filter λ_M

$\lambda_M(\varepsilon) = \varepsilon$	if $M = \varnothing$	LM1
$\lambda_M(\varepsilon) = \delta$	if $M \neq \varnothing$	LM2
$\lambda_M(\delta) = \delta$		LM3
$\lambda_M(a \cdot x) = a \cdot \lambda_M(x)$	if $a \notin A_o \cup A_i$	LM4
$\lambda_M(out(i,j,m,p) \cdot x) = \delta$	if $m \in M$	LM5
$\lambda_M(out(i,j,m,p) \cdot x) = out(i,j,m,p) \cdot \lambda_{M \cup \{m\}}(x)$	if $m \notin M$	LM6
$\lambda_M(in(i,j,m,p) \cdot x) = in(i,j,m,p) \cdot \lambda_{M \setminus \{m\}}(x)$	if $m \in M$	LM7
$\lambda_M(in(i,j,m,p) \cdot x) = \delta$	if $m \notin M$	LM8
$\lambda_M(x + y) = \lambda_M(x) + \lambda_M(y)$		LM9

Table B.18: The process creation operator E_φ

$E_\varphi(\varepsilon) = \varepsilon$		CR1
$E_\varphi(\delta) = \delta$		CR2
$E_\varphi(a \cdot x) = a \cdot E_\varphi(x)$	if $a \notin A_c$	CR3
$E_\varphi(create(i,j,p) \cdot x) = create(i,j,p) \cdot E_\varphi(x \parallel start(j,p) \cdot \varphi(j))$		CR4
$E_\varphi(x + y) = E_\varphi(x) + E_\varphi(y)$		CR5

Definition B.4.10.2.5 *Let $d \in \mathcal{L}$(<inst name>) and $s \in T(\Sigma_{PA_{MSC}})$. The function $\rho_{ext}(d,s) : A \to A$ is for all $a \in A$, $i \in \mathcal{L}$(<inst name>), $m \in \mathcal{L}$(<mid>) and $p \in \mathcal{L}_{<>}$(<par list>) defined in Table B.14. Furthermore, this function is extended to processes in the obvious way. For $x, y \in V$, the external renaming operator $\rho_{ext}(d,s) : T(\Sigma_{PA_{MSC}}) \to T(\Sigma_{PA_{MSC}})$ is defined in Table B.20.*

Definition B.4.10.2.6 *Let $s \in T(\Sigma_{PA_{MSC}})$. The function $\rho_{int}(s) : A \to A$ is for all $a \in A$, $i \in \mathcal{L}$(<inst name>), $m \in \mathcal{L}$(<mid>) and $p \in \mathcal{L}_{<>}$(<par list>) defined in Table B.15. Furthermore, this function is extended to processes in the obvious way. For $x, y \in V$, the internal renaming operator $\rho_{int}(s) : T(\Sigma_{PA_{MSC}}) \to T(\Sigma_{PA_{MSC}})$ is defined in Table B.21.*

Definition B.4.10.2.7 *The function refine : $\mathcal{L}$(<inst name>) $\times\ T(\Sigma_{PA_{MSC}}) \times T(\Sigma_{PA_{MSC}}) \to T(\Sigma_{PA_{MSC}})$, is for all $d \in \mathcal{L}$(<inst name>) and $x, y \in T(\Sigma_{PA_{MSC}})$ defined by*

$$refine(d,x,y) = \lambda_{\varnothing}(\rho_{ext}(d,x)(\varepsilon_{A(d)}(y)) \parallel \rho_{int}(y)(x))$$

Table B.19: Axioms for the renaming operator ε_I.

$\varepsilon_I(\varepsilon) = \varepsilon$	
$\varepsilon_I(\delta) = \delta$	
$\varepsilon_I(a \cdot x) = a \cdot \varepsilon_I(x)$	if $a \notin I$
$\varepsilon_I(a \cdot x) = \varepsilon_I(x)$	if $a \in I$
$\varepsilon_I(x + y) = \varepsilon_I(x) + \varepsilon_I(y)$	

Table B.20: The external renaming operator $\rho_{ext}(d, s)$

$\rho_{ext}(d,s)(out(i,d,m,p)) = out(i, rec_m(s), m, p)$ $\rho_{ext}(d,s)(in(d,i,m,p)) = in(sen_m(s), i, m, p)$ $\rho_{ext}(d,s)(a) = a$ otherwise
$\rho_{ext}(d,s)(\varepsilon) = \varepsilon$ $\rho_{ext}(d,s)(\delta) = \delta$ $\rho_{ext}(d,s)(a \cdot x) = \rho_{ext}(d,s)(a) \cdot \rho_{ext}(d,s)(x)$ $\rho_{ext}(d,s)(x + y) = \rho_{ext}(d,s)(x) + \rho_{ext}(d,s)(y)$

Table B.21: The internal renaming operator $\rho_{int}(s)$

$\rho_{int}(s)(out(i,env,m,p)) = out(i, rec_m(s), m, p)$ $\rho_{int}(s)(in(env,i,m,p)) = in(sen_m(s), i, m, p)$ $\rho_{int}(s)(a) = a$ otherwise
$\rho_{int}(s)(\varepsilon) = \varepsilon$ $\rho_{int}(s)(\delta) = \delta$ $\rho_{int}(s)(a \cdot x) = \rho_{int}(s)(a) \cdot \rho_{int}(s)(x)$ $\rho_{int}(s)(x + y) = \rho_{int}(s)(x) + \rho_{int}(s)(y)$

B.4.10.3 Auxiliary definitions

Definition B.4.10.3.1 *The function* $D : A \to \mathcal{L}(\texttt{<inst name>})$ *is for all* $i, j \in \mathcal{L}(\texttt{<inst name>})$, $at \in \mathcal{L}(\texttt{<at>})$, $t \in \mathcal{L}(\texttt{<tid>})$, $d \in \mathcal{L}(\texttt{<dn>})$, $m \in \mathcal{L}(\texttt{<mid>})$ *and* $p \in \mathcal{L}_{\texttt{<>}}(\texttt{<par list>})$ *defined in Table B.22.*

Table B.22: The function D

$D(action(i,at))$	$=$	i	$D(start(i,p))$	$=$	i
$D(out(i,j,m,p))$	$=$	i	$D(stop(i))$	$=$	i
$D(out(i,env,m,p))$	$=$	i	$D(set(i,t,d))$	$=$	i
$D(in(i,j,m,p))$	$=$	j	$D(set(i,t))$	$=$	i
$D(in(env,j,m,p))$	$=$	j	$D(reset(i,t))$	$=$	i
$D(create(i,j,p))$	$=$	i	$D(timeout(i,t))$	$=$	i

Definition B.4.10.3.2 *Let* $mscbody \in \mathcal{L}(\texttt{<msc body>})$. *The function* $sel(mscbody) : \mathcal{L}(\texttt{<inst name>}) \to T(\Sigma_{PA_{MSC}})$ *is for all* $iid \in \mathcal{L}(\texttt{<inst name>})$ *and* $idef \in \mathcal{L}(\texttt{<inst def>})$ *defined inductively by*

$$sel(\texttt{<>})(iid) = \varepsilon$$

$$sel(idef\ mscbody)(iid) = S_{\texttt{<inst def>}}[\![idef]\!] \qquad \text{if } InstName(idef) = iid$$
$$sel(idef\ mscbody)(iid) = sel(mscbody)(iid) \qquad \text{if } InstName(idef) \neq iid$$

Definition B.4.10.3.3 *Let* $d \in \mathcal{L}$(<inst name>)*. The set* $A(d)$ *is defined by*

$$A(d) = \{a \in A \mid D(a) = d\}$$

Definition B.4.10.3.4 *Let* $m \in \mathcal{L}$(<mid>)*. For* $a \in A$, $x, y \in V$, $i \in \mathcal{L}$(<inst name>), $j \in \mathcal{L}$(<address>), $n \in \mathcal{L}$(<mid>) *and* $p \in \mathcal{L}_{\texttt{<>}}$(<par list>)*, the functions* $rec_m : T(\Sigma_{PA_{MSC}}) \rightarrow \mathcal{L}$(<address>) *and* $sen_m : T(\Sigma_{PA_{MSC}}) \rightarrow \mathcal{L}$(<address>) *are defined in Table B.23.*

Table B.23: The functions rec_m and sen_m

$rec_m(\varepsilon) = env$	
$rec_m(\delta) = env$	
$rec_m(a \cdot x) = rec_m(x)$	if $a \notin A_i$
$rec_m(in(j,i,n,p) \cdot x) = i$	if $n = m$
$rec_m(in(j,i,n,p) \cdot x) = rec_m(x)$	if $n \neq m$
$rec_m(x + y) = rec_m(x)$	if $rec_m(x) = rec_m(y)$
$rec_m(x + y) = env$	if $rec_m(x) \neq rec_m(y)$
$sen_m(\varepsilon) = env$	
$sen_m(\delta) = env$	
$sen_m(a \cdot x) = sen_m(x)$	if $a \notin A_o$
$sen_m(out(i,j,n,p) \cdot x) = i$	if $n = m$
$sen_m(out(i,j,n,p) \cdot x) = sen_m(x)$	if $n \neq m$
$sen_m(x + y) = sen_m(x)$	if $sen_m(x) = sen_m(y)$
$sen_m(x + y) = env$	if $sen_m(x) \neq sen_m(y)$

Definition B.4.10.3.5 *Let* $doc \in \mathcal{L}$(<msc doc>)*. The function* $ChNamed(doc) : \mathcal{L}$(<msc name>) $\rightarrow \mathcal{L}$(<chart>) *is for all* $mscname \in \mathcal{L}$(<msc name>) *defined by*

$$ChNamed(doc)(mscname) = ch \qquad \text{if } ChName(ch) = mscname \wedge ch \in Charts(doc)$$

and is undefined otherwise.

Definition B.4.10.3.6 *Let* $doc \in \mathcal{L}$(<msc doc>)*. The function* $refinements(doc) : \mathcal{L}$(<chart>) $\rightarrow$ $I\!P(\mathcal{L}$(<inst name>) $\times \mathcal{L}$(<submsc>)) *is for all* $ch \in \mathcal{L}$(<chart>) *defined by*

$$refinements(doc)(ch) = \{(d, ChNamed(doc)(d)) \mid d \in DecInstNames(ch)\}$$

B.4.10.4 Auxiliary functions for the semantics of Message Sequence Charts

In this section the auxiliary functions used in defining the semantics of Message Sequence Charts in Section B.4 are defined.

The notation $I\!P(X)$ denotes the power set of a set X, i.e. the set of all subsets of X. The empty set is denoted by $\varnothing$. The set $I\!B$ contains the Boolean values *true* and *false*.

For all $doc \in \mathcal{L}(\texttt{<msc doc>})$, $docname \in \mathcal{L}(\texttt{<doc name>})$, $docbody \in \mathcal{L}(\texttt{<doc body>})$, $ch \in \mathcal{L}(\texttt{<chart>})$, $mscname \in \mathcal{L}(\texttt{<msc name>})$, $mscbody \in \mathcal{L}(\texttt{<msc body>})$, $instdef \in \mathcal{L}(\texttt{<inst def>})$, $instbody \in \mathcal{L}(\texttt{<inst body>})$, $event \in \mathcal{L}(\texttt{<event>})$, $iid \in \mathcal{L}(\texttt{<inst name>})$, $action \in \mathcal{L}(\texttt{<action>})$, $out \in \mathcal{L}(\texttt{<out>})$, $in \in \mathcal{L}(\texttt{<in>})$, $set \in \mathcal{L}(\texttt{<set>})$, $reset \in \mathcal{L}(\texttt{<reset>})$, $timeout \in \mathcal{L}(\texttt{<timeout>})$, $coregion \in \mathcal{L}(\texttt{<coregion>})$, $coevents \in \mathcal{L}(\texttt{<coevents>})$ and $condition \in \mathcal{L}(\texttt{<condition>})$ the following functions are defined

$AllInst : \mathcal{L}(\texttt{<msc body>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst def>}))$
$AllInst : \mathcal{L}(\texttt{<chart>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst def>}))$

$AllInst(\texttt{<>}) = \varnothing$
$AllInst(instdef\ mscbody) = \{instdef\} \cup AllInst(mscbody)$
$AllInst(ch) = AllInst(ChBody(ch))$

$Charts : \mathcal{L}(\texttt{<doc body>}) \rightarrow I\!P(\mathcal{L}(\texttt{<chart>}))$
$Charts : \mathcal{L}(\texttt{<msc doc>}) \rightarrow I\!P(\mathcal{L}(\texttt{<chart>}))$

$Charts(\texttt{<>}) = \varnothing$
$Charts(ch\ docbody) = \{ch\} \cup Charts(docbody)$
$Charts(doc) = Charts(DocBody(doc))$

$ChBody : \mathcal{L}(\texttt{<chart>}) \rightarrow \mathcal{L}(\texttt{<msc body>})$

$ChBody(\texttt{msc}\ mscname\texttt{;}\ mscbody\ \texttt{endmsc;}) = mscbody$
$ChBody(\texttt{submsc}\ mscname\texttt{;}\ mscbody\ \texttt{endsubmsc;}) = mscbody$

$ChName : \mathcal{L}(\texttt{<chart>}) \rightarrow \mathcal{L}(\texttt{<msc name>})$

$ChName(\texttt{msc}\ mscname\texttt{;}\ mscbody\ \texttt{endmsc;}) = mscname$
$ChName(\texttt{submsc}\ mscname\texttt{;}\ mscbody\ \texttt{endsubmsc;}) = mscname$

$CoEvents : \mathcal{L}(\texttt{<coevents>}) \rightarrow I\!P(\mathcal{L}(\texttt{<in>}|\texttt{<out>}))$
$CoEvents : \mathcal{L}(\texttt{<coregion>}) \rightarrow I\!P(\mathcal{L}(\texttt{<in>}|\texttt{<out>}))$

$CoEvents(\texttt{<>}) = \varnothing$
$CoEvents(in\ coevents) = \{in\} \cup CoEvents(coevents)$
$CoEvents(out\ coevents) = \{out\} \cup CoEvents(coevents)$
$CoEvents(\texttt{concurrent}\ coevents\ \texttt{endconcurrent;}) = CoEvents(coevents)$

$CreatedInsts : \mathcal{L}(\texttt{<chart>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst def>}))$

$CreatedInsts(ch) = \{instdef \in AllInst(ch) \mid InstName(instdef) \in CrInstNames(ch)\}$

$CrInstNames : \mathcal{L}(\texttt{<event>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst name>}))$
$CrInstNames : \mathcal{L}(\texttt{<inst body>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst name>}))$
$CrInstNames : \mathcal{L}(\texttt{<inst def>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst name>}))$
$CrInstNames : \mathcal{L}(\texttt{<chart>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst name>}))$

$CrInstNames(in) = \varnothing$
$CrInstNames(out) = \varnothing$

$CrInstNames(\texttt{create}\ iid\texttt{(}p\texttt{);}) = \{iid\}$
$CrInstNames(set) = \varnothing$
$CrInstNames(reset) = \varnothing$
$CrInstNames(timeout) = \varnothing$
$CrInstNames(coregion) = \varnothing$
$CrInstNames(action) = \varnothing$
$CrInstNames(condition) = \varnothing$
$CrInstNames(\texttt{<>}) = \varnothing$
$CrInstNames(\texttt{stop;}) = \varnothing$
$CrInstNames(event\ instbody) = CrInstNames(event) \cup CrInstNames(instbody)$
$CrInstNames(instdef) = CrInstNames(InstBody(instdef))$
$CrInstNames(ch) = \bigcup_{instdef \in AllInst(ch)} CrInstNames(instdef)$

$DecInstNames : \mathcal{L}(\texttt{<chart>}) \rightarrow I\!P(\mathcal{L}(\texttt{<inst name>}))$

$DecInstNames(ch) =$
$\{InstName(instdef) \mid instdef \in AllInst(ch) \wedge IsDecomposed(instdef)\}$

$DocBody : \mathcal{L}(\texttt{<msc doc>}) \rightarrow \mathcal{L}(\texttt{<doc body>})$

$DocBody(\texttt{mscdocument}\ docname\texttt{;}\ docbody\ \texttt{endmscdocument;}) = docbody$

$InstBody : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathcal{L}(\texttt{<inst body>})$

$InstBody(\texttt{instance}\ iid\texttt{;}\ instbody\ \texttt{endinstance;}) = instbody$
$InstBody(\texttt{instance}\ iid\ \texttt{decomposed;}\ instbody\ \texttt{endinstance;}) = instbody$

$InstName : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathcal{L}(\texttt{<inst name>})$

$InstName(\texttt{instance}\ iid\texttt{;}\ instbody\ \texttt{endinstance;}) = iid$
$InstName(\texttt{instance}\ iid\ \texttt{decomposed;}\ instbody\ \texttt{endinstance;}) = iid$

$IsDecomposed : \mathcal{L}(\texttt{<inst def>}) \rightarrow I\!B$

$IsDecomposed(\texttt{instance}\ iid\texttt{;}\ instbody\ \texttt{endinstance;}) = false$
$IsDecomposed(\texttt{instance}\ iid\ \texttt{decomposed;}\ instbody\ \texttt{endinstance;}) = true$

B.5 Concrete textual syntax

In this section the concrete textual syntax from the Recommendation is adapted to support the definition of the formal semantics of Message Sequence Charts. Since some syntactic parts of a Message Sequence Chart are of no importance to the semantics, those are left out. These are the references to SDL, the Message Sequence Chart interface, the instance kinds and the text definitions. Also the Message Sequence Chart and Sub Message Sequence Chart diagrams are left out.

Because of the long names for the nonterminals, the concrete textual syntax is not suited for manipulation in the definition of the semantic functions and auxiliary functions. Therefore, abbreviations for the nonterminals are introduced in Table B.24.

Table B.24: Abbreviations for nonterminals

nonterminal	abbreviation
<message sequence chart document>	<msc doc>
<document body>	<doc body>
<msc document name>	<doc name>
<document body>	<doc body>
<message sequence chart>	<msc>
<message sequence chart name>	<msc name>
<instance name>	<inst name>
<instance definition>	<inst def>
<instance body>	<inst body>
<message input>	<in>
<message output>	<out>
<msg identification>	<msgid>
<message name>	<mn>
<message instance name>	<min>
<parameter list>	<par list>
<parameter name>	<par name>
<condition name>	<cn>
<shared instance list>	<shared inst list>
<timer name>	<tn>
<timer instance name>	<tin>
<duration name>	<dn>
<action text>	<at>

The grammar for Message Sequence Charts as presented in the main text of the Recommendation is not suited for the inductive definition of functions on nonterminals. Therefore, an equivalent grammar, which does not contain the `*`, is given in Table B.25. This means that the grammar generates the same language with respect to the nonterminal `<msc doc>`.

The nonterminals `<at>`, `<cn>`,`<dn>`, `<inst name>`, `<min>`, `<mn>`, `<msc name>`, `<par name>`, `<tin>` and `<tn>` represent identifiers. The symbol `<>` denotes the empty string. The following identifiers are the reserved keywords: `action`, `all`, `concurrent`, `condition`, `create`, `decomposed`, `endconcurrent`, `endinstance`, `endmsc`, `endmscdocument`, `endsubmsc`, `env`, `from`, `in`, `instance`, `msc`, `mscdocument`, `out`, `reset`, `set`, `shared`, `stop`, `submsc`, `timeout` and `to`.

Table B.25: Alternative grammar for Message Sequence Charts

```
<msc doc>                ::=  mscdocument <doc name>;
                                        <doc body>
                              endmscdocument;
<doc body>               ::=  <> | <chart> <doc body>
<chart>                  ::=  <msc> | <submsc>
<msc>                    ::=  msc <msc name>; <msc body> endmsc;
<msc body>               ::=  <> | <inst def> <msc body>
<inst def>               ::=  instance <inst name> [decomposed];
                                      <inst body>
                              endinstance;
<inst body>              ::=  <> | stop; | <event> <inst body>
<event>                  ::=  <out> | <in> | <action> | <create> | <set> |
                              <reset> | <timeout> | <coregion> | <condition>
<out>                    ::=  out <msgid> to <address>;
<msgid>                  ::=  <mid> [(<par list>)]
<mid>                    ::=  <mn> [,<min>]
<par list>               ::=  <par name> [,<par list>]
<address>                ::=  <inst name> | env
<in>                     ::=  in <msgid> from <address>;
<action>                 ::=  action <at>;
<create>                 ::=  create <inst name> [(<par list>)];
<set>                    ::=  set <tid> [(<dn>)];
<tid>                    ::=  <tn> [,<tin>]
<reset>                  ::=  reset <tid>;
<timeout>                ::=  timeout <tid>;
<coregion>               ::=  concurrent <coevents> endconcurrent;
<coevents>               ::=  <> | <out> <coevents> | <in> <coevents>
<condition>              ::=  condition <cn> [shared {<shared inst list> | all}];
<shared inst list>       ::=  <inst name> [,<shared inst list>]
<submsc>                 ::=  submsc <msc name>; <msc body> endsubmsc;
```

Bibliography

[1] J. C. M. Baeten and J. A. Bergstra. Global renaming operators in concrete process algebra. *Information and Computation*, 78, 1988.

[2] J. C. M. Baeten and R. J. van Glabbeek. Merge and termination in process algebra. In K. V. Nori, editor, *Proc. Conf. on Foundations of Software Technology and Theoretical Computer Science VII*, Lecture Notes in Computer Science 287, Pune, 1987. Springer, Berlin.

[3] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, Cambridge, 1990.

[4] J. A. Bergstra. A process creation mechanism in process algebra. In J. C. M. Baeten, editor, *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science 17. Cambridge University Press, Cambridge, 1990.

[5] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information & Control*, 60:109–137, 1984.

[6] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[7] S. Mauw and M. A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4), 1994.

[8] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer, Berlin, 1980.

List of Figures

List of Tables

Index

This index contains different kinds of entries:

- nonterminals from the concrete grammar (e.g. `<create>`)
- keywords from the concrete grammar (e.g. `create`)
- function names (e.g. *CreatedInsts*)
- glossary items (e.g. create)