INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.120

Annex B

(04/1998)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Message Sequence Chart (MSC)

Message Sequence Chart

Annex B: Formal semantics of Message Sequence Charts

ITU-T Recommendation Z.120 – Annex B

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	**Z.120–Z.129**
Extended Object Definition Language (eODL)	Z.130–Z.139
Tree and Tabular Combined Notation (TTCN)	Z.140–Z.149
User Requirements Notation (URN)	Z.150–Z.159
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-computer interfaces for the management of telecommunications networks	Z.360–Z.369
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Distributed processing environment	Z.600–Z.609

For further details, please refer to the list of ITU-T Recommendations.

ITU-T Recommendation Z.120

Message Sequence Chart
Annex B
Formal Semantics of Message Sequence Charts

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 definition of the semantics of Message Sequence Charts. Examples are added to explain the formal definitions.

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 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. This annex describes the semantics of a Message Sequence Chart, a High-level Message Sequence Chart and a Message Sequence Chart document. Substructure references and substitution are not covered in this semantics.

Associated work

Recommendation Z.120: Message Sequence Charts (MSC).

Source

Annex B to ITU-T Recommendation Z.120 was prepared by ITU-T Study Group 10 (1997-2000) and approved under the WTSC Resolution 1 procedure on 1 April 1998.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSC Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU [had/had not] received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

List of Figures

List of Tables

Annex B to Recommendation Z.120

Formal 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 behaviour 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 formal semantics is based on [MR97, MR98]. 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 compositional way. The semantics of a composite MSC is formulated in terms of the composites and the means of composing.

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. This section is subdivided into four parts. In the first part the core language (Basic Message Sequence Charts) is introduced. In the second part other basic concepts are added one by one. The third part extends the language with ordering facilities such as coregions and general orderings. The last part introduces the means offered by MSC to compose MSCs. These means are inline expressions, MSC reference expressions and High-level Message Sequence Charts.

Section B.3 contains an informal introduction on the extension of the Message Sequence Chart language with gates.

In Section B.4 the formal framework for the definition of the semantics is defined. This framework consists of process expressions that are built from a number of constants and operators. These constants and operators are defined by means of term deduction rules. These describe the actions that can be performed by a process.

In order to facilitate the definition of a formal semantics the textual syntax of MSCs has been transformed. These transformations as well as the textual syntax actually used for the definition of the formal semantics are given in Section B.5.

The semantic functions are defined in Section B.6. The main function maps every Message Sequence Chart into an expression over the operators introduced in the Section B.4. The philosophy that has led to these definitions is explained first.

B.2 Message Sequence Charts

B.2.1 Introduction

In this section the language MSC is introduced. The language is best illustrated by the graphical representation, but where the definition of a formal semantics is concerned, the textual representation is preferred. The order in which the features of MSC are introduced differs from the order in which they are defined in the main text of the Recommendation. The order used in this annex is based on the way the features are treated in the formal semantics.

First the core language of Message Sequence Charts is introduced. This core language is called *Basic Message Sequence Chart*. 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.6, 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, incomplete message events, and conditions.

Then the ordering facilities are introduced. These are coregions and causal orderings. Finally, the more intricate possibilities of describing complex systems are considered. These are inline expressions, MSC reference expressions and High-level Message Sequence Charts.

B.2.2 Basic Message Sequence Charts

The body of a Basic Message Sequence Chart is formed by a finite collection of instances. An instance is an abstract entity on which message outputs, message inputs and local actions may be specified. A first example of a Basic Message Sequence Chart is given in Figure B.1.

Next the graphical representation of Basic Message Sequence Charts is explained. Then their meaning is described, and finally the textual representation is introduced.

B.2.2.1 Graphical representation

Graphically an MSC is given by a frame containing the instances. The name of the MSC following the keyword **msc** is placed inside this frame, usually above the instances. For an example see Figure B.1.

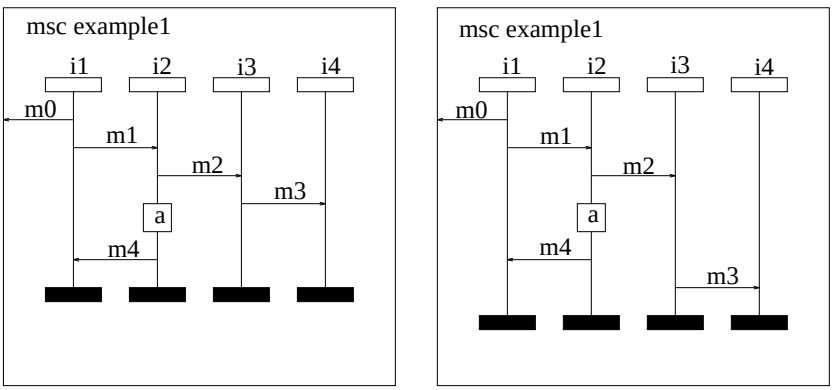


Figure B.1: Example Basic Message Sequence Charts

In the graphical representation there are two ways to draw an instance. These are given in Figure B.2 below. The first is a single vertical axis (line-form) and the second is the so-called column-form. The description

of the instance starts with the *instance head symbol* and ends with the *instance end symbol*. These do not describe creation and termination of the instance, but the start and end of the description. The representation of the instance and the instance head and instance end symbols should be aligned as indicated in Figure B.2. Within one Basic Message Sequence Chart both representations of instances, line-form and column-form, may appear.

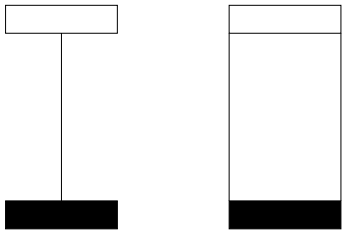


Figure B.2: Instance symbols: line-form and column-form

With every instance an *instance name* is associated. The instance name may be placed above or inside the instance head symbol. Instances are referred to by means of the instance name. Therefore, the instance name must be unique within an MSC.

A local action is denoted by an *action symbol* on an instance with the *action character string* placed in it. A local action describes internal activity of an instance. The action character string is an informal description for this internal activity. When an action symbol is placed on an instance in line-form the instance axis is "removed". If the column-form is used, the width of the action symbol must coincide with the width of the column-form of the instance. Multiple occurrences of an action symbol on an instance must not overlap. See Figure B.3 for examples.

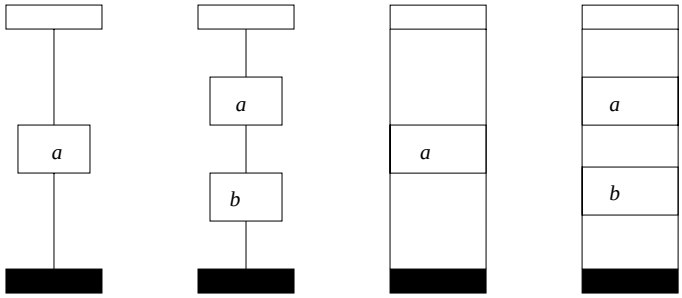


Figure B.3: Placement of local actions on line-form and column-form instances

A message between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. An arrow representing a message may be horizontal or with downward slope. 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, i.e., the surrounding frame. A message received from the environment is represented by an arrow from the exterior of the Message Sequence Chart to the receiving instance. With every message a *message name* is associated. The message name should be placed close to the message arrow.

In principle it is not allowed to have two or more events attached to one point of the instance axis in line-form and column-form or at the same height of the instance axes in the column-form. However, there is one exception to this rule. An *incoming event* and an *outgoing event* may be attached to the same point or at the same height. This is interpreted as if the incoming event is drawn above the outgoing event. Message output events, lost message events, process creation events and timer set and reset events are outgoing events and message input events, found message events and timeout events are incoming events.

B.2.2.2 Intuitive semantics

An MSC is intended to describe a number of executions of the events contained. These events can be local actions, message outputs and message inputs. An MSC does not only describe the events to be executed, it also contains information on the order in which they can be executed. One of the basic assumptions is that all events are executed instantaneously, i.e., it is assumed that the execution of an event consumes no time. Another important assumption is that no two events can be executed at the same time.

As explained before, an MSC consists of a number of instances on which events are specified. The meaning of such an instance is that it executes the events specified in the same order as they are given on the vertical axis from top to bottom. Thus one can say that the time along each instance axis is running from top to bottom. Therefore, the events specified on an instance are totally ordered in time. Consider, for example, instance $i2$ from the MSC given in Figure B.1, then this means that instance $i2$ executes the events "input of $m1$ from instance j", "output of $m2$ to instance $i3$", "action a", and "output of $m4$ to instance $i1$", and also that these events are executed in this order. Although an instance describes the execution of events while time progresses, the instance does not specify the elapse of time in between two consecutive events. It might be the case that the first event is executed at 5 minutes and that the second event is executed at 25 minutes.

The instances of an MSC in principle operate independently of each other. No global notion of time is assumed. The only dependencies between the timing of the instances come from the restriction that a message must be sent before it is received. In Figure B.1 this implies that message $m3$ is received by $i4$ only after it has been sent by $i3$, and, consequently, after the consumption of $m2$ by $i3$. Thus 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. Because of the asynchronous communication, it would even be possible to first send $m3$, then send and receive $m4$, and finally receive $m3$. 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 execution sequences (from a semantic point of view), but in an alternative drawing.

Another consequence of this mode of communication is that overtaking of messages is allowed, as expressed in Figure B.4.

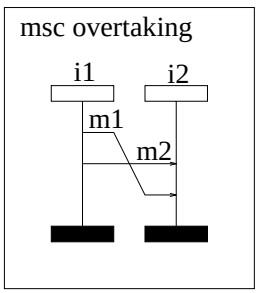


Figure B.4: Basic Message Sequence Chart with overtaking

It is not allowed that a message output is causally depending on its corresponding message input, directly or via other messages [IT96b, IT96a, Ren95]. 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. Such MSCs are often called *inconsistent*.

Consider the first diagram in Figure B.5. 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.

As an example of the indirect causal dependency between a message output and a message input the second diagram in Figure B.5 is considered. Amongst others, there are the following temporal orderings:

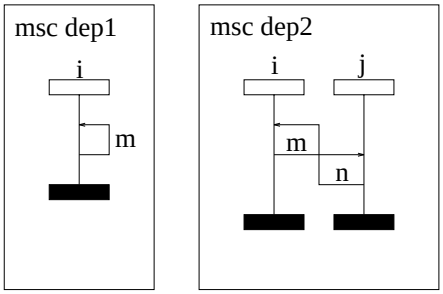


Figure B.5: Two diagrams that violate the static requirements

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.2.2.3 Textual representation

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 annex it is used for the definition of the semantics.

With respect to the textual description the language MSC offers two principal means to describe MSCs. First of all an MSC can be described by giving the behaviour of all its instances in isolation. This way of describing an MSC is called *instance-oriented* and has been incorporated in the language from the beginning. With the appearance of the main text of this Recommendation also another way of representing MSCs has been incorporated: the so-called *event-oriented* description. With the event-oriented descriptions just a list of events is given, for example as they are expected to occur in a trace of the system or as they are encountered while scanning the MSC from top-to-bottom. Besides these two ways of describing an MSC there is also the possibility to describe an MSC by mixing these two descriptions. In this annex the event-oriented textual syntax is used for the definition of a formal semantics.

The textual representation of an MSC consists of the keywords **msc** and **endmsc** and in between those an *msc name* and an *msc body*. The MSC body is defined differently for the three previously mentioned description styles.

In the event-oriented syntax an MSC body consists of a list of event definitions. An event definition is an instance name followed by an instance event. Instance events are message events and local actions.

Textually a message event is described by a message output event and a message input event. If *m* is a message that is sent from instance *i* to instance *j*, textually the corresponding message output event is denoted by "*i* **: out** *m* **to** *j*" and the message input event by "*j* **: in** *m* **from** *i*". In the graphical representation the correspondence between message outputs and message inputs is given by the arrow construction. In the textual representation a message output event and a message input event are corresponding iff

1) the events have the same message name;

2) the instance on which the message output event is specified is the same as the instance indicated by the output address of the message input event;

3) the instance on which the message input event is specified is the same as the instance indicated by the input address of the message output event.

A natural requirement on the textual representation of MSCs is that for every message output event there is at most one corresponding message input event, and vice versa, for every message input event there is at most one message output event. As no dangling message output arrows and message input arrows are allowed, another natural requirement is that for every message output (input) there is at least one corresponding message input (output). Note that for messages that are sent to the environment or that are received from the environment this requirement does not have to be satisfied.

A local action is denoted by the keyword **action** followed by an action character string.

The MSC from Figure B.1 can textually be represented by

```
msc example1;
i1  : out m0 to env;
i1  : out m1 to i2;
i2  : in m1 from i1;
i2  : out m2 to i3;
i3  : in m2 from i2;
i3  : out m3 to i4;
i4  : in m3 from i3;
i2  : action a;
i2  : out m4 to i1;
i1  : in m4 from i2;
endmsc;
```

The textual syntax of MSC is presented in Recommendation Z.120. For the definition of the semantics a simplified version of the textual syntax is used. This simplified textual syntax as well as the explanation of the simplifications can be found in Table B.5.

B.2.3 Additional Basic Concepts

In this section Basic Message Sequence Charts are extended with other basic concepts. These are process creation and termination, timer handling, incomplete messages and conditions.

B.2.3.1 Process creation and process termination

In the language Message Sequence Chart a primitive is incorporated for the dynamic creation of an instance by another instance. Such a creation is denoted by a dashed arrow, the *createline* symbol, from the creating instance to the instance head symbol of the created instance, usually as indicated in the MSC from Figure B.6. An instance can be created only once. A create event may be labeled with a parameter list, i.e., a non-empty list of *parameter names* separated by commas. In case of a process create event the parameter list is placed close to the createline symbol.

An instance can terminate by executing a process stop event. Execution of a process stop is allowed only as the last event in the description of an instance. A process stop is denoted by replacing the instance end symbol by a cross, the *stop* symbol.

In Figure B.6 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 event-oriented textual representation of the Message Sequence Chart in Figure B.6 is given in Figure B.7.

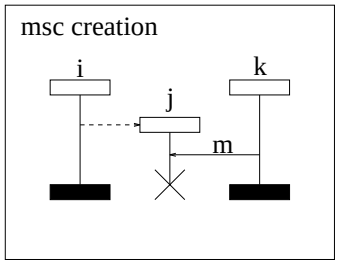


Figure B.6: Process creation and termination

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

Figure B.7: Event-oriented textual syntax

B.2.3.2 Timer handling

In a Message Sequence Chart several timer events can be described. These are the setting of a timer, a timer reset and the expiration of a timer.

In the graphical syntax the timer events can be used stand-alone but also in combinations. First the stand-alone occurrences of timer events are discussed. A timer set event is denoted by an hourglass symbol attached to the instance axis by means of a horizontal or bent line. A timer reset event is denoted by a cross which is attached to the instance axis by means of a horizontal or bent line. A timeout is represented by an hourglass symbol which is attached to the instance axis by means of an horizontal or bent arrow from the hourglass symbol to the instance axis. Examples of the stand-alone occurrences of the timer events are given in Figure B.8. A timer event is labeled by an identifier, the *timer name*. that is placed aside the hourglass symbol or cross. A timer set event may be labeled with an identifier for the duration, the *duration name*. The duration name is placed between brackets after the timer name.

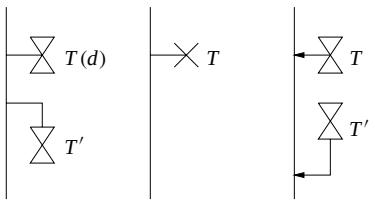


Figure B.8: Timer events in stand-alone mode

The graphical syntax of MSC also leaves room for combining timer events. The language offers the possibility to describe a timer set and a subsequent reset or timeout. Graphically these combinations are indicated by connecting the involved symbols as shown in Figure B.9. Note that for these combinations the timer name may be omitted from the reset and timeout events. 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.

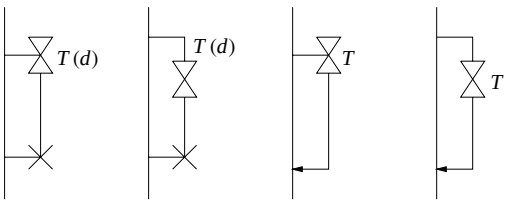


Figure B.9: Combinations of timer events (I)

Besides the combinations of the timer events given above also the following combinations are possible: a timer set symbol connected to a set-reset symbol, a timer set symbol connected to a set-reset symbol connected to a reset symbol, and a timer set symbol connected to a set-reset symbol connected to a timeout symbol. These combinations are given in Figure B.10.

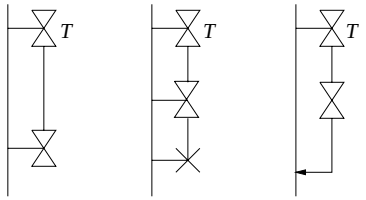


Figure B.10: Combinations of timer events (II)

The language MSC in its current form does not support the specification of a quantitative notion of time, the interpretation of the timer events is only symbolic. This means that set, reset and timeout are interpreted as events. Also, as no formal data language is available at the moment, the duration names that can be associated to a timer set event are symbolic. Any identifier can be written there.

In the textual representation 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*". A duration name can be added between brackets as follows: "**set** *T* **(***d***)**".

B.2.3.3 Incomplete message events

Besides the specification of successful transmission of messages also a lost message and a spontaneously found message can be described. A lost message is a message which is sent but will never be received by the other party in the communication. Symmetrically, a found message is a message which is received but has never been sent. A message name is associated to the lost and found messages.

Graphically a lost message is indicated by a *lost message* symbol, i.e., an arrow from an instance axis to a black dot ("black hole"). To the black dot an *input address* may be associated. This input address, which is either an instance name or the environment, represents the original destination of the message. A found message is indicated by a *found message* symbol, i.e., an arrow from an open dot ("white hole") to an instance axis. An *output address* may be associated to the open dot. This output address, which is either an instance name or the environment, is the original source of the message.

An example of the graphical representation of lost and found messages is given in Figure B.11.

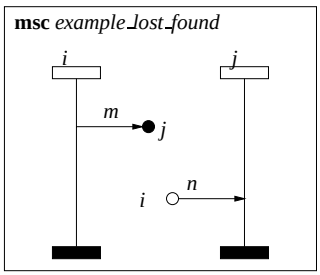


Figure B.11: An MSC with lost and found messages

```
msc example_lost_found;
i   : out m to lost j;
j   : in n from found i;
endmsc;
```

Figure B.12: Event-oriented syntax

Semantically these events are treated just as atomic events. It is not the case that a dynamic semantics is associated to messages such that they can result in lost and/or found messages. Thus these events are introduced to describe the situation where it is known that a message is lost or found.

Consider the MSC from Figure B.11. On instance *i* the sending of a message *m* with destination *j* is described. However the corresponding receive event on instance *j* is missing. Similarly, instance *j* receives a message *n* which should have been sent by instance *i*, but on instance *i* the corresponding send event is missing.

The textual representation of the incomplete messages is very similar to the textual representation of messages. The event-oriented textual representation of the MSC of Figure B.11 is given in Figure B.12.

B.2.3.4 Conditions

Graphically a condition is represented by a *condition* symbol overlapping a number of instances (at least one) and containing a list of *condition names* separated by commas. If an instance is not involved in a condition it is drawn through. In Figure B.13 an example of an MSC with a condition is given. This condition is associated to the instances *i* and *k*, but not to *j*. If a condition contains a list of condition names with more than one entry this is a convenient shorthand for an MSC with a condition symbol for each of these conditions. This shorthand can only be used if all conditions refer to the same set of instances and for all instances involved in the conditions there are no events specified in between the conditions.

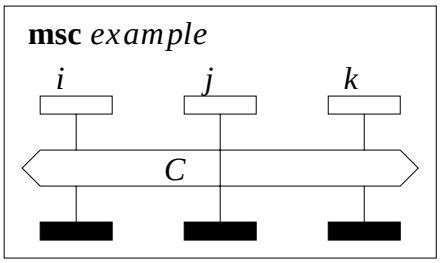


Figure B.13: Graphical representation of conditions

A condition is a first example of an event that can be associated with more than one instance. This type of events is called *multi instance events*. To facilitate the description of multi instance events without repeating them for every instance, the textual syntax is extended with the possibility to describe such an event for all instances involved. For example the condition from Figure B.13 can be described by "i**,**k **: condition** C".

B.2.4 Ordering facilities

B.2.4.1 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 part of the instance axis for which the events specified within that part are assumed to be unordered in time. Within a coregion only *orderable events* may be specified such as message events, local actions, timer events, and process creates. An example of an event that may not be used in a coregion is the stop event.

Graphically, for an instance in line-form a coregion is indicated by dashing a part of the instance axis and for instances in column-form by dashing the same parts of the two vertical lines of the instance. There is also the possibility to use a column-form coregion with a line-form instance. The other combination, a line-form coregion with a column-form instance, is not allowed. In Figure B.14, examples of these three forms are given. Examples of the placement of all orderable events on a coregion are given in Figure B.15.

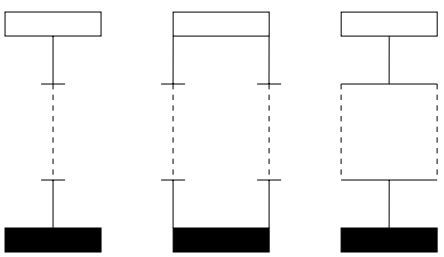


Figure B.14: Graphical representations of coregions

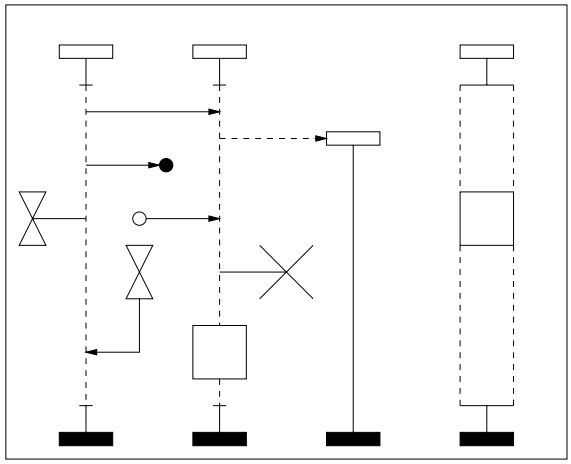


Figure B.15: Placement of events on coregions

In Figure B.16 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.

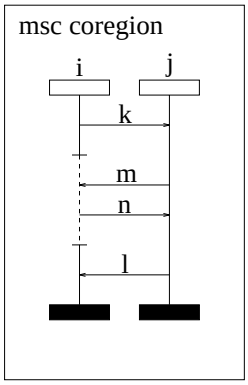


Figure B.16: Message Sequence Chart with a coregion

```
msc coregion;
i   : out k to j;
i   : concurrent;
          in m from j;
          out n to j;
      endconcurrent;
i   : in l from j;
j   : in k from i;
j   : out m to i;
j   : in n from i;
j   : out l to i;
endmsc;
```

Figure B.17: Textual syntax

In the textual notation a coregion is denoted by a list of the orderable events specified within the coregion started with the reserved keyword **concurrent** and ended by the reserved keyword **endconcurrent**. An example of the event-oriented textual syntax of coregions is given in Figure B.17.

B.2.4.2 General orderings

General orderings are introduced to facilitate the description of orderings between events when this ordering cannot be derived from the ordering of the events on an instance and the ordering by means of communication. For example if a local action *a* on instance *i* has to occur before a timeout event on instance *j*. Then the features of the language discussed so far are not sufficient. The only way to describe this with the MSC-language introduced until now is by defining a communication from *i* to *j* where the output occurs after the local action and the input occurs before the timeout event. As MSCs are mostly used for High-level requirements specifications this is undesirable. Also, if many such orderings need to be specified, the additionally introduced communication overhead is disturbing.

Graphically a general ordering of two events is represented by a solid line with an arrowhead in the middle, the *general order symbol* (see Figure B.18). This distinguishes it from normal messages where the arrow-

head is placed on one end of the line. The line may have any orientation and may also be bent. The general order symbol should be attached to the events that need to be ordered. Only orderable events can be used in general orderings.

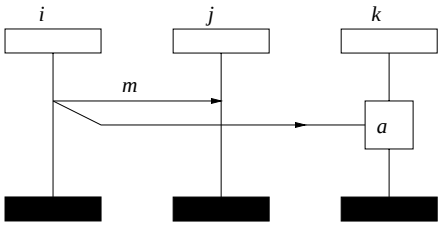


Figure B.18: Example of a general ordering

In case of a local action the general order symbol can start or end at any point of the action symbol. In case of another orderable event the start or end of a causal order symbol coincides with the point of the instance where the event symbol is attached.

The way to describe general orderings as discussed above can also be used to describe the general ordering of orderable events from the same instance. In cases where one of the events to be ordered is not inside a coregion, this either results in an inconsistent MSC or it results in an MSC for which the additional general ordering is superfluous. Examples where two local actions on one instance are causally ordered are given in Figure B.19. In the first MSC the general ordering is superfluous as the local actions are already ordered by the total ordering of events on the instance. The fact that this general ordering is superfluous does not mean that it is not allowed. The second MSC is inconsistent as the local actions are ordered in two conflicting ways.

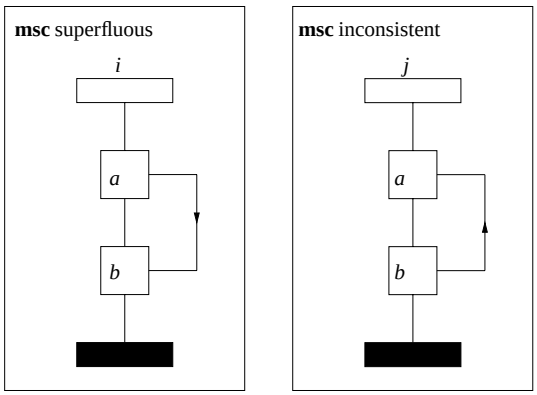


Figure B.19: Example of a general ordering within an instance

A general ordering between two events in the same coregion does give additional information. See Figure B.20 for an example. The input of message *m*, the output of message *n*, and the output of message *o* are specified in a coregion and therefore unordered. But the causal ordering between the input of *m* and the output of *o* defines that the first precedes the latter. Note that although the output of *n* and the output of *o* are specified under each other on the same line they are not ordered.

As an alternative the language MSC offers the possibility to leave the head of the arrow out. Thereby the order symbol is reduced to a single line. These lines are always interpreted from top to bottom. Also crossings of these lines have meaning. Event *a* is ordered causally before event *b* iff there is a line going from *a* to *b* that never goes up. The coregion from Figure B.22 can then also be depicted as shown in Figure B.23.

In the textual syntax general orderings are represented by using the keyword **before** followed by a list of event names. An *event name* refers to an event specified somewhere in the MSC. Thus it can be an event from the same instance or an event from another instance. An event name can be associated to an event in

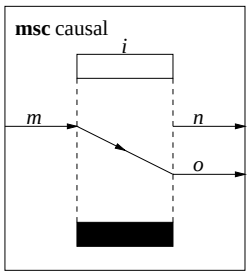


Figure B.20: General ordering within an instance

```
msc causal;
i    : concurrent;
          in m from env before l2;
          out n to env;
          l2 out o to env;
       endconcurrent;
endmsc;
```

Figure B.21: Event-oriented textual syntax

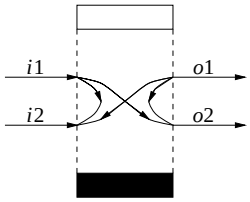


Figure B.22: General ordering within an instance

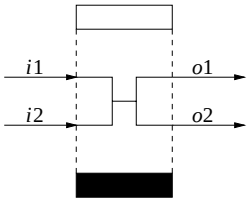


Figure B.23: General ordering within an instance

the textual syntax by placing the event name just before the event. See Figure B.21 for an example of the event-oriented textual representation of the MSC shown in Figure B.20.

An event name can be associated with only one event, i.e., there cannot be two or more events in the same MSC document with the same event name. It is not allowed that an event is ordered before itself. Textually this means that in an event definition an event name associated with an event cannot occur in the event name list following the keyword **before**.

B.2.5 Combining MSCs with composition constructs

MSC based specifications often consist of many different MSCs, instead of one single MSC. MSC offers ways to group single MSCs into *MSC documents*. An MSC document is a collection of MSCs.

MSCs can be put in a wider context by means of *composition operators*. The three primitive operators are **seq**, **par** and **alt**. In the MSC language these concepts of composing MSCs are manifest in different ways: in inline expressions, MSC reference expressions and High-level Message Sequence Charts. MSCs can also be composed using operators to express loops, exceptional behaviour and optional behaviour. For the semantics of these composition operators the notions of *vertical*, *horizontal* and *alternative* composition are used. These notions refer to the semantics of MSC and their intuition is sketched to strenghten the intuition about the primitive composition operators mentioned before.

First the intuitive semantics of the operations vertical, horizontal and alternative composition are given. Then MSC documents are treated, followed by inline expressions and MSC references. The last part of this section describes the use of the composition mechanism in High-level MSC. Wherever possible the graphical syntax for each of these language constructs is treated first, then the (event-oriented) textual syntax, then the requirements and finally the intuitive semantics, if necessary.

B.2.5.1 Vertical, horizontal and alternative composition of MSCs

In this section the intuitive semantics of vertical, horizontal and alternative composition is explained, mainly by providing examples. These examples do not form a complete and precise definition of the semantics. For a formal definition of the semantical equivalent of these operations see Section B.4.

Vertical composition

The vertical composition of two MSCs refers to the operation of placing one MSC at the bottom of another one and then linking the instances they have in common thus obtaining a new MSC.

If the MSCs have *no* instances in common the meaning of the vertical composition is the same as an MSC with the instances of these MSCs placed next to each other. See Figure B.24 for an example. MSC *first* has instances named *i* and *j* and MSC *second* has instances named *k* and *l*. The MSCs have no instances in common, so there are no links to be made. Thus vertical composition of MSCs does not necessarily mean that all events from the first MSC (in the example MSC *first*) have to be executed before any event from the second MSC (MSC *second*) can be executed. In the example this means that the sending of *n* might as well occur before the sending of *m*.

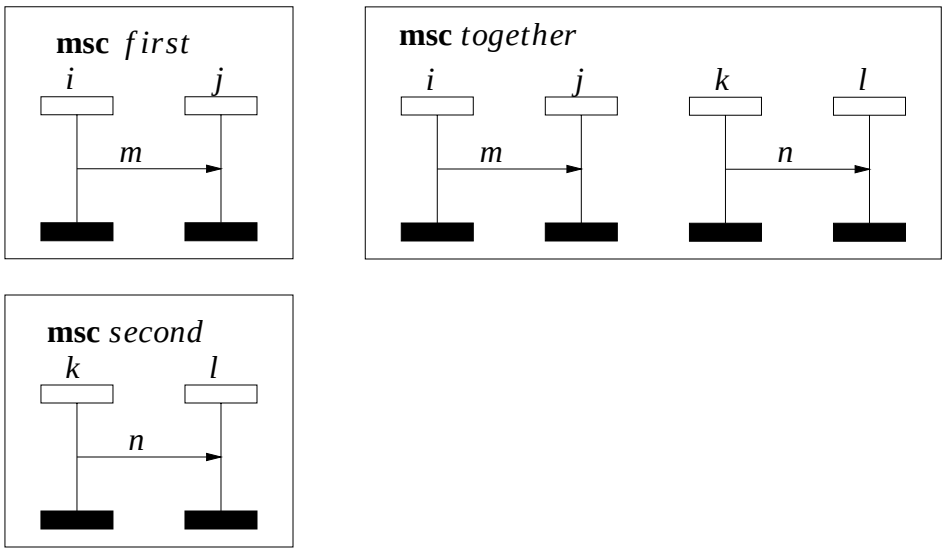


Figure B.24: Vertical composition with disjoint instances

Another case occurs if the MSCs have *all* instances in common. Then all events from an instance of the second MSC have to occur after the events from the same instance of the first MSC. For an example see Figure B.25. The MSCs *first* and *second* have the instances *i*, *j*, and *k* in common. The reception of message *m* by instance *j* necessarily has to precede the reception of message *n* by instance *j* in the resulting MSC *together*. In this example it is still possible that the sending of message *n* by instance *i* which is an event described in MSC *second* is executed before the reception of message *m* by instance *j* in MSC *first*.

Also the situation in which the MSCs have instances *in common* and also have *different* instances is allowed. For example the MSCs *A* and *B* from Figure B.26 have the instance *j* in common, but instance *i* is only described for MSC *A* and instance *k* is only described for MSC *B*. The result of the vertical composition of the MSCs *A* and *B* is given as MSC *AB* in the same figure.

Horizontal composition

The horizontal composition of two MSCs refers to the operation of placing them next to each other. If the MSCs have some or all instances in common, it is assumed that the behaviour of the common instance(s) is the interleaving of the behaviours of these instance(s) in the separate MSCs.

In the case that the MSCs have no instances in common, the horizontal composition is similar to the vertical composition (see Figure B.24). For an example of the case where the MSCs have at least one instance in common, we refer to Figure B.27.

In this example the MSCs *first* and *second* have the instance *j* in common. As stated before, the behaviour of the shared instance is obtained by interleaving the events of the separate instance descriptions. This can be expressed in a coregion with general orderings as shown in MSC *together*.

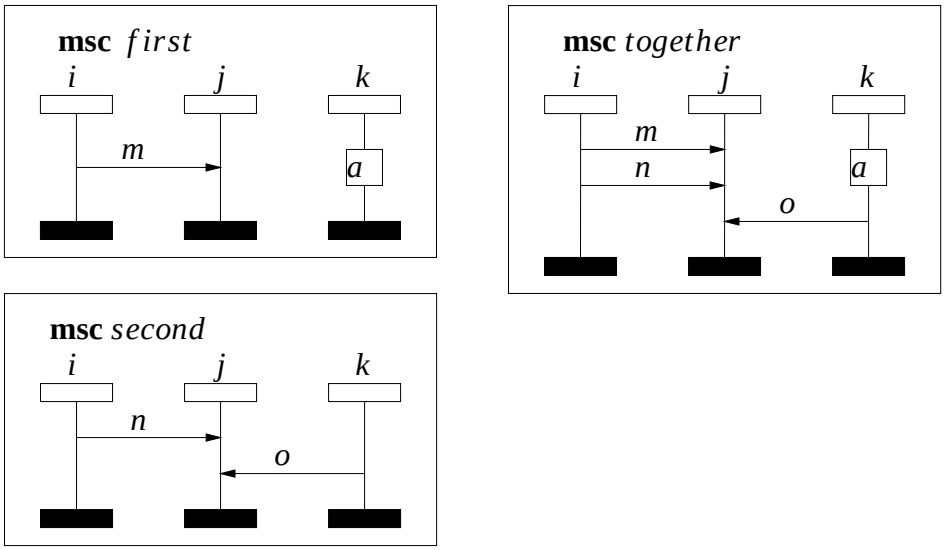


Figure B.25: Vertical composition with a common instance

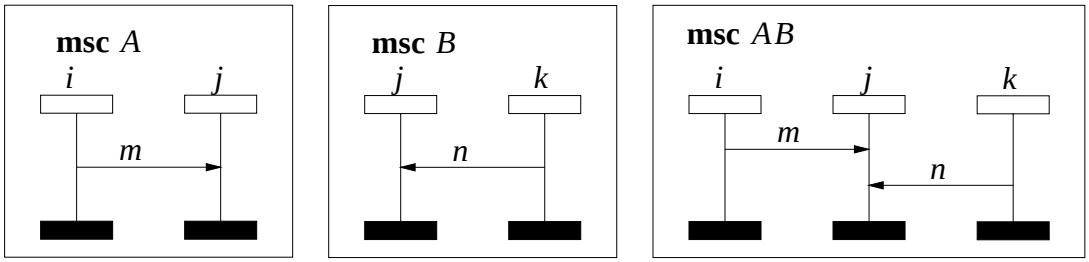


Figure B.26: Vertical composition

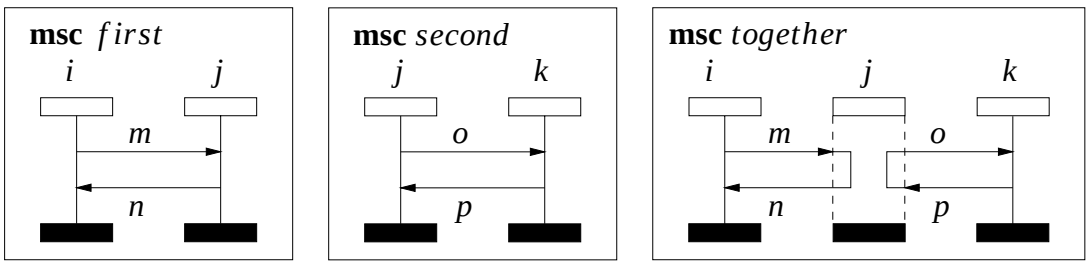


Figure B.27: Horizontal composition with shared instance

Alternative composition

Usually a system is not described by means of one single MSC, instead a number of MSCs is used to describe several alternative scenarios. With the features of MSC introduced so far, alternative scenarios can only be described for one MSC. So each trace contains precisely the same events. For example, it is impossible to describe that either an event *a* or an event *b* is executed. A reasonable means to describe alternatives is by giving one MSC for each of the alternatives. Thus large piles of scenarios come into existence, for example when describing system requirements or when describing a system by giving different Use Cases.

In complex systems there are many points of deviating behaviour. Therefore, it is important to be able to indicate at what point alternatives occur. For that reason the language MSC offers an operator to describe alternatives in an MSC. An important aspect of the meaning of the alternative composition mechanism in MSC is that the moment of choice between the different scenarios is postponed until that choice can no longer be avoided.

Consider the MSCs *A* and *B* as given in Figure B.28. Each of these MSCs has one initial event, the sending of *m* and the sending of *n* respectively. The alternative composition of these MSCs now has two initial events: the sending of message *m* and the sending of message *n*. If the sending of message *m* is executed a choice is made for the execution of MSC *A*. On the other hand, if the sending of message *n* is executed, a choice is made in favour of MSC *B*. Thus, with the execution of an event which can be executed by only one of the alternatives, all alternatives that cannot execute this event are discarded.

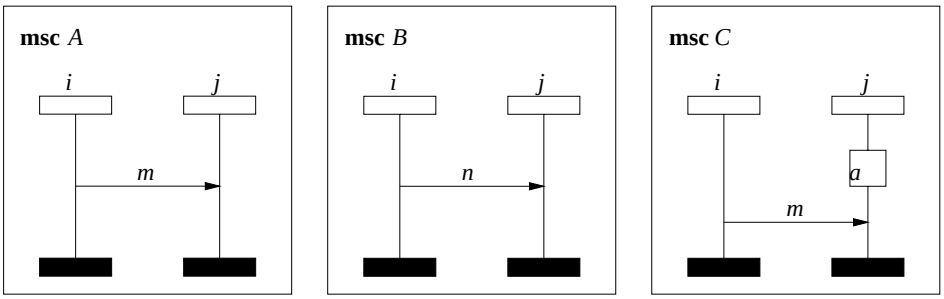


Figure B.28: MSCs

Now consider the MSCs *A* and *C* from Figure B.28. If the local action on instance *j* of MSC *C* is executed necessarily a choice is made in favour of MSC *C*. But, if the sending of message *m* occurs, this event can originate from either MSC *A* or *C*, though it is not clear from which of the two MSCs it originates at the moment of execution of this event. Alternative composition in MSCs is defined in such a way that no choice is made until this cannot be avoided. One could say that after the execution of the sending of message *m* there still are two alternatives: the parts of the MSCs *A* and *C* that remain to be executed. In this specific example now a choice has to be made as the MSCs have no initial events in common anymore.

B.2.5.2 MSC documents

In the following sections the means offered by MSC to compose MSCs are considered. As a consequence it must be possible to describe more than one MSC. For this purpose Message Sequence Chart documents are used.

Graphically an MSC document is given as a frame symbol with a *document head* in it.

Textually, an *MSC document* consists of an *MSC document head* and an *MSC document body*. The MSC document head consists of the keyword **mscdocument** followed by an *MSC document name*. The MSC document body consists of a number of Message Sequence Charts.

For MSC documents the following static requirements are formulated. Within an MSC document there must not be two or more MSCs with the same name. Within the MSCs of an MSC document only references to MSCs specified within that MSC document may be used. An MSC may not be depending on itself, directly

or through a number of references.

B.2.5.3 Inline expressions

Inline expressions provide a means to formulate the composition of MSCs within the MSC language. The operators that can be used are the ones discussed before except for vertical composition.

Graphically an inline expression consists of an *inline expression* symbol that is attached to a number of instances (at least one). This inline expression symbol contains in the left-upper corner one of the keywords **alt**, **par** or **loop**. These keywords indicate the composition operation that is described by the inline expression. The keyword **alt** refers to an alternative composition, the keyword **par** refers to a horizontal composition. The keyword **loop** indicates iteration of the events within the inline expression.

Both alternative and horizontal composition can have any finite, positive number of operands. These operands are all drawn inside the inline expression symbol and they are separated by a dashed vertical line, the *separator* symbol.

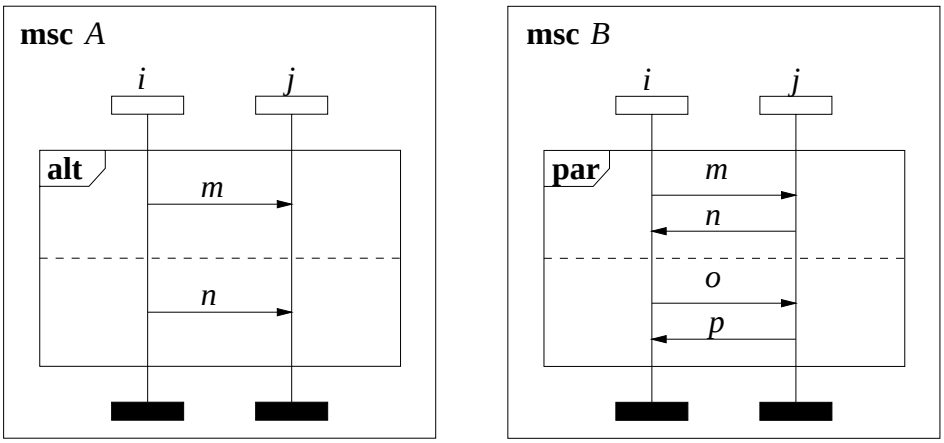


Figure B.29: Examples of inline expressions

Some examples of inline expressions are given in Figure B.29. In MSC *A* an inline expression is attached to the instances *i* and *j*. This inline expression has the keyword **alt** in its upper left corner in order to indicate that the parts of the MSCs that are separated by means of the separator symbol are considered *alternatives*. In this particular example there are two operands. The first describes the sending of a message *m* by instance *i* and its subsequent reception by instance *j*. The second operand describes the sending of a message *n* by instance *i* and its subsequent reception by instance *j*. The meaning of this MSC in terms of sequences of events that can be performed is that either the sending and reception of *m* or the sending and reception of *n* takes place but not both. As soon as the sending of one of the messages takes place it is known which operand is executed.

In MSC *B* the *horizontal composition* of two "MSCs" is indicated by means of the keyword **par**. In this case all events are executed in such a way that the orderings described by the first operand are respected and at the same time the orderings described by the second operand are respected. MSC *B′* from Figure B.30 has the same behaviour as MSC *B*.

An inline loop expression has exactly one operand. This operand is described by means of the part of the MSC that is drawn inside the inline expression symbol. An example is given in Figure B.31. The inline loop expression in MSC *B* describes that the sending and receiving of message *m* occurs zero, one or two times, followed by the sending and receiving of message *p*. Intuitively the behaviour of MSC *B* is the same as the behaviour of MSC *D* from the same figure.

The keyword **loop** is followed by a *loop boundary*. This loop boundary refers to the number of repeated vertical compositions. The loop boundary, if present, indicates the minimal and maximal number of verti-

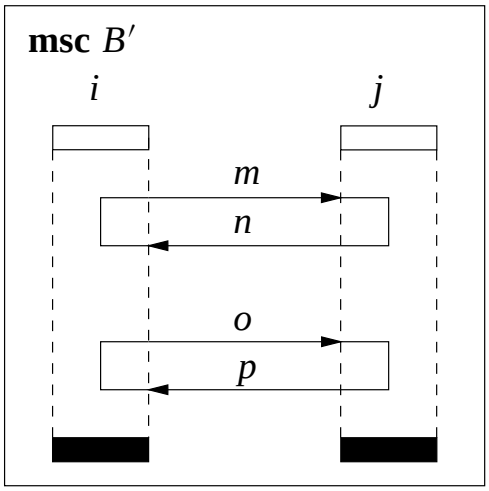


Figure B.30: MSC equivalent to MSC *B*

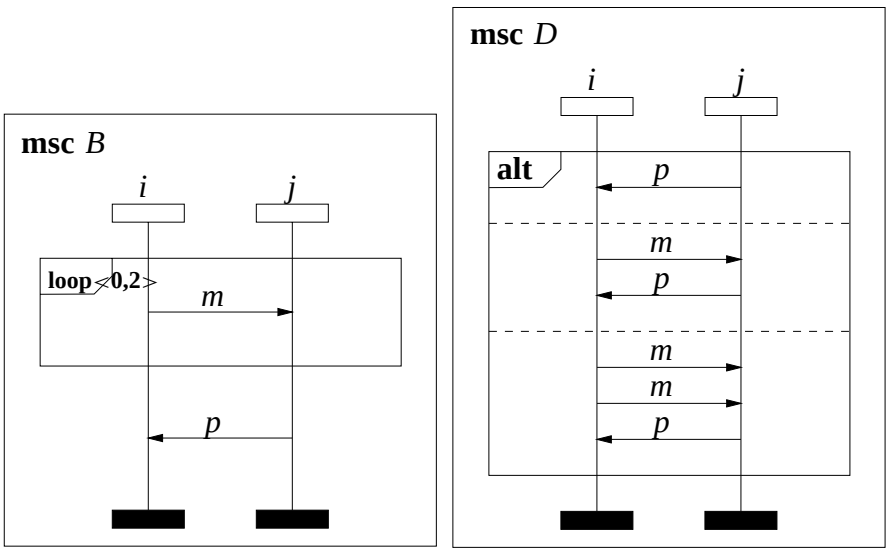


Figure B.31: Examples of inline expressions

cal compositions of the operand. Such a number can either be the keyword **inf**, representing infinity, or a sequence of *natural name*s. A natural name can be any label. For the semantics it is important to be able to interpret the sequences of natural names as natural numbers. For the semantics definition, it is assumed that a natural name can only be the keyword **inf** or a sequence of digits. An interpretation of loop boundaries in $IN \cup \{\infty\}$ is assumed.

The loop boundary is be of the form <n,m> where *n* and *m* are natural numbers or **inf**. The combination **loop** <n,m> means that the operand of the operator is executed at least *n* and at most *m* times. If the interpretation of natural number *n* is greater than the interpretation of natural number *m* (with the standard interpretation) then this means that the operand is executed zero times.

If an instance is not involved in the operands of an inline expression, then it is possible to hide the part of the instance axis of such an instance behind the inline expression. See Figure B.32.

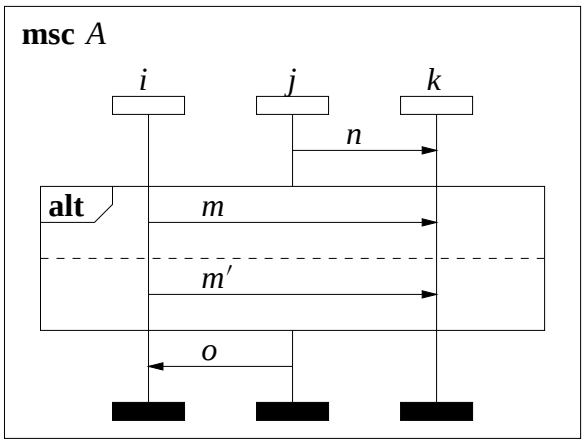


Figure B.32: Examples of inline expressions

B.2.5.4 MSC reference expressions

An MSC reference expression can be used to refer to other MSCs in an MSC document by means of their MSC name. Graphically an MSC reference expression is represented by a textual formula in a rounded frame, the *msc reference* symbol, which is placed on top of a number of instances. This textual formula is an expression containing references to other MSCs in the MSC document via their MSC name. Operators for composing MSCs are: **alt**, **seq**, **par**, **empty**, **loop** operators and parentheses for grouping subexpressions.

An MSC reference expression is indicated in the textual syntax by the keyword **reference** followed by the textual formula.

The event-oriented textual syntax of MSC *D* of Figure B.33 is given by:

msc D;
i,j,k **reference** A;
i,j,k **reference** B;
endmsc;

The binding power of the operators is in descending order as follows: **loop**, **opt**, **exc**, **seq**, **par**, **alt**. The binding powers can be superseded by using parentheses. Examples of MSC reference expressions are:

A (*A* **alt** *B*) **seq** *C* **loop** < 5,16 > (*A* **par** *B*) *A* **seq loop** < 3,**inf** > *B*

There are two requirements that must be satisfied with respect to the instances that are overlapped:

1) If an instance that is present in the enclosing MSC diagram is also present in the MSC reference expression, then the MSC reference symbol must overlap this instance. An instance is present in an MSC reference expression if at least one of the MSCs that are referenced in the expression has an instance with that name.

2) If two MSC reference expressions in the same enclosing MSC diagram share an instance then this instance must be drawn in the enclosing MSC diagram.

Note that these requirements do not say that every instance that is present in the MSC reference expression must be visible in the enclosing MSC. The requirements also do not say that an MSC reference expression may not overlap an instance that is not present in the MSC reference expression.

The reason for these requirements is that otherwise it is possible to draw an MSC with an MSC reference expression such that it is not clear how the events on an instance are ordered. MSCs *D* and *E* from Figure B.33 show two correct ways of combining MSCs *A* and *B* with MSC reference expressions. Because of the requirements it is obligatory that both MSC reference symbols overlap instance *j*.

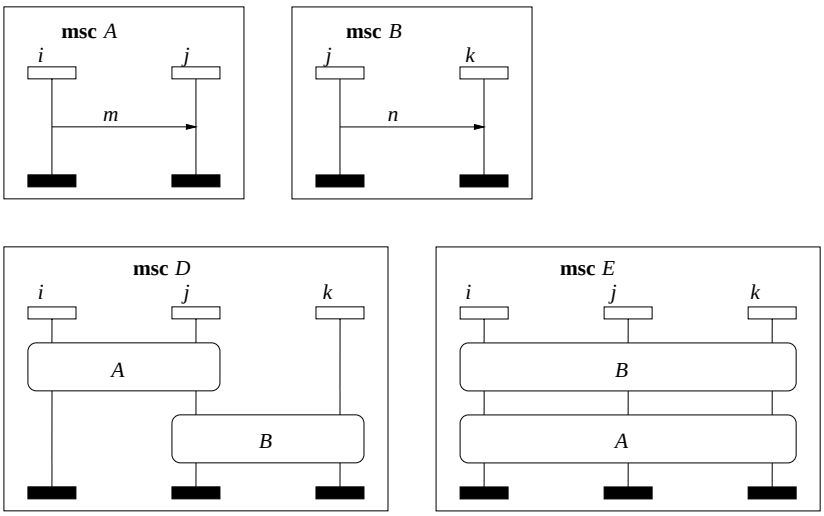


Figure B.33: An example

B.2.5.5 High-level Message Sequence Charts

High-level Message Sequence Charts (HMSC) provide an attractive graphical way to combine Message Sequence Charts using the operators from the former sections. An HMSC is a directed graph, where the nodes are formed by other (H)MSCs and the vertices (arrows) imply an order on the nodes. There are also other elements that make up an HMSC:

1) start node: ▽
2) end node: △
3) msc reference node: ▭
4) condition node: ⬡
5) connection node: ○
6) parallel frame: ▭

These elements are all used in MSC HMSC_Example, in Figure B.34. This MSC depicts the horizontal composition of MSC3 with the left part of the parallel frame. The left part of the parallel frame is the vertical composition of MSC1 with the alternative composition of MSCs MSC2A and MSC2B.

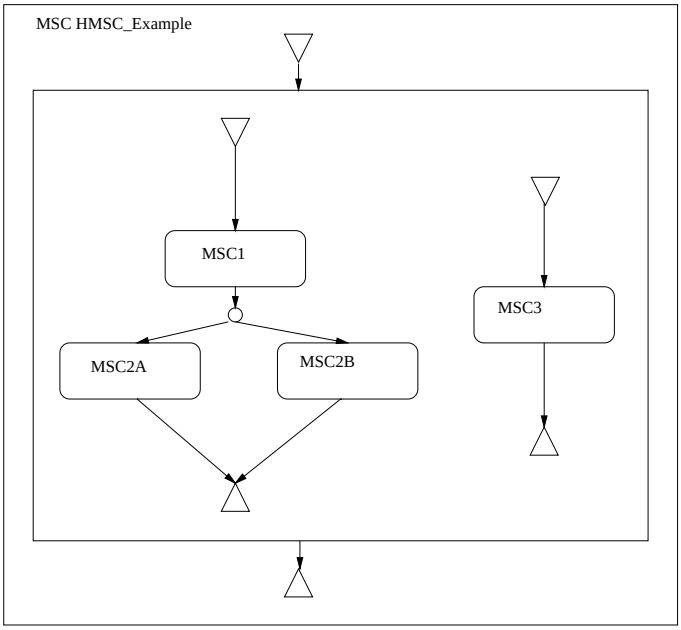


Figure B.34: An example of an High-level Message Sequence Chart

An HMSC is described textually by associating a label to every node of the HMSC except the start symbol. The start symbol is implicitly named since there is only one start symbol for every HMSC. The connections between the start symbol and the other nodes are described first. For example if the start symbol has successor nodes labeled with $l_1, \ldots, l_4$, this is described by "**expr** l_1 **alt** l_2 **alt** l_3 **alt** l_4 **;**". It indicates that there is an arrow from the start symbol to the nodes labeled with $l_1, \ldots, l_n$. Then for every node of the HMSC in isolation its type/contents is described possibly followed by a list of its successor nodes in a "node expression". For example if the HMSC contains a node labeled l with successor nodes labeled l' and l'' and this node labeled l is a reference to an MSC named A, then this is described as follows: "l **:** A **seq (** l' **alt** l'' **) ;**".

There are a number of requirements on HMSCs. First of all, every HMSC must have one start node. Every node must be reachable from this start node. An arrow head is always connected to the upper segment of a node symbol, and the open end of an arrow is always connected to the lower segment. Each node, except the end node, has a successor, i.e. has an outgoing arrow to another (possibly the same) node.

The semantics of an HMSC is relatively easy to explain at this point, since no new operators are introduced. The semantics of MSC reference nodes in HMSC is the same as that for MSC reference nodes for MSC treated in Section B.2.5.4. If two MSC reference nodes are connected via exactly one arrow, they are vertically composed. If an MSC reference node has more than one outgoing arrow, then all the successors of that node are alternatives for the vertical composition with that MSC reference node. Horizontal composition of MSC reference nodes can be achieved with a parallel frame, i.e. a bounding box within the bounding box of the MSC in question. A parallel frame can contain more than one start node. Each start node indicates a *sub MSC*, i.e. an operand for the horizontal composition operator. Within a parallel frame all nodes but the end node must have a successor as well. The meaning of condition nodes is not defined in this semantics. The meaning of connector nodes is void, they disambiguate *crossing* lines in a HMSC document from *splitting* lines.

The arrows in an HMSC can form cycles. This indicates repetition. Figure B.35 shows such a HMSC that contains a cycle. This MSC is equivalent to MSC Loop shown in the same figure. In general, it is not pos-

sible to translate a cycle in an MSC with the **loop**⟨**inf**⟩ operator. If e.g. the HMSC would have an endnode connected to the MSC reference node, the cycle would have to be interpreted with the operator **loop**⟨**1,inf**⟩, The precise explanation of the semantics of cycles in HMSCs is given in Section B.4.10.3.

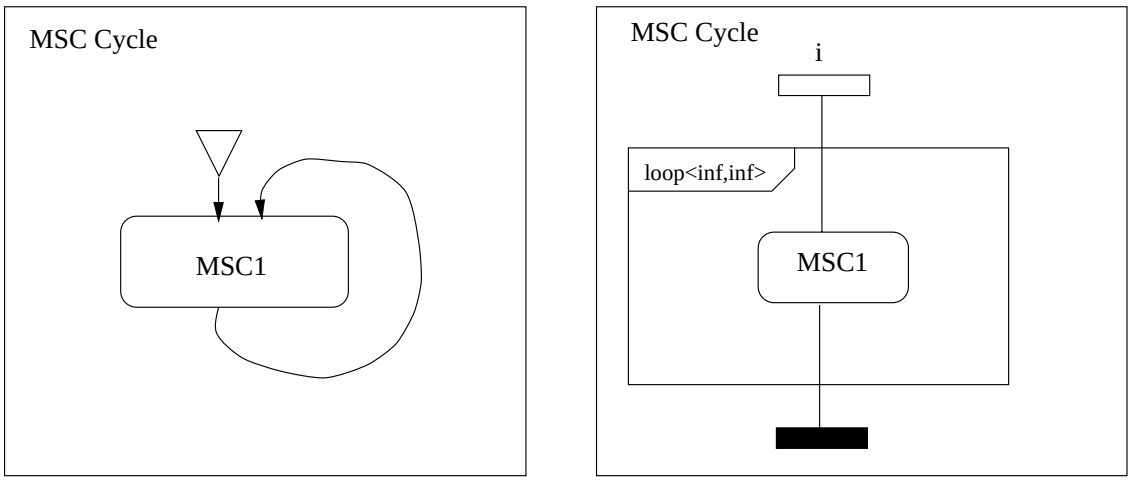


Figure B.35: An example HMSC with a cycle

B.3 Message Sequence Charts with Gates

B.3.1 Gates

B.3.1.1 Motivation

When describing industrial systems with the use of Message Sequence Charts as presented so far one of the biggest problems is the number of instances and the number of events on these instances. The diagrams easily get to big to handle, print, read, etc. In order to solve this problem complex MSCs must be decomposed into smaller MSCs. In general it is impossible to do this by means of horizontal or vertical composition without ever having to cut a message or causal ordering in two parts, where one part is located in the one component and another part is located in another component. An example of such an MSC is given in Figure B.36. To facilitate this gates are introduced.

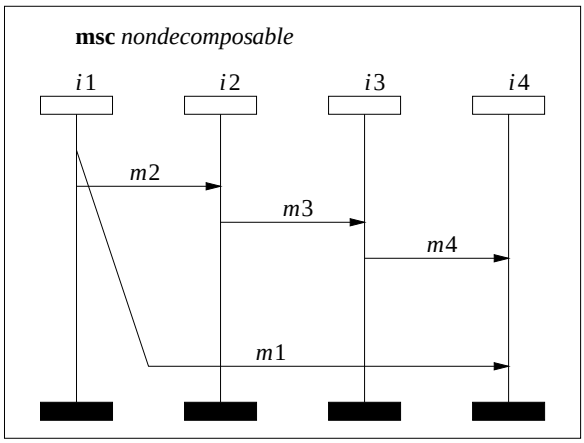


Figure B.36: An MSC that cannot be decomposed.

Gates are implicitly or explicitly named parts of the environment. As such they can be used to describe the interface between an MSC and its environment. Any message arrow or causal order arrow attached to the frame surrounding an MSC defines a gate. In the recommendation there are two types of gates: *message gates* and *order gates*. Message gates are used for message events and order gates are used for causal orderings.

B.3.1.2 Graphical representation of gates

Graphically an explicitly named gate is indicated by associating a gate name with the place where a message arrow or causal order arrow is attached to the frame of the MSC, i.e., the environment. A message gate always has a name, either explicitly given or implicitly defined. By associating a name with the gate on the frame of the MSC the *gate name* is explicitly defined. In this annex it is assumed that all gates have explicitly given names. Examples of explicitly named message gates are the message gates $g1$ and $g2$ in Figure B.37. Graphically it is only possible to distinguish the two types of gates, message gates and order gates, by means of the type of arrow associated to it. If this is a message arrow the gate is a message gate; if it is a causal order arrow, the gate is an order gate.

In principle order gates are treated similarly as message gates. Order gates always have to be named explicitly. See Figure B.37 for an example of an MSC with two order gates $g3$ and $g4$. The gate indicated by the name $g1$ is called an *order in gate* and the gate with gate name $g2$ is an *order out gate*.

So far we have only considered gates as the input address of message output events and as the output address of message input events. The recommendation also allows a gate to be used as the input address of a lost message output event or as the output address of a found message input event.

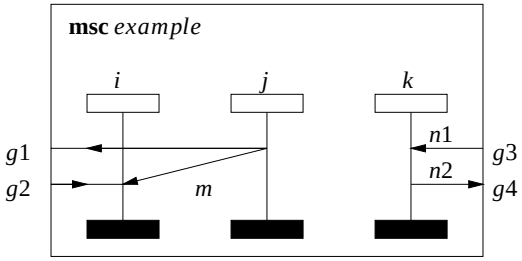


Figure B.37: MSC to illustrate message and order gates.

The intended use of gates is, as already explained in the motivation, for composing and decomposing large specifications and descriptions into more tractable pieces. This will become more apparent in the sections on MSC reference expressions and gates and inline expressions and gates. For now we will only discuss the aspects related to gates on the MSC-level without these composition mechanisms.

B.3.1.3 Semantics of gates

For the semantics of gates we refer to the upcoming two sections where gates are connected on MSC reference expressions and inline expressions.

B.3.1.4 Textual representation of gates

Textually, the name of a message gate can be used as an output or input destination for message output and message input events. An explicitly named gate is textually represented by the keywords **env** and **via** followed by a gate name. For example, the sending of message *n*2 from instance *j* to the gate *g*2 is denoted by "*j* **: out** *n*2 **to env via** *g*2". Textually the MSC from Figure B.37 can be represented by

msc example **;**
i **:** l1 **in** m **from** j **after env via** g1 **;**
j **:** l2 **out** m **to** i **before env via** g2 **;**
k **: in** n1 **from env via** g3 **;**
k **: out** n2 **to env via** g4 **;**
endmsc ;

Textually, a causal order arrow from an event on an instance *i* to a gate *g* is described by "i **:** e **before env via** g". Thus, the keywords **env** and **via** followed by a gate name can be used to describe the destination of the causal order arrow. However, a causal order arrow from a gate *g* on the frame of an MSC to an event *e* on an instance *i* cannot be described in a similar way. For this purpose the MSC gate interface necessarily contains a defining occurrence of a gate and an ordering. There is a simple, elegant solution however. If the textual syntax of MSC is extended with a keyword **after** which can be used on all places where **before** is allowed, then the gate *g*1 can be described in the MSC body by "i **: in** m **from** j **after env via** g1". In Section B.5 the extension of the textual syntax with a keyword **after** is discussed.

For lost message output events and found input events the input address and output address respectively can also be an explicitly named gate. The textual syntax for these output and input addresses is identical to the syntax for message output and input events.

B.3.2 MSC reference expressions and gates

B.3.2.1 Graphical representation

In the previous section we have seen how gate definitions can be described both graphically and textually. In this section we will extend the syntax for MSC reference expressions with gates. An MSC reference ex-

pression is indicated graphically by a textual formula in an MSC reference symbol. As the MSCs referenced in the textual formula can have gates, it should be possible to connect gates from referenced MSCs. For this purpose actual gates are used. An actual gate is defined by connecting a message arrow with the MSc reference expression symbol. By placing a gate name close to the point of connection an explicitly named actual gate is defined. If the gate name is omitted an implicitly named actual gate is defined. In Figure B.38 the different occurrences of gates are named.

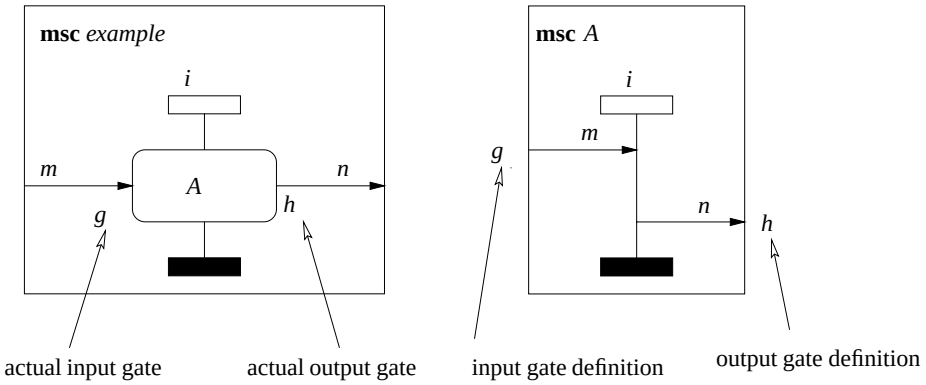


Figure B.38: Terminology on gates.

The actual gates of an MSC reference expression may connect to corresponding constructs in the enclosing MSC. An actual message gate (on an MSC reference symbol) may connect to another actual message gate, an instance, or a message gate definition (implicitly or explicitly named) of the enclosing MSC by means of a message arrow. Similarly, an actual order gate may connect to another actual order gate, an orderable event, or an order gate definition of the enclosing MSC by means of a causal order arrow.

A message arrow can only be connected to an MSC reference symbol if at least one of the MSCs that are referenced has a corresponding gate. If a message *m* is sent to an actual input gate *g* of an MSC reference expression, then the MSC reference expression must contain a reference to an MSC with an input gate definition of gate *g* for a message *m*. If a message *m* is received from an actual output gate *g* of an MSC reference expression, then the MSC reference expression must contain a reference to an MSC with an output gate definition of gate *g* for a message *m*. Examples of the graphical appearance of such connections are given in Figure B.39.

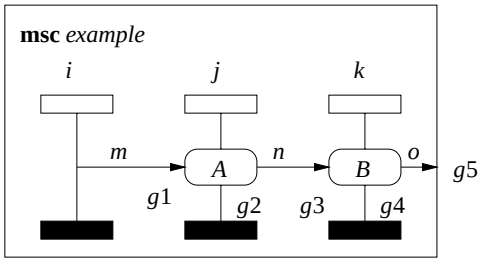


Figure B.39: Gates on MSC reference expressions.

It is important to define what the gates are of an MSC reference expression as the above explanation refers to this notion. The set of gates of an MSC reference expression is the union of the sets of gates of the MSCs referenced by that expression.

It is allowed to connect two message gates from the same MSC reference expression in an enclosing MSC. An example of this situation is the MSC given in Figure B.40. It is also possible to connect gates from different MSCs that are referenced in the same MSC reference expression.

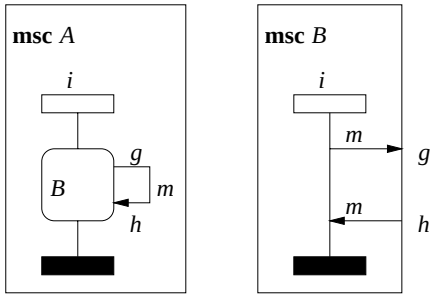


Figure B.40: Connecting gates from the same MSC reference expression.

B.3.2.2 Semantics of MSC reference expressions

MSC reference expressions with gates that are connected on the outside of the MSC reference symbol describe how a message or causal order arrow is continued outside the MSC reference symbol. For the MSC in Figure B.41 a message arrow is drawn from instance *i* to the MSC reference expression. This means that a message *m* is sent by instance *i* to the receiver of the corresponding message input event in MSC *A*. In this case this is instance *j*.

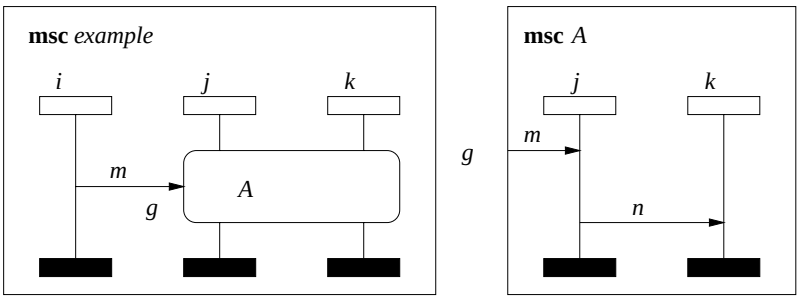


Figure B.41: Connecting a gate.

It is also possible to connect the gates of two MSC reference expressions by means of a message arrow (see Figure B.39). If a gate of an MSC reference *A* is connected to a gate of an MSC reference expression *B* by means of a message arrow with message identifier *m* this means that the output of message *m* inside MSC *A* is connected to the input of message *m* inside MSC *B*. Note that by the requirements these have to exist and have to be unique.

A third possibility is to connect a gate from an MSC reference expression with a gate of the MSC. This means that the message output or input event is sent to or received from the environment of the enclosing MSC. Also, if a gate of an MSC reference expression is not connected this implicitly means that it is connected to the environment of the enclosing MSC. Examples of both situations are given in Figure B.42. From a semantics point of view the two MSCs *A* are equivalent.

So far we have only indicated what the meaning is of connecting gates in the case that the MSC reference expression is only a reference to an MSC by means of its name. However, MSC reference expressions can easily become more complex.

For example the MSC reference expression can be the alternative composition of two MSC reference expressions by means of the keyword **alt**. It can be the case that the one MSC reference expression has a gate *g* and the other has no such gate. An example of this situation is given in Figure B.43.

In case that the MSC *A* is selected for execution, the MSC can only perform the sending of message *m* and its subsequent reception. On the other hand, if MSC *B* is selected, we expect the execution of local action *a* and the output of message *m* in an arbitrary order. Note that in this case, the input of message *m* does not take place.

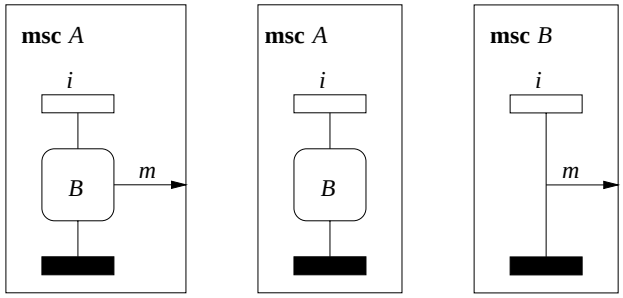


Figure B.42: Propagation of a gate to the environment.

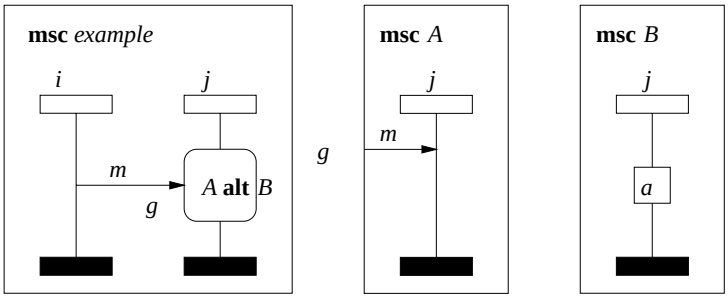


Figure B.43: Connecting a gate.

Thus it is possible that a message is sent by an instance to an instance while the receiver instance never receives the message. For message inputs however it is impossible that this situation arises.

In Figure B.44 an MSC *example* is given that refers to an MSC *A* by means of the MSC reference expression **loop** *A*. Instance *j* receives a message from the gate *g*. In MSC *A* a message is sent to a gate *g*. As a consequence MSC *example* expresses that message *m* is sent an arbitrary number of times, but at least once, to instance *j* and that instance *j* receives message *m* exactly once. The MSC does not specify which occurrence of the sending of message *m* is received. The other occurrences of the sending of message *m* are never received.

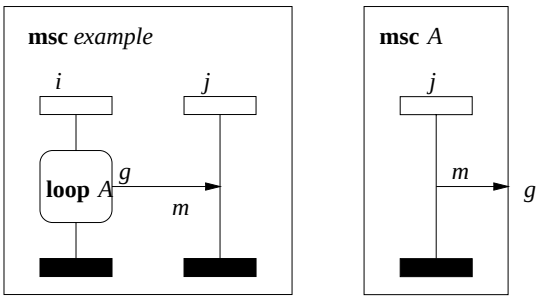


Figure B.44: Gates and loops.

B.3.2.3 Textual representation of MSC reference expressions with gates

It extends the description of MSC reference expressions without gates with an *MSC reference identification* and with a *reference gate interface*. The MSC reference identification is used to unambiguously identify an MSC reference expression. If a gate on an MSC reference symbol acts as output or input address of a message arrow or as the destination of an causal order arrow, this is described textually by the keyword

reference followed by an MSC reference identification and by the keyword **via** and the gate name. The defining occurrence of the MSC reference identification therefore has to be unique.

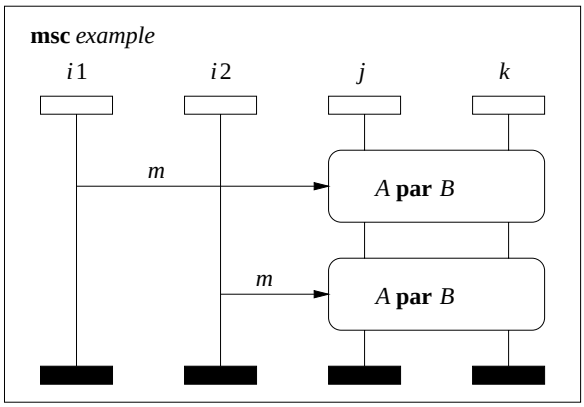


Figure B.45: MSC where MSC reference identifications are needed in the textual description.

Consider for example the MSC from Figure B.45. Graphically it is immediately clear that the output of *m* by instance *i*1 is to the first occurrence of the expression *A* **par** *B* and the output of *m* by instance *i*2 is to the second occurrence of the expression *A* **par** *B*. Textually we need a means to distinguish these two references which have the same appearance. Thereto the MSC reference identification is used. In this example we use *parallel*1 and *parallel*2 as MSC reference identifications for the first and second occurrence of the expression *A* **par** *B* respectively. Textually this MSC is described as follows:

msc C **;**
i1 **: out** m **to reference** parallel1 **via** g **;**
i2 **: out** m **to reference** parallel2 **via** g **;**
j **,** k**: reference** parallel1 **:** A **par** B **;**
j **,** k**: reference** parallel2 **:** A **par** B **;**
endmsc ;

With every MSC reference expression a reference gate interface can be associated. This interface describes how the gates of the MSCs that are referenced in the MSC reference expression are connected in the enclosing diagram. If a gate of the MSC reference expression is not connected in the enclosing MSC no entry in the reference gate interface is required. Syntactically the entries in this interface are described similar to the descriptions of the gates in the MSC gate interface.

B.3.3 Inline expressions and gates

B.3.3.1 Graphical representation of inline expressions with gates

Graphically an inline expression is indicated by an inline expression symbol or an exc inline expression symbol. Inside the inline expression symbol the operands are described in the form of an anonymous MSC (i.e., an MSC without an MSC name) without instance head and end symbols. A message arrow or causal order arrow that is attached to the inline expression symbol constitutes a gate definition. At the same time a continuation of this arrow in the enclosing MSC describes a connection of this gate. Thus, for inline expressions the definition of a gate (of the anonymous MSC) coincides with the use of the gate (the actual gate). As was the case for gates on MSC reference symbols the actual gates can be explicitly or implicitly named. Again, it is assumed in this annex that all gates are explicitly named. In Figure B.46 the gate definitions and actual gates are indicated.

A message gate on the inline expression symbol can be connected by means of a message arrow. Similarly, an order gate on an inline expression symbol can be connected by means of a causal order symbol.

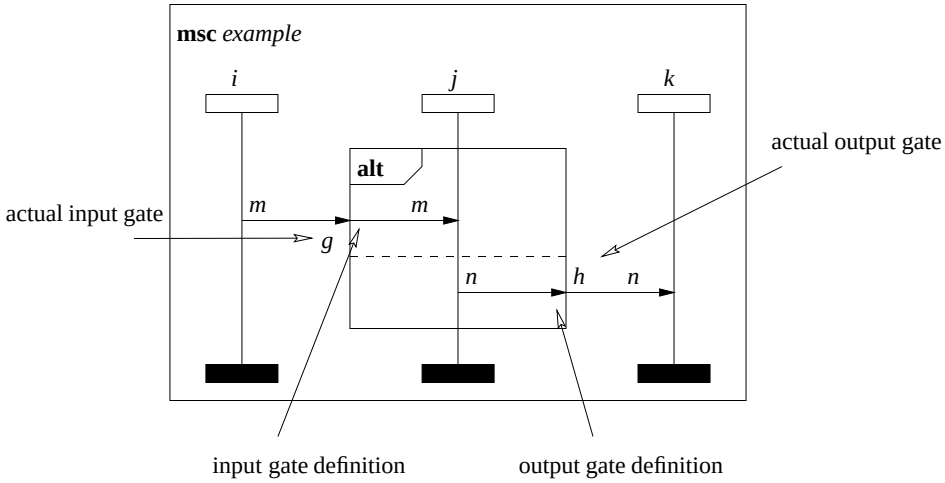


Figure B.46: Terminology of gates on inline expressions.

B.3.3.2 Semantics of inline expressions with gates

If a message arrow or causal order arrow is connected to the inline expression symbol internally, but not externally this indicates that the gate propagates to the frame of the enclosing MSC. The gate name remains the same.

For all occurrences of a gate on an inline expression the internal address of the different occurrences of this gate must be identical. The reason for this requirement is that in the textual syntax there is no means to distinguish the different occurrences of the gate.

B.3.3.3 Textual representation of inline expressions with gates

The introduction of an *inline expression identification* in the textual representation of inline expressions with gates is motivated similarly as the introduction of the MSC reference identification in the previous section.

If a gate of an inline expression is the output or input address of a message arrow this is described by means of the keyword **inline** followed by the inline expression identifier and by the keyword **via** and the gate name.

For each operand of the inline expression an *inline gate interface* can be described. Such an inline gate interface describes both the internal and external connections of the gates on the inline expression symbol.

B.4 Process theory for Message Sequence Charts

B.4.1 Introduction

In this section a number of constructs are defined operationally by means of deduction rules. These constructs are

- special constants δ and ε;
- atomic actions $a \in A$;
- delayed choice operator $\mp$;
- delayed parallel composition $\|$;
- weak sequential composition $\circ$;
- bounded repetition *;
- unbounded repetition $^\infty$;
- generalized parallel composition $\|^S$ and generalized weak sequential composition $\circ^S$;
- renaming operator ρ_f;
- recursion variables.

Together these constants and operators form the signature Σ. The deduction rules defining the operational behaviour of these operators is explained and illustrated by means of examples. With these constants and operators expressions can be built. The operators have the following binding powers in decreasing order:

1) the unary operators *, $^\infty$ and ρ_f;
2) the binary operators $\|$, $\|^S$, $\circ$ and $\circ^S$;
3) the binary operator $\mp$.

For operators with equal binding power brackets are associated from the left. As a consequence of these binding rules the expression $a \circ b \mp c$ should be read as $(a \circ b) \mp c$. Another example is the expression $a \circ b \| c$ which should be read as $(a \circ b) \| c$.

B.4.2 Operational semantics

In this section terminology is introduced with respect to the mathematical framework that is used to define an operational semantics. Both terminology and notation are taken from [BV95]. The goal of an operational semantics is, given an expression denoting a process in a certain state, to describe all possible activities that can be performed by the process in that state and to describe the state of the process after such an activity. This expression represents the initial state of the MSC. The activities that are considered for the operational semantics of MSC are the execution of an event and the termination of the MSC. Also the states resulting after such activities are described by means of expressions. If from a state s an event a can be performed and the resulting state is represented by the expression s', then this is usually denoted by the ternary relation $s \xrightarrow{a} s'$. If in a given state s the process is capable of terminating immediately and successfully, this is indicated by means of $s\downarrow$.

The predicate $-\downarrow \subseteq \mathcal{P}$ is called the *termination predicate* as it indicates that a process has the possibility to terminate immediately and successfully. The set $\mathcal{P}$ denotes all expressions that can be built from the

constants and operators in the signature. It is assumed that all events are represented by atomic actions from the set A. Then the ternary relation $- \bar{\rightarrow} - \subseteq \mathcal{P} \times A \times \mathcal{P}$ is called the *transition relation*.

This predicate and these relations are defined by means of deduction rules (operational rules). A deduction rule is of the form $\frac{H}{C}$ where H is a set of premises and C is the conclusion. Each individual premise and the conclusion are of the form $s \stackrel{a}{\rightarrow} s'$ or $s\downarrow$ for arbitrary $s, s' \in \mathcal{P}$ and $a \in A$. Such a deduction rule should be interpreted as follows: If all premises are true, the conclusion, by definition, also holds. A special kind of deduction rule appears if the set of premises is empty ($H = \varnothing$). Such a deduction rule is also called a deduction axiom and usually simply denoted by the conclusion C. An example of a deduction axiom is deduction axiom (At 1) given in Table B.1:

$$\frac{}{a \stackrel{a}{\rightarrow} \varepsilon}.$$

This deduction axiom expresses that a process that is in a state represented by the atomic action a can perform event a and thereby evolves into a state represented by the expression ε. This expression ε indicates the state in which no events can be performed but in which it is possible to terminate successfully and immediately. This is expressed by the deduction axiom (E 1) also from Table B.1:

$$\frac{}{\varepsilon\downarrow}.$$

These are the only rules for expressions $a \in A$ and ε. The expression ε is used to denote an MSC without events.

Clearly the process a cannot terminate and the process ε cannot perform events. Note that these *negative* results are not explicitly defined. The following convention applies: If it is impossible to derive $s\downarrow$, then by definition not $s\downarrow$, which is denoted by $s \not\downarrow$. Similarly, if it is impossible to derive $s \stackrel{a}{\rightarrow} s'$, then by definition not $s \stackrel{a}{\rightarrow} s'$. This is usually denoted as $s \stackrel{a}{\not\rightarrow} s'$. Such negative results can also be used in the set of premises, and then these are called *negative premises*. The notation $s \stackrel{a}{\rightarrow}$ expresses that a process represented by the expression s can perform action a. This does not say anything about the resulting state after the execution of a. Formally, $s \stackrel{a}{\rightarrow}$ means that there exists a state s' such that $s \stackrel{a}{\rightarrow} s'$. Then $s \stackrel{a}{\not\rightarrow}$ should be read as there does not exists a state s' such that $s \stackrel{a}{\rightarrow} s'$, or for all states s', it is the case that $s \stackrel{a}{\not\rightarrow} s'$. These abbreviations extend to the relation $- \bar{\cdots\rightarrow} -$ to be introduced in Section B.4.7.

B.4.3 Equivalence of processes

Through the relations $\stackrel{a}{\rightarrow}$ and $\cdots\stackrel{a}{\rightarrow}$ and the predicate $\downarrow$, the *behaviour* of a process is defined. Using this behaviour it is possible to formally define when two processes should be considered equal. Many different notions of equivalence have been studied in literature. For MSC the prefered notion of equivalence is *bisimulation* [Par81].

Definition B.4.3.1 (Bisimulation relation) A binary relation $B \subseteq \mathcal{P} \times \mathcal{P}$ is called a bisimulation relation if for all $a \in A$ and $s, t \in \mathcal{P}$ with sBt the following conditions hold:

$$\forall_{s' \in \mathcal{P}}(s \stackrel{a}{\rightarrow} s' \Rightarrow \exists_{t' \in \mathcal{P}}(t \stackrel{a}{\rightarrow} t' \wedge s'Bt')),$$

$$\forall_{s' \in \mathcal{P}}(s \cdots\stackrel{a}{\rightarrow} s' \Rightarrow \exists_{t' \in \mathcal{P}}(t \cdots\stackrel{a}{\rightarrow} t' \wedge s'Bt')),$$

$$\forall_{t' \in \mathcal{P}}(t \stackrel{a}{\rightarrow} t' \Rightarrow \exists_{s' \in \mathcal{P}}(s \stackrel{a}{\rightarrow} s' \wedge s'Bt')),$$

$$\forall_{t' \in \mathcal{P}}(t \cdots\stackrel{a}{\rightarrow} t' \Rightarrow \exists_{s' \in \mathcal{P}}(s \cdots\stackrel{a}{\rightarrow} s' \wedge s'Bt')),$$

and

$$s\downarrow \Rightarrow t\downarrow,$$

$$t\downarrow \Rightarrow s\downarrow .$$

Two closed terms $p, q \in \mathcal{P}$ are bisimilar, notation $p \leftrightarroweq q$, if there exists a bisimulation relation B such that pBq.

Intuitively, two bisimilar processes can execute the same actions, and if they do so, will result in bisimilar processes again. For a concise treatment of bisimulation refer to [BW90].

B.4.4 Deadlock, empty process and atomic actions

In this section the smallest building blocks of the signature are introduced. These are divided into the special constants and the atomic actions. There are two special constants: δ and ε. The deadlock constant δ represents a process that cannot execute an event and cannot terminate. The empty process ε represents a process that cannot execute an event, but contrary to deadlock it terminates successfully.

The set of atomic actions is a parameter of the term algebra. In the context of Message Sequence Charts it is chosen to represent the events of the MSC language such as output and input of a message, timer statements, and local actions. As is the case with MSC, each smallest event is defined on an instance. To mimic this in the term algebra the existence of a total mapping $\ell : A \rightarrow Id$ is assumed which associates to an atomic action an identifier representing an instance name.

The operational semantics contains two relations and one predicate on processes. The *transition* relation $x \xrightarrow{a} x'$ means that process x can perform event a and thereby evolves into process x'. Stated differently: in a state x event a can be performed and state x' will then be entered. The *termination* predicate $x\downarrow$ means that the process x can terminate immediately and successfully. The *permission* relation $x \cdots\!\xrightarrow{a} x'$ will be explained later when relevant.

Table B.1: Deduction rules for constants: $a, b \in A$

$$\frac{}{\varepsilon\downarrow}\text{(At 1)} \qquad \frac{}{a \xrightarrow{a} \varepsilon}\text{(E 1)}$$

As indicated before, the empty process ε is capable of terminating immediately and successfully. This is expressed by the deduction rule $\varepsilon\downarrow$ (in the form of a deduction axiom). An atomic action a can execute event a and thereby it evolves into the empty process: $a \xrightarrow{a} \varepsilon$.

The deduction rules for the permission relation will be discussed when the time is right. For now they will only play a minor role.

B.4.5 Delayed choice

The structured operational semantics associated to delayed choice by means of the deduction rules presented in Table B.2 eminently illustrates the purposes of this operator. The deduction rules for $\xrightarrow{a}$ clearly express that $x \mp y$ can perform an a-transition thereby resolving the choice if exactly one of its operands can, and in the case that both operands can perform an a-transition the choice is not yet resolved.

The deduction rules for the termination predicate $\downarrow$ and the transition relation $\xrightarrow{a}$ from Table B.2 are taken from [BM94] where the delayed choice operator was introduced in the setting of bisimulation semantics as a

Table B.2: Deduction rules for delayed choice

$$\frac{x\downarrow}{x \mp y\downarrow}\text{(DC 1)} \qquad \frac{y\downarrow}{x \mp y\downarrow}\text{(DC 2)}$$

$$\frac{x \xrightarrow{a} x',\ y \not\xrightarrow{a}}{x \mp y \xrightarrow{a} x'}\text{(DC 3)} \qquad \frac{x \not\xrightarrow{a},\ y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'}\text{(DC 4)} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'}\text{(DC 5)}$$

means of composing MSCs. The deduction rules (DC 1) and (DC 2) express that the alternative composition of two processes has the option to terminate if and only if at least one of the alternatives has this option.

Example B.4.5.1 The process $action(i, a) \mp \varepsilon$ has an option to terminate as the second alternative has this option. On the contrary the process $action(i, a) \mp action(j, b)$ does not have an option to terminate as none of its alternatives can terminate.

The deduction rules (DC 3) and (DC 4) express that, in the situation that exactly one of the alternatives can execute an action a, the alternative composition can execute this event as well and that the execution of this event resolves the choice.

Example B.4.5.2 The process $action(i, a) \mp action(j, b)$ can execute the action $action(i, a)$ and the action $action(j, b)$. In both cases the action can be executed by only one of the alternatives. Thus in both cases making a choice between the alternatives cannot be avoided. Operationally this is seen as follows:

$$action(i, a) \mp action(j, b) \xrightarrow{action(i,a)} \varepsilon$$

and

$$action(i, a) \mp action(j, b) \xrightarrow{action(j,b)} \varepsilon.$$

Deduction rule (DC 5) deals with the situation that both alternatives can execute an action a. It states that, in that case, the alternative composition can execute a and, moreover, that there remain two alternatives.

Example B.4.5.3 The process $action(i, a) \mp action(i, a)$ has two alternatives both of which can execute action $action(i, a)$. The choice between the alternatives is not resolved. Operationally this can be seen as follows:

$$action(i, a) \mp action(i, a) \xrightarrow{action(i,a)} \varepsilon \mp \varepsilon.$$

The delayed choice is commutative and associative and deadlock is a unit for delayed choice. These properties are exactly what was required for the delayed choice. These properties enable the definition of a multinary delayed choice operator as in the following definition.

Definition B.4.5.4 (Multinary delayed choice) Let I be a finite set. Let P_i be a process term in which only the variable i occurs freely. Then the multinary delayed choice operator is defined by

$$\bigmp_{i\in I} P_i = \begin{cases} \delta & \text{if } I = \varnothing, \\ P_j \mp \left(\bigmp_{i\in I\setminus\{j\}} P_i\right) & \text{if } j \in I. \end{cases}$$

B.4.6 Delayed parallel composition

The delayed parallel composition of two processes is the interleaved execution of the events of the processes while maintaining the ordering of events as specified by the processes in isolation. This operator is a delayed version of the interleaving operators normally used. If both processes that are composed by means of delayed parallel composition can perform the same event, it is not visible which of the two is actually executed. In other word, a delayed choice is made between the two occurrences. In this aspect the delayed parallel composition operator used for the semantics of MSC differs from the interleaving operators of ACP-style process algebras [BW90]. The deduction rules for the delayed parallel composition operator are given in Table B.3.

Table B.3: Deduction rules for delayed parallel composition

$$\frac{x\downarrow,\ y\downarrow}{x \parallel y\downarrow}\text{(DP 1)}$$

$$\frac{x \xrightarrow{a} x',\ y \not\xrightarrow{a}}{x \parallel y \xrightarrow{a} x' \parallel y}\text{(DP 2)} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x' \parallel y \mp x \parallel y'}\text{(DP 3)} \qquad \frac{x \not\xrightarrow{a},\ y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'}\text{(DP 4)}$$

Deduction rule (DP 1) expresses that the delayed parallel composition of two processes has an option to terminate if and only if both processes have this option.

Example B.4.6.1 The process $(a \mp \varepsilon) \parallel (b \mp \varepsilon)$ has an option to terminate as both $a \mp \varepsilon$ and $b \mp \varepsilon$ have this option. Operationally this is seen as follows: $a \mp \varepsilon\downarrow$ and $b \mp \varepsilon\downarrow$ and therefore by deduction rule (DP 1) also $(a \mp \varepsilon) \parallel (b \mp \varepsilon)\downarrow$.

The process $a \parallel \varepsilon$ does not have an option to terminate as the left-hand side a of the delayed parallel composition does not have this option ($a \not\downarrow$) and therefore deduction rule (DP 1) is not applicable.

The deduction rules (DP 2) and (DP 4) express that if exactly one of the operands of a delayed parallel composition can execute an action a, then the delayed parallel composition can and it is known which operand has actually executed a.

Example B.4.6.2 The process $a \parallel b$ is capable of performing action a and thereby it evolves into the process $\varepsilon \parallel b$. But it is also possible for this process to perform action b and then the process $a \parallel \varepsilon$ remains.

The deduction rule (DP 3) expresses that, in a situation that both operands can execute an action a, the delayed parallel composition can execute an a and moreover that it is not known which operand executed a. This is seen in the deduction rule by the term $x' \parallel y \mp x \parallel y'$. The first alternative results from the execution of a by process x and the second from the execution of a by process y. The fact that the process $x \parallel y$ evolves into the process $x' \parallel y \mp x \parallel y'$ indicates that it is not known which a has been executed.

Example B.4.6.3 An example illustrating the delayed nature of the delayed parallel composition is the process $a \parallel a$. It can perform the following sequence of transitions:

$$a \parallel a \xrightarrow{a} \varepsilon \parallel a \mp a \parallel \varepsilon \xrightarrow{a} \varepsilon \parallel \varepsilon \mp \varepsilon \parallel \varepsilon\downarrow\ .$$

The delayed parallel composition is commutative and associative and the empty process is a unit for delayed parallel composition. These properties enable the definition of a multinary delayed parallel composition operator as in the following definition.

Definition B.4.6.4 (Multinary delayed parallel composition) Let I be a finite set. Let P_i be a process term in which only the variable i occurs freely. Then the multinary delayed parallel composition operator is defined by

$$\Big\|_{i\in I} P_i = \begin{cases} \varepsilon & \text{if } I = \varnothing, \\ P_j \parallel \left(\Big\|_{i\in I\setminus\{j\}} P_i \right) & \text{if } j \in I. \end{cases}$$

B.4.7 Weak sequential composition

In order to explain the weak sequential composition operator it is necessary to consider the purpose of this operator in the semantics of MSC. The weak sequential composition operator is introduced to represent the vertical composition of MSCs. It has a similar behaviour as the delayed parallel composition operator, but additionally it maintains the ordering of events from instances that the MSCs have in common. Thus an event on instance i in the second MSC can only take place in situations where all events on instance i (if any) in the first MSC have already taken place.

However, there is a complication with respect to alternatives. Suppose that an MSC A is given that describes two alternatives. The first alternative only describes a local action a on instance i and the second alternative only contains a local action b on instance j. Suppose that this MSC is composed vertically with an MSC B that only contains a local action c on instance i. The vertical composition of the first alternative of MSC A with MSC B should not allow the execution of local action c as it must be preceded by local action a. The vertical composition of the second alternative of MSC A with MSC B can execute local action c as there are no events in the second alternative of MSC A that must precede the execution of local action c. Thus, one alternative of MSC A does not allow the execution of local action c and one alternative does allow the execution of local action c. The expected result is that the execution of local action c is allowed and moreover that if local action c is executed the first alternative disappears.

In order to deal with this aspect of the weak sequential composition operator the permission relation $\cdots\overset{a}{\rightarrow}$ is used. The proposition $x \cdots\overset{a}{\rightarrow} x'$ states that process x allows the execution of an action a and thereby evolves into the process x' due to the resolving of choices. On the other hand, the proposition $x \not\cdots\overset{a}{\rightarrow}$ indicates that x does not allow the execution of action a from a process with which x is composed vertically.

In an MSC every event is associated with an instance on which it is defined. In the operational semantics this is incorporated by assuming a mapping $\ell : A \rightarrow I$, where I represents the set of all instance names, which associates with an atomic action $a \in A$ the name of the instance it is defined on $\ell(a)$.

The deduction rules for the permission relation are given in Table B.4 and the deduction rules for weak sequential composition are given in Table B.5.

Deduction rule (WS 1) expresses that the weak sequential composition of two processes has an option to terminate if and only if both processes have this option.

Example B.4.7.1 The process $\varepsilon \circ (a \mp \varepsilon)$ has the option to terminate as both operands have this option: $\varepsilon\downarrow$ and $a \mp \varepsilon\downarrow$.

The deduction rules (WS 2), (WS 3) and (WS 4) deal with the transitions of the vertical composition of two processes. In the case that x can execute a and either y cannot execute a or x does not allow the execution of a by y, only the execution of a by x can take place. This is expressed by deduction rule (WS 2).

Example B.4.7.2 Suppose that $\ell(a) \neq \ell(b)$. The process $a \circ b$ can execute action a and evolves into the process $\varepsilon \circ b$ since $a \overset{a}{\rightarrow} \varepsilon$ and $b \overset{a}{\not\rightarrow}$.

In the case that both x and y can execute action a and x allows the execution of a by y, there are two possibilities for executing action a. A delayed choice of the individual occurrences of action a results. This is expressed by deduction rule (WS 3).

Table B.4: Deduction rules for the permission relation

$$\frac{}{\varepsilon \overset{a}{\cdots\!\rightarrow} \varepsilon} \qquad \frac{\ell(a) \neq \ell(b)}{b \overset{a}{\cdots\!\rightarrow} b}$$

$$\frac{x \overset{a}{\cdots\!\rightarrow} x',\ y \overset{a}{\cdots\!\rightarrow} y'}{x \mp y \overset{a}{\cdots\!\rightarrow} x' \mp y'}\text{(DC 6)} \qquad \frac{x \overset{a}{\cdots\!\rightarrow} x',\ y \overset{a}{\not\cdots\!\rightarrow}}{x \mp y \overset{a}{\cdots\!\rightarrow} x'}\text{(DC 7)} \qquad \frac{x \overset{a}{\not\cdots\!\rightarrow},\ y \overset{a}{\cdots\!\rightarrow} y'}{x \mp y \overset{a}{\cdots\!\rightarrow} y'}\text{(DC 8)}$$

$$\frac{x \overset{a}{\cdots\!\rightarrow} x',\ y \overset{a}{\cdots\!\rightarrow} y'}{x \parallel y \overset{a}{\cdots\!\rightarrow} x' \parallel y'}\text{(DP 5)} \qquad \frac{x \overset{a}{\cdots\!\rightarrow} x',\ y \overset{a}{\cdots\!\rightarrow} y'}{x \circ y \overset{a}{\cdots\!\rightarrow} x' \circ y'}\text{(WS 5)}$$

Table B.5: Deduction rules for weak sequential composition

$$\frac{x{\downarrow},\ y{\downarrow}}{x \circ y{\downarrow}}\text{(WS 1)}$$

$$\frac{x \overset{a}{\rightarrow} x',\ x \overset{a}{\not\cdots\!\rightarrow} \vee y \overset{q}{\not\rightarrow}}{x \circ y \overset{a}{\rightarrow} x' \circ y}\text{(WS 2)} \qquad \frac{x \overset{a}{\rightarrow} x',\ x \overset{a}{\cdots\!\rightarrow} x'',\ y \overset{a}{\rightarrow} y'}{x \circ y \overset{a}{\rightarrow} x' \circ y \mp x'' \circ y'}\text{(WS 3)}$$

$$\frac{x \overset{q}{\not\rightarrow},\ x \overset{a}{\cdots\!\rightarrow} x',\ y \overset{a}{\rightarrow} y'}{x \circ y \overset{a}{\rightarrow} x' \circ y'}\text{(WS 4)}$$

Example B.4.7.3 Suppose that $\ell(a) \neq \ell(b)$. The process $(a \mp b) \circ a$ can execute action a and thereby evolves into the process $\varepsilon \circ a \mp b \circ \varepsilon$. The first alternative of the resulting process describes the result of the execution of a by $a \mp b$ and the second alternative describes the result of the execution of a by the process a. Note that due to the execution of the second a, the alternative a from $a \mp b$ is not present anymore since $a \mp b \stackrel{a}{\cdots\!\rightarrow} b$.

In the case that x cannot execute an action a and y can and x permits the execution of a by y, there is one possibility of executing a. This is expressed by deduction rule (WS 4).

Example B.4.7.4 Suppose that $\ell(a) \neq \ell(b)$. The process $a \circ b$ can execute an action b since the second operand of the vertical composition can ($b \stackrel{b}{\rightarrow} \varepsilon$) and the first operand allows this ($a \stackrel{b}{\cdots\!\rightarrow} a$). The resulting process after the execution of action b is $a \circ \varepsilon$.

The n-times repeated application of weak sequential composition x^n is introduced as a shorthand. No operational rules are given for this operator.

Definition B.4.7.5 Let $n \in IN$. Then for $x \in \mathcal{P}$ the process x^n is defined inductively as follows:

$$x^n = \begin{cases} \varepsilon & \text{if } n = 0, \\ x \circ x^{n-1} & \text{if } n > 0. \end{cases}$$

Another convenient shorthand is the expression $x^{[m,n]}$ where $x \in \mathcal{P}$ and $m, n \in IN \cup \{\infty\}$. This expression indicates that at least m and at most n copies of x are composed by means of weak sequential composition. For example the expression $x^{[2,4]}$ represents the expression $x \circ x \mp x \circ (x \circ x) \mp x \circ (x \circ (x \circ x))$. If the minimal number of repetitions exceeds the maximal number of repetitions it is assumed that x is executed zero times.

Definition B.4.7.6 Let $m, n \in IN \cup \{\infty\}$. Then for $x \in \mathcal{P}$ the process $x^{[m,n]}$ is defined as follows:

$$x^{[m,n]} = \begin{cases} \varepsilon & \text{if } m > n, \\ \mathop{\Large\mp}\limits_{m \leq i \leq n} x^i & \text{if } m \leq n \text{ and } n \neq \infty, \\ x^\infty & \text{if } m = n = \infty, \\ x^m \circ x^* & \text{if } m < n \text{ and } n = \infty. \end{cases}$$

B.4.8 Generalization of the composition operators

In this section generalized versions of the delayed parallel composition operator and the weak sequential composition operator are defined. These generalization are necessary to capture ordering requirements that need to be satisfied that refer to events from the different processes that are composed horizontally or vertically. The operators for delayed parallel and weak sequential composition are generalized by labeling them with a set of ordering requirements. An ordering requirement is a triple of the form $a \stackrel{n}{\mapsto} b$ where a and b are different atomic actions and n is a natural number. As a notational shorthand $a \stackrel{0}{\mapsto} b$ is written as $a \mapsto b$. Often the curly brackets of the set of ordering requirements are simply omitted.

The deduction rules for the generalized parallel composition operator are given in Table B.6 and the deduction rules for the generalized weak sequential composition operator are given in Table B.7. The auxiliary predicate *enabled* and the auxiliary mapping *upd* are explained and defined below.

Definition B.4.8.1 For $a \in A$ and S a set of ordering requirements:

$$\begin{array}{lcl} \mathit{enabled}(a, S) & \Longleftrightarrow & \forall_{b,c \in A, n \in IN}\, b \stackrel{n}{\mapsto} c \in S \implies (c \not\equiv a \vee n > 0), \\ \mathit{upd}(a, S) & = & \{b \stackrel{n}{\mapsto} c \mid b \stackrel{n}{\mapsto} c \in S \wedge b \not\equiv a \wedge c \not\equiv a\} \\ & \cup & \{b \stackrel{n+1}{\mapsto} c \mid b \stackrel{n}{\mapsto} c \in S \wedge b \equiv a\} \\ & \cup & \{b \stackrel{n-1}{\mapsto} c \mid b \stackrel{n}{\mapsto} c \in S \wedge c \equiv a \wedge n > 0\}. \end{array}$$

One difference between the deduction rules for $\|^S$ and $\circ^S$ and the deduction rules for $\|$ and $\circ$ is that the execution of an event a is restricted to the situations where $enabled(a, S)$ holds. Before this predicate can be explained first the interpretation of the ordering requirements must be explained.

The ordering requirement $a \overset{n}{\mapsto} b$ expresses that every $m + n^{th}$ execution of event b must be preceded by at least m executions of event a. The natural number n basically describes the difference in the number of executions of a and b. Thus if the set S contains the requirement $a \overset{0}{\mapsto} b$, it is not allowed to execute event b. The predicate $enabled(a, S)$ holds if and only if there is no ordering requirement in the set S that does not allow the execution of event a.

Table B.6: Deduction rules for generalized parallel composition

$$\frac{x\downarrow,\ y\downarrow}{x \|^S y\downarrow}\text{(HC 1)}$$

$$\frac{x \overset{a}{\rightarrow} x',\ y \overset{a}{\not\rightarrow},\ enabled(a, S)}{x \|^S y \overset{a}{\rightarrow} x' \|^{upd(a,S)} y}\text{(HC 2)} \qquad \frac{x \overset{a}{\rightarrow} x',\ y \overset{a}{\rightarrow} y',\ enabled(a, S)}{x \|^S y \overset{a}{\rightarrow} x' \|^{upd(a,S)} y \mp x \|^{upd(a,S)} y'}\text{(HC 3)}$$

$$\frac{x \overset{a}{\not\rightarrow},\ y \overset{a}{\rightarrow} y',\ enabled(a, S)}{x \|^S y \overset{a}{\rightarrow} x \|^{upd(a,S)} y'}\text{(HC 4)} \qquad \frac{x \cdots\overset{a}{\rightarrow} x',\ y \cdots\overset{a}{\rightarrow} y'}{x \|^S y \cdots\overset{a}{\rightarrow} x' \|^S y'}\text{(HC 5)}$$

Execution of an event a can lead to an update of the set of ordering requirements as follows. If event a occurs as the righthand side of a requirement this means that the natural number must be decreased by one. If event a occurs as the lefthand side of a requirement this means that the natural number must be increased by one. The requirements in which a does not occur, are not changed. For the purpose of updating the set S due to the execution of event a the mapping $upd(a, S)$ is used.

Table B.7: Deduction rules for generalized weak sequential composition

$$\frac{x\downarrow,\ y\downarrow}{x \circ^S y\downarrow}\text{(VC 1)} \qquad \frac{x \overset{a}{\rightarrow} x',\ x \not\cdots\overset{a}{\rightarrow} \vee y \overset{a}{\not\rightarrow},\ enabled(a, S)}{x \circ^S y \overset{a}{\rightarrow} x' \circ^{upd(a,S)} y}\text{(VC 2)}$$

$$\frac{x \overset{a}{\rightarrow} x',\ x \cdots\overset{a}{\rightarrow} x'',\ y \overset{a}{\rightarrow} y',\ enabled(a, S)}{x \circ^S y \overset{a}{\rightarrow} x' \circ^{upd(a,S)} y \mp x'' \circ^{upd(a,S)} y'}\text{(VC 3)}$$

$$\frac{x \overset{a}{\not\rightarrow},\ x \cdots\overset{a}{\rightarrow} x',\ y \overset{a}{\rightarrow} y',\ enabled(a, S)}{x \circ^S y \overset{a}{\rightarrow} x' \circ^{upd(a,S)} y'}\text{(VC 4)} \qquad \frac{x \cdots\overset{a}{\rightarrow} x',\ y \cdots\overset{a}{\rightarrow} y'}{x \circ^S y \cdots\overset{a}{\rightarrow} x' \circ^S y'}\text{(VC 5)}$$

Note that for both operators the deduction rules are similar to the deduction rules for their non-generalized counterparts. In fact $\|^{\varnothing} = \|$ and $\circ^{\varnothing} = \circ$.

Example B.4.8.2 Consider the process $?m \|^{!m \mapsto ?m} !m$. If the ordering requirement is not considered, i.e., the process $?m \| !m$ is considered, the actions $!m$ and $?m$ would be executed in any order. However, the presence of the requirement $!m \mapsto ?m$ blocks the execution of $?m$ as long as $!m$ has not been executed.

Thus the only possible execution for this process is

$$?m \parallel {}^{!m \mapsto ?m} !m \xrightarrow{!m} ?m \parallel {}^{!m \mapsto ?m}_{1} \varepsilon \xrightarrow{?m} \varepsilon \parallel {}^{!m \mapsto ?m} \varepsilon\downarrow .$$

B.4.9 Renaming operator

A mapping $f : A \to A$ is called an injective renaming iff for all $a, b \in A$, if $f(a) = f(b)$ then $a = b$. The renaming operator ρ_f renames an atomic action a into $f(a)$.

Table B.8: Deduction rules for renaming

$$\frac{x\downarrow}{\rho_f(x)\downarrow} \qquad \frac{x \overset{f^{-1}(a)}{\cdots\!\!\rightarrow} x'}{\rho_f(x) \overset{a}{\cdots\!\!\rightarrow} \rho_f(x')} \qquad \frac{x \overset{f^{-1}(a)}{\rightarrow} x'}{\rho_f(x) \overset{a}{\rightarrow} \rho_f(x')}$$

B.4.10 Repetitive behaviour

B.4.10.1 Iteration

The process x^* is the process that is capable of executing x any number of times including zero times. The choice of how many times the x is executed, however, is delayed. The consecutive occurrences of the process x are composed by means of weak sequential composition. Intuitively speaking the process x^* represents the process

$$\mathop{\Large\mp}_{i \geq 0} x^i = \varepsilon \mp x \mp x \circ x \mp \cdots .$$

The deduction rules for iteration are presented in Table B.9. The operation of the iteration operator is closely related to the operation of the weak sequential composition and the delayed choice as will be clear from the explanation of the deduction rules.

Table B.9: Deduction rules for iteration

$$\frac{}{x^*\downarrow}\text{(IT 1)}$$

$$\frac{x \overset{a}{\rightarrow} x',\ x \not\cdots\!\!\rightarrow^{a}}{x^* \overset{a}{\rightarrow} x' \circ x^*}\text{(IT 2)} \qquad \frac{x \overset{a}{\rightarrow} x',\ x \overset{a}{\cdots\!\!\rightarrow} x''}{x^* \overset{a}{\rightarrow} x''^* \circ (x' \circ x^*)}\text{(IT 3)}$$

$$\frac{x \not\cdots\!\!\rightarrow^{a}}{x^* \overset{a}{\cdots\!\!\rightarrow} \varepsilon}\text{(IT 4)} \qquad \frac{x \overset{a}{\cdots\!\!\rightarrow} x'}{x^* \overset{a}{\cdots\!\!\rightarrow} x'^*}\text{(IT 5)}$$

The process x^* has the option to execute x zero times and thus it that has the option to terminate successfully and immediately. This is what is expressed by deduction rule (IT 1).

The process x^* can perform an event a if the process x can do so. To determine what the resulting process will be it is of importance whether x also permits the event a. Suppose that $x \overset{a}{\rightarrow} x'$.

1) In the case that x does not permit event a, i.e., $x \not\cdot\stackrel{a}{\cdot}\rightarrow$, the only possibility for the execution of event a is the a from the first x in each of the sequences x^i ($i \geq 1$). If the alternative x^i ($i \geq 1$) is considered in isolation then it can perform an a event and the resulting process will be $x' \circ x^{i-1}$. Since this execution of a by the first occurrence of x in each of the alternatives is delayed the resulting process is

$$\mathop{\mp}_{i \geq 1} x' \circ x^{i-1},$$

which equals

$$x' \circ \mathop{\mp}_{i \geq 1} x^{i-1},$$

or in the formulation chosen for deduction rule (IT 2)

$$x' \circ x^*.$$

2) In the case that x does permit an a and thereby evolves into x'', i.e., $x \cdots\stackrel{a}{\rightarrow} x''$, there are many more possibilities for the execution of the a. Again the choice is delayed. For each of the sequences x^i an a event can be executed by any of the occurrences of x. Thus, if the alternative x^i is considered in isolation, then it can perform an a event and it thereby evolves into

$$\mathop{\mp}_{1 \leq j \leq i} x''^{j-1} \circ (x' \circ x^{i-j}).$$

Thus process x^* can perform an a event and thereby evolves into the process

$$\mathop{\mp}_{i \geq 1} \mathop{\mp}_{1 \leq j \leq i} x''^{j-1} \circ (x' \circ x^{i-j}),$$

which is equal to

$$x''^* \circ (x' \circ x^*).$$

This is expressed by deduction rule (IT 3).

Example B.4.10.1.1 Consider the process a^*. This process describes an arbitrary number of executions of action a. Only the first occurrence of a can be executed as all actions a necessarily are defined on the same instance and $a \not\cdot\stackrel{a}{\cdot}\rightarrow$. Thus $a^* \stackrel{a}{\rightarrow} \varepsilon \circ a^*$.

Example B.4.10.1.2 Consider the process $(a \circ b)^*$ where $\ell(a) \neq \ell(b)$. The first occurrence of b can be executed as a allows this ($a \cdots\stackrel{b}{\rightarrow} a$). The other occurrences of b cannot be executed as the previous occurrences of b prohibit this $a \circ b \not\cdot\stackrel{b}{\cdot}\rightarrow$. Thus, $(a \circ b)^* \stackrel{b}{\rightarrow} (a \circ \varepsilon) \circ (a \circ b)^*$.

Example B.4.10.1.3 Consider the process $(a \mp b)^*$ where $\ell(a) \neq \ell(b)$. Then $a \mp b \cdots\stackrel{a}{\rightarrow} b$ and $a \mp b \stackrel{a}{\rightarrow} \varepsilon$. Deduction rule (IT 3) then gives $(a \mp b)^* \stackrel{a}{\rightarrow} b^* \circ (\varepsilon \circ (a \mp b)^*)$. This result can be explained as follows. Consider the alternative $(a \mp b)^i$ for some $i \geq 1$. Clearly this alternative can execute action a. If $(a \mp b)^i$ is represented by

$$(a_1 \mp b_1) \circ (a_2 \mp b_2) \circ \cdots \circ (a_i \mp b_i)$$

one can observe that each a_j ($1 \leq j \leq i$) can be executed. The result of executing the j^{th} occurrence of a, i.e., a_j is then given by the following scheme

$$\begin{array}{ccccccccccc}
(a_1 \mp b_1) & \circ & (a_2 \mp b_2) & \circ \cdots \circ & (a_{j-1} \mp b_{j-1}) & \circ & (a_j \mp b_j) & \circ & (a_{j+1} \mp b_{j+1}) & \circ \cdots \circ & (a_i \mp b_i) \\
\cdots\stackrel{a}{\rightarrow} & & \cdots\stackrel{a}{\rightarrow} & & \cdots\stackrel{a}{\rightarrow} & & \stackrel{a}{\rightarrow} & & & & \\
b_1 & \circ & b_2 & \circ \cdots \circ & b_{j-1} & \circ & \varepsilon & \circ & (a_{j+1} \mp b_{j+1}) & \circ \cdots \circ & (a_i \mp b_i).
\end{array}$$

The delayed choice of all these possibilities gives $(a \mp b)^* \stackrel{a}{\rightarrow} b^* \circ (\varepsilon \circ (a \mp b)^*)$. The deduction rule expresses that an arbitrary occurrence of a can be executed and that as a consequence all previous occurrences of a are removed.

If process x does not permit the execution of action a, then x^* permits the execution of action a (IT 4). The reason for this is that x^* has the empty process ε as one of its alternatives. If, on the other hand, the process x does permit the execution of a and thereby evolves into x', then x^* also permits the execution of a and it evolves into x'^* (IT 5).

B.4.10.2 Unbounded repetition

The unbounded repetition of the process x, i.e., x^∞, corresponds to the notion where fresh copies of x are composed by means of weak sequential composition ad infinitum. The fact that the operation of unbounded repetition is so closely linked with the operation of weak sequential composition is visible in the deduction rules presented in Table B.10.

Table B.10: Deduction rules for unbounded repetition

$$\frac{x \xrightarrow{a} x',\ x \not\cdot\!\xrightarrow{a}}{x^\infty \xrightarrow{a} x' \circ x^\infty}\text{(UR 1)} \qquad \frac{x \xrightarrow{a} x',\ x \cdots\!\xrightarrow{a} x''}{x^\infty \xrightarrow{a} x''^* \circ (x' \circ x^\infty)}\text{(UR 2)}$$

$$\frac{x \cdots\!\xrightarrow{a} x'}{x^\infty \cdots\!\xrightarrow{a} x'^\infty}\text{(UR 3)}$$

Next consider the transition relation. There are only two relevant (disjoint) cases. The first is where x can execute an a event and x does not permit an a event, and the second is where x can execute an a event and also permits an a event. The other case, i.e., where x cannot execute an a event, does not give rise to a transition of x^∞ as none of the copies of x can execute the a event.

1) Suppose that x can perform an a event and thereby evolves into x' and suppose that x does not permit an a event. Then, following the deduction rules for weak sequential composition, the process x^∞ can only execute the a event from the first copy of x. Thus x^∞ performs the a event as well and thereby evolves into the process $x' \circ x^\infty$. This is expressed by deduction rule (UR 1).

2) Alternatively, if x permits and a event and thereby evolves into x'', there are in principle infinitely many possibilities for the execution of the a event, due to the permission for a each of the copies can perform the a event. Thus the deduction rule expresses that one of the copies of x will perform the a event. All preceding copies thus evolve into x''. Thus the process x^∞ evolves into the process $x''^* \circ (x' \circ x^\infty)$ after the execution of action a. This is expressed by deduction rule (UR 2).

The deduction rule for the permission relation (UR 3) is based directly on the deduction rule for weak sequential composition.

B.4.10.3 Recursion

The language *HMSC* can be used to describe infinitary behaviour. Therefore, the semantic domain is extended with recursive specifications. Let Σ be a signature and let V be a set of recursion variables. A recursive specification $E(V)$ is a set of equations

$$\{X = s_X(V) \mid X \in V\},$$

where each $s_X(V)$ is a term over the signature Σ and the set of variables V.

Let E be a recursive specification in which X occurs as a recursion variable. Then $\langle X \mid E \rangle$ denotes the solution for X with respect to the recursive specification E. For t a term possibly containing recursion variables, the process $\langle t \mid E \rangle$ denotes the process t with all occurrences of recursion variables r replaced by their solution $\langle r \mid E \rangle$.

An operational semantics for recursion which generates exactly one solution for every recursive specification is given by the deduction rules in Table B.12. In order to define the structured operational semantics for recursion it is necessary to determine whether the process represented by a recursion variable is capable of performing actions and of permitting actions. Therefore, auxiliary predicates $\ulcorner \xrightarrow{a} \urcorner$ and $\ulcorner \not\cdot\xrightarrow{a} \urcorner$ are introduced which are defined by the deduction rules in Table B.11.

Table B.11: Auxiliary predicates and relations for recursion ($a \in A$, $X = s_X \in E$)

$$\frac{}{\delta^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{\ell(a) = \ell(b)}{b^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner,\ y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{x \mp y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}$$

$$\frac{x^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{x \circ^S y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{x \circ^S y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{x \parallel^S y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{x \parallel^S y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}$$

$$\frac{x^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{x^{\infty\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{}{a^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \xrightarrow{a} \urcorner}{x \mp y^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{y^{\ulcorner} \xrightarrow{a} \urcorner}{x \mp y^{\ulcorner} \xrightarrow{a} \urcorner}$$

$$\frac{x^{\ulcorner} \xrightarrow{a} \urcorner,\ \mathit{enabled}(a, S)}{x \parallel^S y^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{y^{\ulcorner} \xrightarrow{a} \urcorner,\ \mathit{enabled}(a, S)}{x \parallel^S y^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \xrightarrow{a} \urcorner,\ \mathit{enabled}(a, S)}{x \circ^S y^{\ulcorner} \xrightarrow{a} \urcorner}$$

$$\frac{x^{\ulcorner} \cdots\xrightarrow{a} \urcorner,\ y^{\ulcorner} \xrightarrow{a} \urcorner,\ \mathit{enabled}(a, S)}{x \circ^S y^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \xrightarrow{a} \urcorner}{x^{*\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \xrightarrow{a} \urcorner}{x^{\infty\ulcorner} \xrightarrow{a} \urcorner}$$

$$\frac{x^{\ulcorner} \xrightarrow{f^{-1}(a)} \urcorner}{\rho_f(x)^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{x^{\ulcorner} \not\cdot\xrightarrow{f^{-1}(a)} \urcorner}{\rho_f(x)^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner} \qquad \frac{\langle s_X \mid E \rangle^{\ulcorner} \xrightarrow{a} \urcorner}{\langle X \mid E \rangle^{\ulcorner} \xrightarrow{a} \urcorner} \qquad \frac{\langle s_X \mid E \rangle^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}{\langle X \mid E \rangle^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner}$$

The proposition $x^{\ulcorner} \xrightarrow{a} \urcorner$ indicates that the process x can execute action a. The process $\langle X \mid E \rangle$ can execute a if s_X can. This is expressed by the deduction rules in Table B.11. The proposition $x^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ indicates that the process x cannot permit the execution of a. The process $\langle X \mid E \rangle$ cannot permit the execution of a if s_X cannot. This is expressed by the last deduction rule in Table B.11.

Example B.4.10.3.1 Consider the recursive specification given by $X = \varepsilon \circ Y$ and $Y = \varepsilon \circ X$. Does X permit a? In order to answer this question consider $X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$. Observe that $X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ iff $\varepsilon \circ Y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ iff $\varepsilon^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ or $Y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ iff $Y^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ iff $\varepsilon \circ X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ iff $X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$. So $X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ depends on $X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$. This means that $X^{\ulcorner} \not\cdot\xrightarrow{a} \urcorner$ is not derivable, so $X^{\ulcorner} \cdots\xrightarrow{a} \urcorner$.

The reason for splitting the definition of transition and permission into two phases, possibility to transit or permit and result after transition or permission, is that the the recursive specification has to be modified to obtain the results.

Table B.12: Structured operational semantics for recursion ($a \in A$, $X = s_X \in E$)

$$\frac{\langle s_X \mid E\rangle\downarrow}{\langle X \mid E\rangle\downarrow} \qquad \frac{\langle X \mid E\rangle \stackrel{a}{\rightarrow}}{\langle X \mid E\rangle \stackrel{a}{\rightarrow} \langle \hat{X}^a \mid \hat{E}^a\rangle} \qquad \frac{\langle X \mid E\rangle \stackrel{a}{\cdots\!\rightarrow}}{\langle X \mid E\rangle \stackrel{a}{\cdots\!\rightarrow} \langle \check{X}^a \mid \check{E}^a\rangle}$$

Definition B.4.10.3.2 Let E be a recursive specification. Then

$$\begin{aligned} \hat{E}^a &= \{\hat{X}^a = \hat{\rho}_a(s_X) \mid X = s_X \in E\} \cup E \cup \check{E}^a, \\ \check{E}^a &= \{\check{X}^a = \check{\rho}_a(s_X) \mid X = s_X \in E\}. \end{aligned}$$

where $\hat{\rho}_a$ is defined inductively by

$$\begin{aligned}
\hat{\rho}_a(\varepsilon) &= \delta \\
\hat{\rho}_a(\delta) &= \delta \\
\hat{\rho}_a(b) &= \begin{cases} \varepsilon & \text{if } a \equiv b \\ \delta & \text{if } a \not\equiv b \end{cases} \\
\hat{\rho}_a(x \mp y) &= \hat{\rho}_a(x) \mp \hat{\rho}_a(y) \\
\hat{\rho}_a(x \circ^S y) &= \begin{cases} \hat{\rho}_a(x) \circ^{upd(a,S)} y \mp \check{\rho}_a(x) \circ^{upd(a,S)} \hat{\rho}_a(y) & \text{if } y \stackrel{a}{\rightarrow} \text{ and } enabled(a, S) \\ \hat{\rho}_a(x) \circ^{upd(a,S)} y & \text{if } y \stackrel{g}{\not\rightarrow} \text{ and } enabled(a, S) \end{cases} \\
\hat{\rho}_a(x \parallel^S y) &= \begin{cases} \hat{\rho}_a(x) \parallel^{upd(a,S)} y \mp x \parallel^{upd(a,S)} \hat{\rho}_a(y) & \text{if } x \stackrel{a}{\rightarrow}, y \stackrel{a}{\rightarrow} \text{ and } enabled(a, S) \\ \hat{\rho}_a(x) \parallel^{upd(a,S)} y & \text{if } x \stackrel{a}{\rightarrow}, y \stackrel{g}{\not\rightarrow} \text{ and } enabled(a, S) \\ x \parallel^{upd(a,S)} \hat{\rho}_a(y) & \text{if } x \stackrel{g}{\not\rightarrow}, y \stackrel{a}{\rightarrow} \text{ and } enabled(a, S) \\ \delta & \text{otherwise} \end{cases} \\
\hat{\rho}_a(x^*) &= \begin{cases} \hat{\rho}_a(x) \circ x^* & \text{if } x \not\cdot\stackrel{a}{\rightarrow} \text{ and } x \stackrel{a}{\rightarrow} \\ \check{\rho}_a(x)^* \circ (\hat{\rho}_a(x) \circ x^*) & \text{if } x \stackrel{a}{\cdots\!\rightarrow} \text{ and } x \stackrel{a}{\rightarrow} \\ \delta & \text{if } x \stackrel{g}{\not\rightarrow} \end{cases} \\
\hat{\rho}_a(x^\infty) &= \begin{cases} \hat{\rho}_a(x) \circ x^\infty & \text{if } x \not\cdot\stackrel{a}{\rightarrow} \text{ and } x \stackrel{a}{\rightarrow} \\ \check{\rho}_a(x)^* \circ (\hat{\rho}_a(x) \circ x^\infty) & \text{if } x \stackrel{a}{\cdots\!\rightarrow} \text{ and } x \stackrel{a}{\rightarrow} \\ \delta & \text{if } x \stackrel{g}{\not\rightarrow} \end{cases} \\
\hat{\rho}_a(\rho_f(x)) &= \rho_f(\hat{\rho}_{f^{-1}(a)}(x)) \\
\hat{\rho}_a(X) &= \hat{X}^a
\end{aligned}$$

and where $\check{\rho}_a$ is defined inductively by

$$\begin{aligned}
\check{\rho}_a(\varepsilon) &= \varepsilon \\
\check{\rho}_a(\delta) &= \delta \\
\check{\rho}_a(b) &= \begin{cases} b & \text{if } \ell(a) \neq \ell(b) \\ \delta & \text{if } \ell(a) = \ell(b) \end{cases} \\
\check{\rho}_a(x \mp y) &= \check{\rho}_a(x) \mp \check{\rho}_a(y) \\
\check{\rho}_a(x \circ^S y) &= \begin{cases} \check{\rho}_a(x) \circ^S \check{\rho}_a(y) & \text{if } x \stackrel{a}{\cdots\!\rightarrow} \text{ and } y \stackrel{a}{\cdots\!\rightarrow} \\ \delta & \text{otherwise} \end{cases} \\
\check{\rho}_a(x \parallel^S y) &= \begin{cases} \check{\rho}_a(x) \parallel^S \check{\rho}_a(y) & \text{if } x \stackrel{a}{\cdots\!\rightarrow} \text{ and } y \stackrel{a}{\cdots\!\rightarrow} \\ \delta & \text{otherwise} \end{cases} \\
\check{\rho}_a(x^*) &= \begin{cases} \varepsilon & \text{if } x \not\cdot\stackrel{a}{\rightarrow} \\ \check{\rho}_a(x)^* & \text{if } x \stackrel{a}{\cdots\!\rightarrow} \end{cases} \\
\check{\rho}_a(x^\infty) &= \check{\rho}_a(x)^\infty \\
\check{\rho}_a(\rho_f(x)) &= \rho_f(\check{\rho}_{f^{-1}(a)}(x)) \\
\check{\rho}_a(X) &= \check{X}^a
\end{aligned}$$

Example B.4.10.3.3 Let $a, b \in A$ such that $\ell(a) \neq \ell(b)$. Consider the recursive specification given by

$$E = \{X = a \circ Y, Y = b \circ X\}$$

Can X execute action a? Yes it can as can be derived as follows: $X \xrightarrow{b}$ iff $a \circ Y \xrightarrow{b}$ iff $a \xrightarrow{b}$ and $Y \xrightarrow{b}$ iff $b \circ X \xrightarrow{b}$ iff $b \xrightarrow{b}$ or $b \cdots \xrightarrow{b}$ and $X \xrightarrow{b}$ iff $true$. So $X \xrightarrow{b}$. Now, what is the resulting process after the execution of b? The deduction rules state that $\langle X \mid E \rangle \xrightarrow{b} \langle \hat{X}^b \mid \hat{E}^b \rangle$ where

$$\hat{E}^b \quad = \quad \{\hat{X}^b = \delta \circ Y \mp a \circ \hat{Y}^b, \hat{Y}^b = \varepsilon \circ X \mp \delta \circ \hat{X}^b, X = a \circ Y, Y = b \circ X\}$$

Suppose that $c \in A$ such that $\ell(c) \neq \ell(a)$ and $\ell(c) \neq \ell(b)$. Now, consider the process $X \circ c$. In order to find out if this process can execute action c it is relevant to establish whether $X \overset{c}{\cdots}\rightarrow$. The deduction rule states that this is the case if $X \overset{c}{\cdots} \rightarrow$. Consider the following derivation: $X \not\cdot \xrightarrow{c}$ iff $a \circ Y \not\cdot \xrightarrow{c}$ iff $a \not\cdot \xrightarrow{c}$ or $Y \not\cdot \xrightarrow{c}$ iff $Y \not\cdot \xrightarrow{c}$ iff $b \circ X \not\cdot \xrightarrow{c}$ iff $b \not\cdot \xrightarrow{c}$ or $X \not\cdot \xrightarrow{c}$ iff $X \not\cdot \xrightarrow{c}$. Thus it is not possible to derive $X \not\cdot \xrightarrow{c}$. Therefore, $X \overset{c}{\cdots} \rightarrow$.

What is the resulting process after the execution of c. Following the deduction rules the resulting process is $\check{X}^c \circ \varepsilon$ where the process $\check{X}^c$ is defined by the recursive specification:

$$\check{E}^c = \{\check{X}^c = a \circ \check{Y}^c, \check{Y}^c = b \circ \check{X}^c\}$$

Example B.4.10.3.4 Consider the recursive specification $E = \{X = X\}$. Then clearly $X \overset{a}{\not\rightarrow}$, $X \overset{a}{\cdots} \rightarrow$ and $X \not\downarrow$ for all $a \in A$. Consider the recursive specification $E = \{X = Y, Y = X\}$. Also, in this case $X \overset{a}{\not\rightarrow}$, $X \overset{a}{\cdots} \rightarrow$ and $X \not\downarrow$ for all $a \in A$.

Example B.4.10.3.5 Next, consider the recursive specification $E = \{X = a \circ (Y \mp Z),\ Y = b \circ X,\ Z = c \circ X\}$, where a, b, and c are pairwise independent actions. Consider the process $\langle X \mid E \rangle \circ b'$ where b' and b are dependent. The action b' can only be executed if the process $\langle X \mid E \rangle$ permits the execution of b'. Then the recursive specification $E_{b'}$ must be constructed: $E_{b'} = \{X_{b'} = a \circ (Y_{b'} \mp Z_{b'}),\ Y_{b'} = \delta,\ Z_{b'} = c \circ X_{b'}\}$. Hence $\langle X \mid E \rangle \overset{b'}{\cdots} \rightarrow \langle X_{b'} \mid E_{b'} \rangle$. Thus the process $\langle X \mid E \rangle \circ b'$ is capable of performing the action b' and thereby evolves into the process $\langle X_{b'} \mid E_{b'} \rangle$. This example shows that by permitting action b' the choice for executing the b actions is resolved.

B.5 Textual syntax of MSC for the semantics

In this section we present the textual syntax that has actually been used for the definition of the formal semantics. In Section B.5.2 the textual syntax is given and in Section B.5.1 the changes that have lead to this textual syntax are explained.

B.5.1 Changes to the textual syntax

The textual syntax of MSC as presented in Recommendation Z.120 is changed in several aspects for the definition of the formal semantics. These changes can be subdivided into several categories. In Section B.5.1.1, we explain the changes to the textual syntax due to the fact that certain concepts are not treated in the formal semantics in this thesis. In Section B.5.1.7, we optimize the textual syntax by removing irrelevant information. In Section B.5.1.8, we explain the optimization of the textual syntax by considering certain constructions as abbreviations of other constructions. In Section B.5.1.9, we extend the textual syntax for the purpose of defining the formal semantics. In Section B.5.1.10, we explain the assumptions that have led to a further simplification of the textual syntax.

Besides the changes presented in the following sections also reformulations of the BNF rules have taken place in order to facilitate the definition of the formal semantics. These reformulations are replacing a nonterminal in the righthand sides of BNF rules by its productions, reformulating a BNF rule such that it facilitates inductive definitions and the introduction of new nonterminals to facilitate definitions.

All changes explained below are given with respect to the textual syntax of MSC as presented in Recommendation Z.120. The nonterminal ⟨⟩ denotes the empty word.

B.5.1.1 Parts of the language that are not treated

B.5.1.2 Instance-oriented representation

The textual syntax of MSC offers the possibility to describe an MSC in an instance-oriented way, in an event-oriented way and even by mixing these two description styles. For the definition of the semantics it is assumed that the MSC is represented in an event-oriented way. This restriction has great consequences for the textual syntax that is used for the definition of the formal semantics. These consequences are listed below:

- The MSC statements that are produced by the sequence of nonterminals ⟨old instance head statement⟩ ⟨instance event list⟩ are used to give the user of the language MSC the possibility to describe an instance in isolation. This combination is removed as an alternative for the productions of the nonterminal ⟨msc statement⟩.

- The shared conditions, shared MSC reference expressions and shared inline expressions are only used for the instance-oriented textual syntax and can therefore be omitted as alternative productions in the rule for the nonterminal ⟨non-orderable event⟩.

- As a consequence of the above ommissions a number of nonterminals is not necessary anymore. These are removed.

B.5.1.3 Instance decomposition

No semantics is provided for instance decomposition. As a consequence it is not necessary to indicate that an instance is decomposed by means of the productions of the nonterminal ⟨decomposition⟩ in the BNF rule for the nonterminal ⟨instance head statement⟩.

B.5.1.4 Substitution

No semantics is provided for the substitution mechanism in MSC reference expressions. The optional use of the nonterminal ⟨parameter substitution⟩ in the BNF rule for the nonterminal ⟨msc ref loop expr⟩ is therefore removed.

B.5.1.5 Incomplete message events and gates

No semantics is provided for lost and found message events that are sent to or received from the environment. This has several consequences for the textual syntax.

- A lost message event can only be sent to an instance or the environment without a gate name being associated with it. Similarly, a found message event can only be received from an instance or the environment without a gate name being associated with it. Therefore, the BNF rules for the nonterminals ⟨incomplete message output⟩ and ⟨incomplete message input⟩ is replaced by the rules

  ⟨incomplete message output⟩ ::= **out** ⟨msg identification⟩ **to lost** [⟨instance name⟩ | **env**]
  ⟨incomplete message input⟩ ::= **in** ⟨msg identification⟩ **from found** [⟨instance name⟩ | **env**]

- As incomplete message events cannot be sent to the environment or received from the environment the nonterminals ⟨output dest⟩ and ⟨input dest⟩ can be simplified to (and thus replaced by) the nonterminals ⟨output address⟩ and ⟨input address⟩ respectively.

B.5.1.6 Natural names

Natural names are used to specify the loop boundaries. For the semantics it is relevant that these natural names can be interpreted as natural numbers. Therefore, the nonterminal ⟨natural name⟩ has been replaced by the nonterminal ⟨decimal digit⟩.

B.5.1.7 Irrelevant information

In the textual syntax of MSC at several places information is provided that is irrelevant for the semantics. For the purpose of defining the semantics of MSC it is assumed that the MSCs do not contain this type of information.

- All parts of the textual syntax that specify comments are removed. The BNF rule for the nonterminal ⟨end⟩ is replaced by the BNF rule

  ⟨end⟩ ::= **;**

  As a consequence all occurrences of the nonterminal ⟨end⟩ are replaced by the terminal **;**. Furthermore, the possibility to have a text definition as an MSC statement is removed.

- Graphical parts of the grammar are removed. These are the nonterminals ⟨document head area⟩ and ⟨msc diagram⟩ which occur in the BNF rules for ⟨msc document head⟩ and ⟨msc document body⟩, respectively.

- The part of the MSC document head that contains references to external sources is removed. The BNF rule for the nonterminal ⟨document head⟩ is replaced by the BNF rule

  ⟨document head⟩ ::= **mscdocument** ⟨msc document name⟩ **;**

- The optional MSC interface (⟨msc interface⟩) is removed as it contains no information that is relevant to the definition of the formal semantics. This is only possible due to the extension of the textual syntax with a keyword **after** as explained in Section B.5.1.9.

- The part of the syntax referring to instance head and end statements (⟨instance head statement⟩ and ⟨instance end statement⟩) is removed. The information which instances are described in the MSC is only used as additional information that is useful when drawing an MSC starting from the textual representation. Also, this information is used to interpret the keyword **all**. We assume that all occurrences of the keyword **all** are replaced by the corresponding list of instance names (see Section B.5.1.8).

B.5.1.8 Shorthands

In the textual syntax of MSC at a number of places shorthands can be used in the textual syntax. For the purpose of defining semantics these can be treated as if they were replaced by their unabbreviated representations.

- The textual syntax for event definitions is restricted to contain exactly one instance event or multi instance event.

  ⟨event definition⟩ ::= ⟨instance name⟩ **:** ⟨instance event⟩ **;**
  | ⟨instance name list⟩ **:** ⟨multi instance event⟩ **;**

  The original event definitions that have more than one instance event or multi instance event can be replaced according to the following scheme:

  $$\begin{array}{lll} i & : & e_1; \\ & & e_2; \\ & & \vdots \\ & & e_n; \end{array}$$

  is replaced by

  $$\begin{array}{lll} i & : & e_1; \\ i & : & e_2; \\ \vdots & \vdots & \vdots \\ i & : & e_n; \end{array}$$

  A similar scheme is used for replacing the event definitions with more than one multi instance event. As a consequence the nonterminals ⟨instance event list⟩ and ⟨multi instance event list⟩ are redundant.

- The possibility to use the keyword **all** as a means to refer to all instances defined in the MSC is removed. It is assumed that all occurrences of this keyword are replaced by a list of instance names. The BNF rule for the nonterminal ⟨instance name list⟩ is replaced by the rule

  ⟨instance name list⟩ ::= ⟨instance name⟩ | ⟨instance name⟩ **,** ⟨instance name list⟩

- The possibility to use the keyword **loop** with only one inf-natural is removed.

  ⟨loop boundary⟩ ::= < ⟨inf natural⟩ **,** ⟨inf natural⟩ >

  The loop boundaries with one inf-natural can be replaced by a loop boundary with two inf-naturals according to the following scheme: $< k >$ is replaced by $< k,k >$.

- The possibility to use the keyword **loop** without specifying a loop boundary and the possibility to use the loop boundary without using the keyword **loop** are removed. An occurrence of the keyword **loop** without a loop boundary is considered a shorthand for the combination **loop** <1,**inf**>. An occurrence of a loop boundary *l* without the keyword **loop** is considered a shorthand for the combination **loop** *l*.
- The option inline expression is considered a shorthand for an alternative inline expression with two operands where the second operand is an empty MSC. The exception inline expression is considered a shorthand for an alternative inline expression where the second operand is the part of the MSC following the exception inline expression.
- The option MSC reference expression is considered a shorthand for an alternative MSC reference expression with two operands where the second operand is an empty MSC. The exception MSC reference expression is considered a shorthand for an alternative MSC reference expression where the second operand is the part of the MSC following the exception MSC reference expression.

B.5.1.9 Extensions

In favour of symmetry, the textual syntax is adapted in such a way that besides the already present before part, for orderable events, an additional after part is created such that both events in a causal ordering have the information that they are causally ordered. This change has several consequences for the textual syntax:

- The BNF rule for the nonterminal ⟨orderable event⟩ is replaced by the rule

⟨orderable event⟩ ::= [⟨event name⟩]
{ ⟨message event⟩
| ⟨incomplete message event⟩
| ⟨create⟩
| ⟨timer statement⟩
| ⟨action⟩
}
[**before** ⟨event name list⟩]
[**after** ⟨event name list⟩]

- The BNF rules for ⟨actual order in gate⟩, ⟨inline order out gate⟩ and ⟨inline order in gate⟩ are replaced by the rules

⟨actual order in gate⟩ ::= ⟨gate name⟩ **after** ⟨order dest⟩
⟨inline order out gate⟩ ::= ⟨gate name⟩ **after** ⟨order dest⟩ [**external before** ⟨order dest⟩]
⟨inline order in gate⟩ ::= ⟨gate name⟩ **before** ⟨order dest⟩ [**external after** ⟨order dest⟩]

B.5.1.10 Assumptions

- It is assumed that the message name alone is sufficient for establishing the correspondence between message input and message output events. As a consequence the optional message instance name and parameter list are removed.
- It is assumed that the MSC reference names and the inline expression names are unique with respect to the MSC document. The nonterminals ⟨msc reference name⟩ and ⟨inline expr name⟩ are replaced by the nonterminal ⟨ref name⟩.
- It is assumed that the timer name alone is sufficient for establishing if timer events correspond. Thus the nonterminal ⟨timer instance name⟩ is removed.
- It is assumed that every MSC reference expression or inline expression has an MSC reference identification or an inline expression identification respectively.

- It is assumed that there are no implicitly defined gates. As a consequence the via-part in the BNF rules for the nonterminals ⟨output address⟩ and ⟨input address⟩ becomes obligatory.

  Also, the optional gate name in the BNF rules for the nonterminals ⟨actual out gate⟩, ⟨actual in gate⟩, ⟨def out gate⟩ and ⟨def in gate⟩ becomes obligatory.

- It is assumed that all external and internal connections of gates of an inline expression are described in its inline gate interfaces. For MSC reference expressions it is assumed that all external connections are described in its MSC reference gate interface.

B.5.2 Textual syntax for semantics definition

If there are multiple rules for one nonterminal then this should be read as an extension and not as a replacement.

B.5.2.1 MSC documents

⟨msc document⟩ ::= **mscdocument** ⟨msc document name⟩ **;** ⟨msc document body⟩
⟨msc document body⟩ ::= ⟨⟩ | ⟨message sequence chart⟩ ⟨msc document body⟩

B.5.2.2 Message Sequence Charts

⟨message sequence chart⟩ ::= **msc** ⟨msc name⟩ **;** ⟨msc body⟩ **endmsc ;**

B.5.2.3 Events

⟨action⟩ ::= **action** ⟨action character string⟩
⟨message event⟩ ::= ⟨message output⟩ | ⟨message input⟩
⟨message output⟩ ::= **out** ⟨message name⟩ **to** ⟨input address⟩
⟨message input⟩ ::= **in** ⟨message name⟩ **from** ⟨output address⟩
⟨incomplete message event⟩ ::= ⟨incomplete message output⟩ | ⟨incomplete message input⟩
⟨incomplete message output⟩ ::= **out** ⟨message name⟩ **to lost** [⟨instance name⟩ | **env**]
⟨incomplete message input⟩ ::= **in** ⟨message name⟩ **from found** [⟨instance name⟩ | **env**]
⟨create⟩ ::= **create** ⟨instance name⟩ [**(** ⟨parameter list⟩ **)**]
⟨stop⟩ ::= **stop**
⟨timer statement⟩ ::= ⟨set⟩ | ⟨reset⟩ | ⟨timeout⟩
⟨set⟩ ::= **set** ⟨timer name⟩ [**(** ⟨duration name⟩ **)**]
⟨reset⟩ ::= **reset** ⟨timer name⟩
⟨timeout⟩ ::= **timeout** ⟨timer name⟩
⟨condition⟩ ::= **condition** ⟨condition name list⟩

⟨output address⟩ ::= ⟨instance name⟩
| **env via** ⟨gate name⟩
| ⟨reference identification⟩ **via** ⟨gate name⟩
⟨input address⟩ ::= ⟨instance name⟩
| **env via** ⟨gate name⟩
| ⟨reference identification⟩ **via** ⟨gate name⟩
⟨reference identification⟩ ::= **reference** ⟨ref name⟩
| **inline** ⟨ref name⟩
⟨parameter list⟩ ::= ⟨parameter name⟩ [**,** ⟨parameter list⟩]
⟨condition name list⟩ ::= ⟨condition name⟩ { **,** ⟨condition name⟩ }*

B.5.2.4 Causally ordered events

⟨orderable event⟩ ::= ⟨message event⟩ | ⟨incomplete message event⟩ | ⟨create⟩
| ⟨timer statement⟩ | ⟨action⟩
⟨ordered event⟩ ::= ⟨event name⟩ ⟨orderable event⟩ **before** ⟨event name list⟩
| ⟨event name⟩ ⟨orderable event⟩ **after** ⟨event name list⟩
| ⟨event name⟩ ⟨orderable event⟩
before ⟨event name list⟩
after ⟨event name list⟩
⟨event name list⟩ ::= ⟨order dest⟩ [**,** ⟨event name list⟩]
⟨order dest⟩ ::= ⟨event name⟩
| **env via** ⟨gate name⟩
| ⟨reference identification⟩ **via** ⟨gate name⟩

B.5.2.5 Coregions

⟨coregion⟩ ::= **concurrent ;** ⟨coevent list⟩ **endconcurrent**
⟨coevent list⟩ ::= ⟨⟩ | ⟨orderable event⟩ **;** ⟨coevent list⟩

B.5.2.6 MSC bodies

⟨msc body⟩ ::= ⟨⟩ | ⟨event definition⟩ ⟨msc body⟩
⟨event definition⟩ ::= ⟨instance name⟩ **:** ⟨instance event⟩ **;**
| ⟨instance name list⟩ **:** ⟨multi instance event⟩ **;**
⟨instance event⟩ ::= ⟨orderable event⟩ | ⟨non-orderable event⟩
⟨non-orderable event⟩ ::= ⟨stop⟩ | ⟨coregion⟩
⟨multi instance event⟩ ::= ⟨condition⟩ | ⟨msc reference⟩ | ⟨inline expr⟩
⟨instance name list⟩ ::= ⟨instance name⟩ | ⟨instance name⟩ **,** ⟨instance name list⟩

B.5.2.7 MSC reference expressions

⟨msc reference⟩ ::= **reference** ⟨ref name⟩ **:**
⟨msc ref expr⟩ ⟨reference gate interface⟩
⟨msc ref expr⟩ ::= ⟨msc ref par expr⟩ | ⟨msc ref par expr⟩ **alt** ⟨msc ref expr⟩
⟨msc ref par expr⟩ ::= ⟨msc ref seq expr⟩ | ⟨msc ref seq expr⟩ **par** ⟨msc ref par expr⟩
⟨msc ref seq expr⟩ ::= ⟨msc ref loop expr⟩ | ⟨msc ref loop expr⟩ **seq** ⟨msc ref seq expr⟩
⟨msc ref loop expr⟩ ::= [**loop** ⟨loop boundary⟩] ⟨expr body⟩
⟨expr body⟩ ::= **empty** | ⟨msc name⟩ | **(** ⟨msc ref expr⟩ **)**

⟨loop boundary⟩ ::= < ⟨inf natural⟩ **,** ⟨inf natural⟩ >
⟨inf natural⟩ ::= **inf** | ⟨decimal digit⟩+
⟨decimal digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨reference gate interface⟩ ::= ⟨⟩ | **; gate** ⟨ref gate⟩ ⟨reference gate interface⟩
⟨ref gate⟩ ::= ⟨actual out gate⟩ | ⟨actual in gate⟩
| ⟨actual order out gate⟩ | ⟨actual order in gate⟩
⟨actual out gate⟩ ::= ⟨gate name⟩ **out** ⟨message name⟩ **to** ⟨input address⟩
⟨actual in gate⟩ ::= ⟨gate name⟩ **in** ⟨message name⟩ **from** ⟨output address⟩
⟨actual order out gate⟩ ::= ⟨gate name⟩ **before** ⟨order dest⟩
⟨actual order in gate⟩ ::= ⟨gate name⟩ **after** ⟨order dest⟩

B.5.2.8 Inline expressions

⟨inline expr⟩ ::= ⟨loop expr⟩ | ⟨alt expr⟩ | ⟨par expr⟩
⟨loop expr⟩ ::= **loop** ⟨loop boundary⟩ **begin** ⟨ref name⟩ **;**
⟨inline gate interface⟩ ⟨msc body⟩
loop end
⟨alt expr⟩ ::= **alt begin** ⟨ref name⟩ **;** ⟨alt list⟩ **alt end**
⟨alt list⟩ ::= ⟨inline gate interface⟩ ⟨msc body⟩
| ⟨inline gate interface⟩ ⟨msc body⟩ **alt ;** ⟨alt list⟩
⟨par expr⟩ ::= **par begin** ⟨ref name⟩ **;** ⟨par list⟩ **par end**
⟨par list⟩ ::= ⟨inline gate interface⟩ ⟨msc body⟩
| ⟨inline gate interface⟩ ⟨msc body⟩ **par ;** ⟨par list⟩

⟨inline gate interface⟩ ::= ⟨⟩ | **gate** ⟨inline gate⟩ **;** ⟨inline gate interface⟩
⟨inline gate⟩ ::= ⟨inline out gate⟩ | ⟨inline in gate⟩
| ⟨inline order out gate⟩ | ⟨inline order in gate⟩
⟨inline out gate⟩ ::= ⟨def out gate⟩ **external out** ⟨message name⟩ **to** ⟨input address⟩
⟨inline in gate⟩ ::= ⟨def in gate⟩ **external in** ⟨message name⟩ **from** ⟨output address⟩
⟨inline order out gate⟩ ::= ⟨gate name⟩ **after** ⟨order dest⟩ **external before** ⟨order dest⟩
⟨inline order in gate⟩ ::= ⟨gate name⟩ **before** ⟨order dest⟩ **external after** ⟨order dest⟩
⟨def out gate⟩ ::= ⟨gate name⟩ **in** ⟨message name⟩ **from** ⟨output address⟩
⟨def in gate⟩ ::= ⟨gate name⟩ **out** ⟨message name⟩ **to** ⟨input address⟩

B.5.2.9 High-level Message Sequence Charts

⟨message sequence chart⟩ ::= **msc** ⟨msc name⟩ **; expr** ⟨msc expression⟩ **endmsc ;**
⟨msc expression⟩ ::= ⟨start⟩ ⟨node expression list⟩
⟨start⟩ ::= ⟨label name list⟩ **;**
⟨node expression list⟩ ::= ⟨⟩ | ⟨node expression⟩ ⟨node expression list⟩
⟨node expression⟩ ::= ⟨label name⟩ **:** { ⟨node⟩ **seq (** ⟨label name list⟩ **)** | **end** } **;**
⟨node⟩ ::= **empty**
| ⟨msc name⟩
| ⟨par expression⟩
| **condition** ⟨condition name list⟩
| **connect**
| **(** ⟨msc ref expr⟩ **)**
⟨par expression⟩ ::= **expr** ⟨msc expression⟩ **endexpr**
| **expr** ⟨msc expression⟩ **endexpr par** ⟨par expression⟩
⟨label name list⟩ ::= ⟨label name⟩ | ⟨label name⟩ **alt** ⟨label name list⟩

B.6 Semantics of Message Sequence Charts

B.6.1 Introduction

This section contains the denotational semantics of the fragment of MSC that is considered in this annex. This denotational semantics associates to an MSC in textual representation a process term. The textual representation used in the definition of this denotational semantics is identical to the textual syntax used in Section B.2. For some nonterminals there is no explicitly defined semantical mapping. This is the case if it is defined in terms of alternatives and nonterminals only. The semantics of such a nonterminal is obtained by considering the semantics of the nonterminals that are the alternatives on the righthand side of the BNF rule.

B.6.2 The approach

B.6.2.1 General introduction

In this section a denotational semantics for MSC documents is defined. It consists of a family of mappings $[\![]\!]$ which transform (part of) an MSC in textual representation into a process expression over the signature introduced in Section B.4. On the level of these process expressions an operational semantics has been defined in Section B.4. Two process expressions are consider to be equivalent if and only if they are (strongly) bisimilar. For a definition of this notion of equivalence we refer to Section B.4. Thus, the intended model of these process expressions is the term model modulo strong bisimulation. This notion of strong bisimulation is a congruence relation with respect to all operators from the signature. This means that in reasoning with/about those process expressions it is allowed to reason in a context.

The semantics is defined compositionally. Basically, this also refers to this notion of congruence discussed before. It also amounts to the fact the when defining the semantics of a piece of textual syntax no information is used that is only available in the context in which that part of syntax is used.

B.6.2.2 MSC documents

An MSC document contains a finite number of MSCs. In MSCs references to other MSCs can be used by means of the unique MSC names. A reference to an MSC with name A can be dealt with semantically be substituting the MSC name with the body of the MSC with that name. However, the approach that is followed in this annex is such that for every MSC in the MSC document a recursive equation is given that associates with an MSC with name A the equation $\overline{A} = S$ where $\overline{A}$ is a recursion variable associated with the MSC with name A and S is the semantics of the body of this MSC. As a consequence the semantics of an MSC document thus consists of a set of recursive equations.

B.6.2.3 Message Sequence Charts

Then both the semantics of an MSC with name A in the context of such an MSC document and the semantics of a reference to such an MSC are given by the recursion variable $\overline{A}$. This approach allows to consider the semantics of an MSC document by considering the semantics of every MSC in isolation.

B.6.2.4 Message Sequence Chart bodies

The body of an MSC in event-oriented textual representation basically consists of a list of event definitions. The intuition of such a list of event definitions is that these can be thought of as being composed vertically in the same order as the event definitions appear in the event-oriented representation. The approach that is followed to obtain the semantics of an MSC body can then be paraphrased by: an MSC body is the vertical composition of the event definitions that are contained.

In the approach towards the definition of the formal semantics in this annex one such event definition is almost an MSC on its own. It differs from an MSC in the following aspects:

1) It does not have a name.

2) It can have dangling message arrows and dangling general order arrows.

In Figure B.47 an MSC is given and in Figure B.48 its decomposition into three MSC fragments is given by means of horizontal dashed lines. Additionally the textual syntax of each of the MSC fragments is given in the figure.

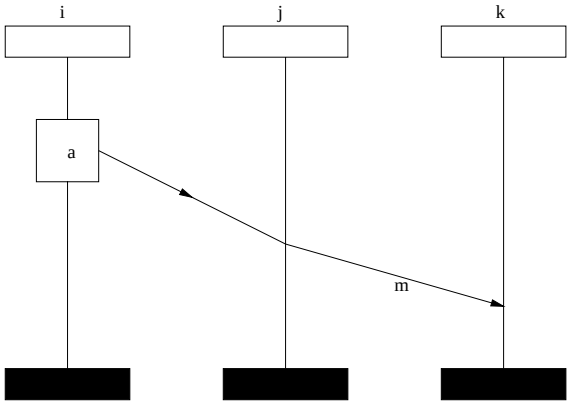


Figure B.47: An example MSC

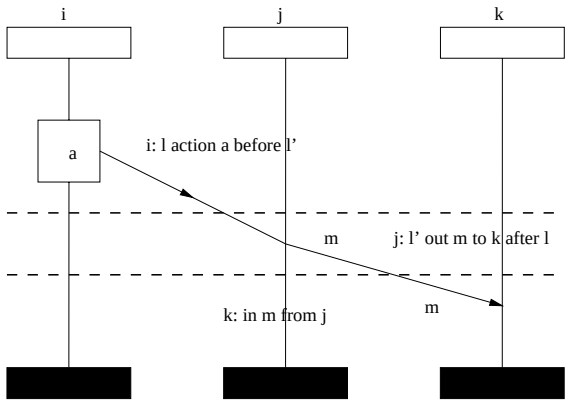


Figure B.48: Attributed example MSC

The MSC is decomposed into three *MSC fragments*. Each MSC fragment describes one event. Additionally textually the MSC fragments contain enough information to establish how the dangling message arrows and general order arrows are to be connected. For example the fact that local action *a* precedes the output of message *m* is available in the event names *l* and *l′* and the parts of the textual syntax that describe "*l* before *l′*" and "*l′* after *l*" respectively. In isolation the three MSC fragments could be represented as given in Figure B.49. In this figure dangling arrows are connected with the frame around the MSC fragment and the information that is necessary for determining if the dangling arrows should be connected is described close to the connection with the frame.

The connection of two dangling message arrows is appropriate if one is an output arrow and one an input arrow and if they agree on sender and receiver instance name and message name. The connection of two dangling general ordering arrows is appropriate if one is an outgoing arrow and the other an incoming arrow

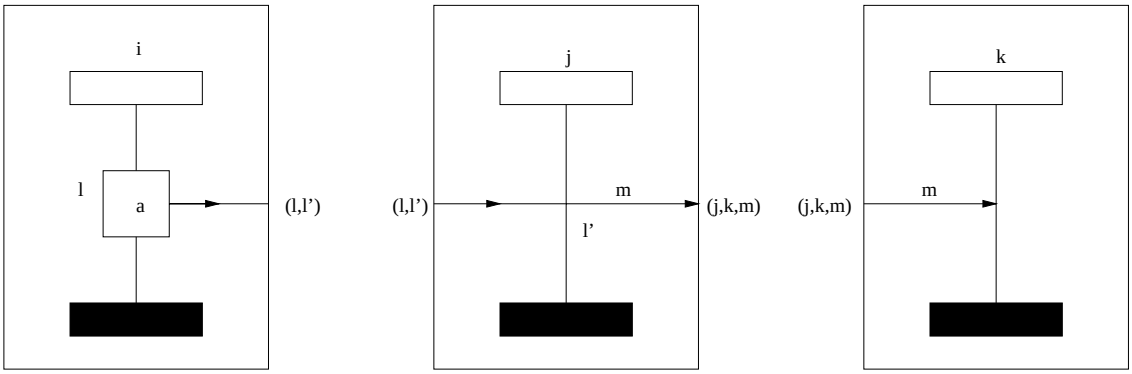


Figure B.49: Decomposed example MSC

and they refer to the general ordering. Graphically this cannot be detected, but textually this referring is done via event names.

In the example only very simple MSC fragments appeared. In principle, also more complex MSC fragments are used in the semantics definition. The following MSC fragments are distinguished:

1) Single instance events: An event attached to an instance with some dangling causal arrows. Examples of this type of MSC fragments are a local action a with event name l on instance i which is causally ordered before the events referred to by means of the event names $l_1, \cdots, l_N$ and causally ordered after the events referred to by the event names $k_1, \cdots, k_M$. Another example is the output of a message m from instance i to instance j.

2) Multi instance events: A multi instance event attached to a number of instances (at least one). The only such multi instance event is a condition.

3) Vertical composition: The vertical composition of two MSC fragments is again considered an MSC fragment. In such a vertical composition corresponding dangling arrows are connected and the required orderings are maintained.

4) Coregions: A coregion contains a number of single instance events.

5) Inline expressions: The composition of a number of MSC fragments by means of an operator.

6) MSC reference expressions:

B.6.2.5 Events

The single instance events are in the semantics denoted by atomic actions. The semantics of single instance events is considered in Section B.6.4. These atomic actions can be labeled by an event name and a set denoting the dangling general ordering arrows (see Section B.6.5). This is necessary as this information is needed when single instance events are composed vertically or horizontally.

B.6.2.6 Complex MSC fragments

Coregions, inline expressions and MSC reference expressions are also considered MSC fragments as these cannot necessarily be decomposed into smaller MSC fragments that are composed vertically and at the same time can occur in an MSC at every place one of the other MSC fragments can.

A coregion is the horizontal composition of a number of events that are defined on the same instance. Also for this horizontal composition dangling arrows need to be connected if appropriate. A coregion as a whole

can still have dangling arrows of both types. The horizontal composition mechanism used for obtaining a coregion from its events is defined in Section B.6.6. The semantics of a coregion is formally described in Section B.6.7.

Inline expressions are a means to describe the composition of two MSC fragments that have no dangling arrows. Thus for vertical and horizontal composition in inline expressions there is no need to connect dangling arrows.

An MSC reference expression is a textual formula which describes a composition of MSCs by means of a number of operators. The smallest building blocks of MSC reference expressions are references to other MSCs by means of their MSC name. Semantically, these are dealt with by means of recursion variables. This also means that a recursive equation must be given for such a recursion variable. This is the reason for associating a recursive specification with an MSC document.

B.6.3 Semantics of an MSC document

The semantics associated with an MSC document is a set of recursive equations. The recursion variables used in these recursive equations are the following: for every MSC in the MSC document a recursion variable is introduced. For an MSC with name id, this recursion variable is denoted as $\overline{id}$. Additionally recursion variables are introduced for every non-start node of the HMSCs in the MSC document (see Section B.6.11).

The mapping MSC associates with an MSC document a set of pairs of MSC names with their textual representation as they appear in that MSC document.

Definition B.6.3.1
The mapping $MSC : \mathcal{L}(\langle\text{msc document}\rangle) \rightarrow I\!P(\mathcal{L}(\langle\text{msc name}\rangle) \times \mathcal{L}(\langle\text{message sequence chart}\rangle))$ is for $docid \in \mathcal{L}(\langle\text{msc document name}\rangle)$ and $docbody \in \mathcal{L}(\langle\text{msc document body}\rangle)$ defined as follows:

$$MSC(\textbf{mscdocument}\ docid\ \textbf{;}\ docbody) = MSC(docbody).$$

The mapping $MSC : \mathcal{L}(\langle\text{msc document body}\rangle) \rightarrow I\!P(\mathcal{L}(\langle\text{msc name}\rangle) \times \mathcal{L}(\langle\text{message sequence chart}\rangle))$ is for $msc \in \mathcal{L}(\langle\text{message sequence chart}\rangle)$ and $docbody \in \mathcal{L}(\langle\text{msc document body}\rangle)$ defined inductively as follows:

$$\begin{array}{lcl} MSC() & = & \varnothing, \\ MSC(msc\ docbody) & = & \{(Name(msc), msc)\} \cup MSC(docbody), \end{array}$$

where the mapping $Name : \mathcal{L}(\langle\text{message sequence chart}\rangle) \rightarrow \mathcal{L}(\langle\text{msc name}\rangle)$ is for $id \in \mathcal{L}(\langle\text{msc name}\rangle)$, $body \in \mathcal{L}(\langle\text{msc body}\rangle)$ and $expr \in \mathcal{L}(\langle\text{msc expression}\rangle)$ defined as follows:

$$\begin{array}{lcl} Name(\textbf{msc}\ id\ \textbf{;}\ body\ \textbf{endmsc ;}) & = & id, \\ Name(\textbf{msc}\ id\ \textbf{;}\ \textbf{expr}\ expr\ \textbf{endmsc ;}) & = & id. \end{array}$$

As an MSC document cannot contain two or more MSCs with the same MSC name this set of pairs can be considered a mapping. In the sequel we will write $MSC(id)$ if we mean msc such that $(id, msc) \in MSC(doc)$. Note that we must be certain that we only do this for id such that there actually is an MSC with that name in the MSC document.

The mapping Eqs associates to an MSC document the set of recursive equations that describe the semantics of the MSCs in the MSC document. For an MSC (not an HMSC) this equation is of the form $\overline{id} = S$ where id is the name of the MSC and S is the semantics of the body of the MSC. The definition of the mapping Eqs for HMSCs is given in Definition B.6.11.1 in Section B.6.11.

Definition B.6.3.2 For $docid \in \mathcal{L}(\langle\text{msc document name}\rangle)$ and $docbody \in \mathcal{L}(\langle\text{msc document body}\rangle)$

$$Eqs(\textbf{mscdocument}\ docid\ \textbf{;}\ docbody) = Eqs(docbody).$$

For $msc \in \mathcal{L}(\langle\text{message sequence chart}\rangle)$ and $docbody \in \mathcal{L}(\langle\text{msc document body}\rangle)$

$$\begin{aligned} Eqs() &= \varnothing, \\ Eqs(msc\ docbody) &= Eqs(msc) \cup Eqs(docbody). \end{aligned}$$

For $id \in \mathcal{L}(\langle\text{msc name}\rangle)$ and $body \in \mathcal{L}(\langle\text{msc body}\rangle)$

$$Eqs(\textbf{msc}\ id\ \textbf{;}\ body\ \textbf{endmsc ;}) = \{\overline{id} = [\![body]\!]\}.$$

The semantics of an MSC msc with MSC name id from a given MSC document doc is then given by the solution of the recursion variable $\overline{id}$ in the recursive specification that consists of the equations $Eqs(doc)$. The notation introduced in Section B.4 is $\langle \overline{id} \mid Eqs(doc)\rangle$.

Definition B.6.3.3 Let $doc \in \mathcal{L}(\langle\text{msc document}\rangle)$. For arbitrary $msc \in \mathcal{L}(\langle\text{message sequence chart}\rangle)$ such that $(Name(msc), msc) \in MSC(doc)$

$$[\![msc]\!]_{doc} = \langle \overline{Name(msc)} \mid Eqs(doc)\rangle.$$

The way in which the semantics of MSC documents and MSCs is treated in this section makes it possible to deal with references to an MSC by using the appropriate recursion variable for the semantics. For example an MSC reference expression to an MSC A is semantically represented by $\overline{A}$.

B.6.4 Semantics of events

In this section the semantics for events is defined. In the Recommendation several types of events are distinguished. The first distinction is between single instance events and multi instance events. A single instance event is an event that is defined on exactly one instance. A multi instance event is an event that can be defined on one or more instances. Besides this distinction there is also a distinction between orderable and non-orderable events. An orderable event is an event that can be used in a general ordering and an non-orderable event is an event that may not be used in a general ordering. In Table B.13 the events that are present in the language MSC are placed in the correct class.

Table B.13: Classes of events

event	**single instance**	**multi instance**
non-orderable	instance stop	condition
orderable	local action (incomplete) message event instance create timer events	

B.6.4.1 Local actions

Local actions are represented in the semantics by atomic actions from the set A_{act} defined below. A local action that is defined on an instance i with action name a is denoted by $action(i, a)$.

Definition B.6.4.1.1 The set A_{act} is defined as follows:

$$A_{act} = \{action(i, a) \mid i \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge a \in \mathcal{L}(\langle\text{action character string}\rangle)\}.$$

Definition B.6.4.1.2 Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $a \in \mathcal{L}(\langle\text{action character string}\rangle)$

$$[\![\textbf{action}\ a]\!]_i = action(i, a).$$

B.6.4.2 Message events

The atomic actions that represent message output and message input events have four parameters. For a message output event the following information is maintained:

1) the name of the instance on which the event is executed;
2) an abstract representation of the gate via which the message is sent (if available);
3) the name of the instance that should receive the message (if available);
4) the name of the message.

Message output events as they occur in the textual syntax have either a gate part or a receiver instance name. Thus it would be possible to combine these two parameters into one. If a message output event is placed in a context which is capable of executing the corresponding message input event, then the name of the receiver instance becomes available. This does not mean that the gate part is not relevant anymore in such a case. The information through which gate the message goes remains relevant since omission of this information would make it impossible to distinguish two message output events that are sent through different gates.

At first sight it seems to be sufficient to maintain the name of the gate via which the message is sent to the environment. However, as there can be more than one reference to an MSC in an MSC document (even in one MSC), this still does not mean that the different occurrences of the message output event can be distinguished. For this purpose the reference identification is added to the gate name. The reference identification must therefore be unique within the MSC document.

For the input address of a message output event there are three possibilities. If it is an instance name then the message is not sent via a gate and the receiver instance name is known. This is indicated in the gate part by $\_$. If the input address of a message output event is a gate g in the environment this is indicated by means of $env(g)$. If the input address of a message output event is an actual gate g of an MSC reference expression or inline expression with reference identification l, then this is indicated by (l, g). This way the three possibilities can easily be distinguished. These three different notations for the representation of the gate parameter of the message output and input events are combined in the set $\mathcal{AMG}$ which is defined in Definition B.6.4.2.1. Besides these notations this set also contains elements of the form $((l, g), (l', g'))$ where l and l' are reference identifications and g and g' are gate names. These are added explicitly for the purpose of finding corresponding message output and message input events (see Section B.6.6).

Definition B.6.4.2.1 (Abstract Message Gate) The set $\mathcal{AMG}$ is defined as follows:

$$\begin{aligned}\mathcal{AMG} = \; & \{\_, env(g), (l, g), ((l, g), (l', g')) \mid g, g' \in \mathcal{L}(\langle\text{gate name}\rangle) \\ & \wedge\, l, l' \in \mathcal{L}(\langle\text{ref name}\rangle)\}.\end{aligned}$$

Definition B.6.4.2.2 The sets A_{out} and A_{in} are defined as follows:

$$\begin{aligned}A_{out} \;=\; & \{out(i, G, j, m), out(i, G, \_, m) \\ & \mid i, j \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge G \in \mathcal{AMG} \wedge m \in \mathcal{L}(\langle\text{message name}\rangle)\}, \\ A_{in} \;=\; & \{in(i, G, j, m), in(\_, G, j, m) \\ & \mid i, j \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge G \in \mathcal{AMG} \wedge m \in \mathcal{L}(\langle\text{message name}\rangle)\}.\end{aligned}$$

Definition B.6.4.2.3 Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $m \in \mathcal{L}(\langle\text{message name}\rangle)$, $g \in \mathcal{L}(\langle\text{gate name}\rangle)$, $j \in \mathcal{L}(\langle\text{instance name}\rangle)$, and $l \in \mathcal{L}(\langle\text{ref name}\rangle)$

$$\begin{aligned}[\![\textbf{out } m \textbf{ to } j]\!]_i &= out(i, \_, j, m), \\ [\![\textbf{out } m \textbf{ to env via } g]\!]_i &= out(i, env(g), \_, m), \\ [\![\textbf{out } m \textbf{ to reference } l \textbf{ via } g]\!]_i &= out(i, (l,g), \_, m), \\ [\![\textbf{out } m \textbf{ to inline } l \textbf{ via } g]\!]_i &= out(i, (l,g), \_, m), \\ [\![\textbf{in } m \textbf{ from } j]\!]_i &= in(j, \_, i, m), \\ [\![\textbf{in } m \textbf{ from env via } g]\!]_i &= in(\_, env(g), i, m), \\ [\![\textbf{in } m \textbf{ from reference } l \textbf{ via } g]\!]_i &= in(\_, (l,g), i, m), \\ [\![\textbf{in } m \textbf{ from inline } l \textbf{ via } g]\!]_i &= in(\_, (l,g), i, m).\end{aligned}$$

B.6.4.3 Incomplete message events

Lost message output events and found message input events are represented by atomic actions from the sets A_{lost} and A_{found} respectively.

Definition B.6.4.3.1 The sets A_{lost} and A_{found} are defined as follows:

$$\begin{array}{lcl} A_{lost} & = & \{\textit{lost-out}(i, j, m), \textit{lost-out}(i, \_, m), \textit{lost-out}(i, \textit{env}, m) \\ & & \mid i, j \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge m \in \mathcal{L}(\langle\text{message name}\rangle)\}, \\ A_{found} & = & \{\textit{lost-in}(i, j, m), \textit{lost-in}(\_, j, m), \textit{lost-in}(\textit{env}, j, m) \\ & & \mid i, j \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge m \in \mathcal{L}(\langle\text{message name}\rangle)\}. \end{array}$$

The first parameter of these atomic actions refers to the sender of the message, the second parameter refers to the receiver of the message and the third parameter represents the message identification. For a lost message output event it is possible that the receiver is an instance, a gate or unknown. If the intended receiver is a gate this is indicates by *env*. The case that the intended receiver is unknown is indicated by _. Similarly, for a found message input the sender can be an instance, a gate or unknown.

Definition B.6.4.3.2
Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $m \in \mathcal{L}(\langle\text{message name}\rangle)$ and $j \in \mathcal{L}(\langle\text{instance name}\rangle)$

$$\begin{array}{lcl} [\![\textbf{out } m \textbf{ to lost}]\!]_i & = & \textit{lost-out}(i, \_, m), \\ [\![\textbf{out } m \textbf{ to lost } j]\!]_i & = & \textit{lost-out}(i, j, m), \\ [\![\textbf{out } m \textbf{ to lost env}]\!]_i & = & \textit{lost-out}(i, \textit{env}, m), \\ [\![\textbf{in } m \textbf{ from found}]\!]_i & = & \textit{lost-in}(\_, i, m), \\ [\![\textbf{in } m \textbf{ from found } j]\!]_i & = & \textit{lost-in}(j, i, m), \\ [\![\textbf{in } m \textbf{ from found env}]\!]_i & = & \textit{lost-in}(\textit{env}, i, m). \end{array}$$

B.6.4.4 Instance create and instance stop events

Instance create events are represented by atomic actions from the set A_{cr} and instance stop events are represented by atomic actions from the set A_{stop}.

Definition B.6.4.4.1 The sets A_{cr} and A_{stop} are defined as follows:

$$\begin{array}{lcl} A_{cr} & = & \{\textit{create}(i, j, p), \textit{create}(i, j, \_) \mid i, j \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge p \in \mathcal{L}(\langle\text{parameter list}\rangle)\}, \\ A_{stop} & = & \{\textit{stop}(i) \mid i \in \mathcal{L}(\langle\text{instance name}\rangle)\}. \end{array}$$

The first parameter of these atomic actions represents the instance on which the event is defined. In case of a create event the second parameter of the atomic action is the name of the created instance and the third parameter represents the parameter list. If the parameter list is not specified for a create event, this is indicated in the atomic action by denoting the third parameter by _

Definition B.6.4.4.2
Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $j \in \mathcal{L}(\langle\text{instance name}\rangle)$ and $p \in \mathcal{L}(\langle\text{parameter list}\rangle)$,

$$\begin{array}{lcl} [\![\textbf{create } j]\!]_i & = & \textit{create}(i, j, \_), \\ [\![\textbf{create } j\textbf{(}p\textbf{)}]\!]_i & = & \textit{create}(i, j, p), \\ [\![\textbf{stop}]\!]_i & = & \textit{stop}(i). \end{array}$$

B.6.4.5 Timer events

Timer events are represented by atomic actions from the set A_{timer}.

Definition B.6.4.5.1 The set A_{timer} is defined as follows:

$$A_{timer} = \{ \; set(i, t, d), set(i, t, \_), reset(i, t), timeout(i, t) \mid i \in \mathcal{L}(\langle\text{instance name}\rangle) \wedge t \in \mathcal{L}(\langle\text{timer name}\rangle) \wedge d \in \mathcal{L}(\langle\text{duration name}\rangle) \}.$$

The first parameter of these atomic actions represents the name of the instance on which the timer event is defined, the second parameter represents the name of the timer and the third parameter represents the duration name associated with the timer set event. If no duration name is associated with the timer set event this is denoted by _. If in a timer event no duration name occurs this is represented in the atomic action by denoting its last parameter by _.

Definition B.6.4.5.2
Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $t \in \mathcal{L}(\langle\text{timer name}\rangle)$ and $d \in \mathcal{L}(\langle\text{duration name}\rangle)$,

$$\begin{aligned} [\![\textbf{set } t]\!]_i &= set(i, t, \_), \\ [\![\textbf{set } t\textbf{(}d\textbf{)}]\!]_i &= set(i, t, d), \\ [\![\textbf{reset } t]\!]_i &= reset(i, t), \\ [\![\textbf{timeout } t]\!]_i &= timeout(i, t). \end{aligned}$$

B.6.4.6 Conditions

Although conditions are not really events, they are only used as a means to restrict vertical composition in HMSCs, they are best treated in this section. With a condition no atomic action is associated. As a condition does not disallow any further events it is represented by the empty process ε.

Definition B.6.4.6.1 Then, for $cl \in \mathcal{L}(\langle\text{condition name list}\rangle)$

$$[\![\textbf{condition } cl]\!] = \varepsilon.$$

B.6.5 Semantics of causally ordered events

Semantically, events are represented by atomic actions. These atomic actions can have parameters which play a symbolic role. For example the output of a message with name *m* by instance *i* with receiver instance *j* is represented by $out(i, \_, j, m)$. The corresponding message input event is represented by $in(i, \_, j, m)$. With these parameters enough information is available to decide whether a message output and a message input are corresponding. For the correspondence of events that are involved in a causal ordering this is not so easy. For example if a local action with name *a* on instance *i* must precede a local action with name *b* on instance *j* then this cannot be determined from the atomic actions $action(i, a)$ and $action(j, b)$ representing these events. This implies that additional information has to be maintained.

There are three situations that need to be considered.

- the other end of the causal ordering is an event attached to an instance;
- the other end of the causal ordering is a gate on the frame of the MSC;
- the other end of the causal ordering is a gate on the frame of an MSC reference expression or an inline expression.

For each of these situations different information is available. Therefore, three different representations are used. Additionally, this has the advantage that the three situations can be distinguished.

In the first situation both events that are involved in the causal ordering are known via the event names. Therefore the causal ordering can easily be represented via the event names. For example, the event "i_1 **:** l_1 e_1 **before** l_2" describes that the event e_1 with event name l_1 is causally ordered before an unknown event with event name l_2. This is represented by labeling the atomic action representing the event e_1 with the pair

$l_1 \mapsto l_2$. The corresponding event, say "i_2 **:** l_2 e_2 **after** l_1", is labeled with the pair $l_1 \mapsto l_2$ as well. Thus it is easy to establish that these two events are ordered.

In the second situation only one of the events is available. In a broader context however the gate may be connected to another gate or event and then both events will be available. Thus, even although there is only one event, it still is necessary to maintain the information. An example of this situation is the event "i_1 **:** l_1 e_1 **before env via** g". The available information in this case is that the event with event name l_1 is ordered before an event that might be connected to gate g. This is represented by $l_1 \mapsto env(g)$. Later we will see that if this MSC is placed in a context in which the gate g is connected the information will be changed accordingly.

The third situation is comparable to the second situation. In this case however, it is known that the order arrow connects to an actual gate. Textually this is indicated by a reference to an MSC reference expression or an inline expression by means of a reference identification. An example is the event "i_1 **:** l_1 e_1 **before reference** l_2 **via** g". As there can be more than one occurrence of gate g due to multiple references to MSCs, the reference identification is essential information. The causal ordering is represented by the pair $l_1 \mapsto (l_2, g)$.

With an orderable event an event name can be associated. These event names are used to refer to an event when describing a causal ordering. The event names are also necessary to distinguish multiple occurrences of the same event. As a result it is necessary to label an atomic action representing an event by its event name. As one event can be involved in many general orderings the atomic action is labeled with a set of ordering requirements.

The sets $\mathcal{AOD}$ and $\mathcal{AOR}$ represent the information that is provided textually when an event is causally ordered. An abstract order destination, that is an element of the set $\mathcal{AOD}$, describes one half of a causal ordering. An abstract ordering requirement, that is an element of the set $\mathcal{AOR}$, describes both halves of a causal ordering.

Definition B.6.5.1 (Abstract Order Destination and Abstract Ordering Requirement) The set $\mathcal{AOD}$ is defined as follows:

$$\begin{array}{rcl} \mathcal{AOD} & = & \mathcal{L}(\langle\text{event name}\rangle) \\ & \cup & \{env(g) \mid g \in \mathcal{L}(\langle\text{gate name}\rangle)\} \\ & \cup & \mathcal{L}(\langle\text{ref name}\rangle) \times \mathcal{L}(\langle\text{gate name}\rangle) \\ & \cup & (\mathcal{L}(\langle\text{ref name}\rangle) \times \mathcal{L}(\langle\text{gate name}\rangle))^2. \end{array}$$

The set $\mathcal{AOR}$ is defined as follows:

$$\mathcal{AOR} \quad = \quad \mathcal{AOD} \times \mathcal{AOD}$$

Not all elements of $\mathcal{AOR}$ will appear in the semantics.

The mapping S associates with an order destination an element of the set $\mathcal{AOD}$, that is, an abstract order destination, as explained informally before.

Definition B.6.5.2
The mapping $S : \mathcal{L}(\langle\text{order dest}\rangle) \rightarrow \mathcal{AOD}$ is for $e \in \mathcal{L}(\langle\text{event name}\rangle)$, $l \in \mathcal{L}(\langle\text{ref name}\rangle)$ and $g \in \mathcal{L}(\langle\text{gate name}\rangle)$, defined as follows:

$$\begin{array}{lcl} S(e) & = & e, \\ S(\textbf{env via } g) & = & env(g), \\ S(\textbf{reference } l \textbf{ via } g) & = & (l, g), \\ S(\textbf{inline } l \textbf{ via } g) & = & (l, g). \end{array}$$

Then, some notation is introduced for the set of all atomic actions and for labelled atomic actions.

Definition B.6.5.3 (Labelled atomic actions) The set A is defined as follows:

$$A = A_{act} \cup A_{out} \cup A_{in} \cup A_{lost} \cup A_{found} \cup A_{cr} \cup A_{stop} \cup A_{timer}.$$

The sets LA, LA_{out}, LA_{in} and LA_{msg} are defined as follows:

$$\begin{array}{lcl} LA & = & \{a, a_e, a_e^O \mid a \in A \wedge e \in \mathcal{L}(\langle\text{event name}\rangle) \wedge O \subseteq \mathcal{AOR}\}, \\ LA_{out} & = & \{a, a_e, a_e^O \mid a \in A_{out} \wedge e \in \mathcal{L}(\langle\text{event name}\rangle) \wedge O \subseteq \mathcal{AOR}\}, \\ LA_{in} & = & \{a, a_e, a_e^O \mid a \in A_{in} \wedge e \in \mathcal{L}(\langle\text{event name}\rangle) \wedge O \subseteq \mathcal{AOR}\}, \\ LA_{msg} & = & LA_{out} \cup LA_{in}. \end{array}$$

The mapping $\ell : A \to \mathcal{L}(\langle\text{instance name}\rangle)$ is defined as follows:

$$\begin{array}{lcllcl} \ell(action(i, a)) & = & i, & \ell(lost\text{-}in(\_, j, m)) & = & j, \\ \ell(out(i, G, j, m)) & = & i, & \ell(lost\text{-}in(env, j, m)) & = & j, \\ \ell(out(i, G, \_, m)) & = & i, & \ell(create(i, j, p)) & = & i, \\ \ell(in(i, G, j, m)) & = & j, & \ell(create(i, j, \_)) & = & i, \\ \ell(in(\_, G, j, m)) & = & j, & \ell(stop(i)) & = & i, \\ \ell(lost\text{-}out(i, j, m)) & = & i, & \ell(set(i, t, d)) & = & i, \\ \ell(lost\text{-}out(i, \_, m)) & = & i, & \ell(set(i, t, \_)) & = & i, \\ \ell(lost\text{-}out(i, env, m)) & = & i, & \ell(reset(i, t)) & = & i, \\ \ell(lost\text{-}in(i, j, m)) & = & j, & \ell(timeout(i, t)) & = & i. \end{array}$$

The mapping $\ell : LA \to \mathcal{L}(\langle\text{instance name}\rangle)$ is for $a \in A$, $e \in \mathcal{L}(\langle\text{event name}\rangle)$ and $O \subseteq \mathcal{AOR}$ defined as follows:

$$\begin{array}{lcl} \ell(a) & = & \ell(a), \\ \ell(a_e) & = & \ell(a), \\ \ell(a_e^O) & = & \ell(a). \end{array}$$

The semantics of an ordered event is obtained as follows. The event that is ordered is translated into an atomic action as defined in Section B.6.4. This atomic action is labelled with the event name and a set of abstract ordering requirements.

Definition B.6.5.4 (Ordered events) Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $l \in \mathcal{L}(\langle\text{event name}\rangle)$, enl, $enl' \in \mathcal{L}(\langle\text{event name list}\rangle)$ and $e \in \mathcal{L}(\langle\text{orderable event}\rangle)$ an orderable event,

$$\begin{array}{lcl} [\![l\ e\ \textbf{before}\ enl]\!]_i & = & ([\![e]\!]_i)_l^{before_l(enl)}, \\ [\![l\ e\ \textbf{after}\ enl]\!]_i & = & ([\![e]\!]_i)_l^{after_l(enl)}, \\ [\![l\ e\ \textbf{before}\ enl\ \textbf{after}\ enl']\!]_i & = & ([\![e]\!]_i)_l^{before_l(enl) \cup after_l(enl')}. \end{array}$$

where the mappings $before_l, after_l : \mathcal{L}(\langle\text{event name list}\rangle) \to I\!P(\mathcal{AOR})$ are, for $d \in \mathcal{L}(\langle\text{order dest}\rangle)$ and $enl \in \mathcal{L}(\langle\text{event name list}\rangle)$, defined as follows:

$$\begin{array}{lcllcl} before_l(d) & = & \{(l, S(d))\}, & after_l(d) & = & \{(S(d), l)\}, \\ before_l(d\ \textbf{,}\ enl) & = & \{(l, S(d))\} \cup before_l(enl), & after_l(d\ \textbf{,}\ enl) & = & \{(S(d), l)\} \cup after_l(enl). \end{array}$$

B.6.6 Vertical and horizontal composition of MSC fragments

If two MSC fragments are composed vertically or horizontally, it is possible that the MSC fragments contain corresponding message events or corresponding causally ordered events. A message output event and a message input event are considered to be corresponding if they have the same message name and either the same sender instance and receiver instance, or the message output is sent to a gate which is connected to the gate from which the message input is received. In a similar way it can be established that two causally ordered events are corresponding. In Definition B.6.6.1 these notions are formalized.

Definition B.6.6.1
The relation $\circ\!\!\rightarrow\!\!\circ \subseteq LA \times LA$ is the smallest relation that satisfies: for all $i, j \in \mathcal{L}(\langle\text{instance name}\rangle)$, $m \in \mathcal{L}(\langle\text{message name}\rangle)$, $G \in \mathcal{AMG}$, $O, O' \subseteq \mathcal{AOR}$ and $e, e' \in \mathcal{L}(\langle\text{event name}\rangle)$

$$\begin{array}{l} out(i, \_, j, m)_e^O \circ\!\!\rightarrow\!\!\circ in(i, \_, j, m)_{e'}^{O'}, \\ out(i, G, \_, m)_e^O \circ\!\!\rightarrow\!\!\circ in(\_, G, j, m)_{e'}^{O'}. \end{array}$$

The relation $\circ\!\!\rightarrow\!\!-\!\!\circ \subseteq LA \times LA$ is for $a, b \in A$, $O, O' \subseteq \mathcal{AOR}$ and $e, e' \in \mathcal{L}(\langle\text{event name}\rangle)$ defined by

$$a_e^O \circ\!\!\rightarrow\!\!-\!\!\circ b_{e'}^{O'} \iff (e, e') \in O \cap O' \vee (e, e') \in O \circ O',$$

where $\circ : I\!P(\mathcal{AOR}) \rightarrow I\!P(\mathcal{AOR})$ is for $O, O' \subseteq \mathcal{AOR}$ defined by

$$O \circ O' = \{(s, u) \mid \exists_{t \in \mathcal{AOD}}(s, t) \in O \wedge (t, u) \in O'\}.$$

The relation $\hookrightarrow \subseteq LA \times LA$ is the smallest relation that satisfies: for all $a \in A$, $O, O' \subseteq \mathcal{AOR}$, $e, e' \in \mathcal{L}(\langle\text{event name}\rangle)$, $i, j \in \mathcal{L}(\langle\text{instance name}\rangle)$ and $p \in \mathcal{L}(\langle\text{parameter list}\rangle)$ defined by

$$create(i, j, p)_e^O \hookrightarrow a_{e'}^{O'} \iff \ell(a_{e'}^{O'}) = j.$$

The mapping α associates with a process the set of atomic actions. The mapping M associates with a process the set of atomic actions that refer to the message output and message input events that occur in the process. The mapping O associates with a process the set of all atomic actions that refer to an ordered event.

Definition B.6.6.2 The mapping $\alpha : \mathcal{P} \rightarrow I\!P(LA)$ is for $\otimes \in \{\mp, \circ^S, \|^S \mid S \subseteq LA \times IN \times LA\}$, $\odot \in \{^*, ^\infty, ^{[m,n]} \mid m, n \in IN \cup \{\infty\}\}$, $a \in LA$ and $x, y \in \mathcal{P}$ defined as follows:

$$\begin{array}{lcl}
\alpha(\varepsilon) & = & \varnothing, \\
\alpha(\delta) & = & \varnothing, \\
\alpha(a) & = & \{a\}, \\
\alpha(x \otimes y) & = & \alpha(x) \cup \alpha(y), \\
\alpha(x^{\odot}) & = & \alpha(x), \\
\alpha(\rho_f(x)) & = & \{f(a) \mid a \in \alpha(x)\}, \\
\alpha(\langle X \mid E \rangle) & = & \langle X_\alpha \mid E_\alpha \rangle,
\end{array}$$

where E_α is defined as follows:

$$E_\alpha = \{X_\alpha = \alpha(s_X) \mid X = s_X \in E\}.$$

The mapping $M : \mathcal{P} \rightarrow I\!P(LA_{msg})$ is for $\otimes \in \{\mp, \circ^S, \|^S \mid S \subseteq LA \times IN \times LA\}$, $\odot \in \{^*, ^\infty, ^{[m,n]} \mid m, n \in IN \cup \{\infty\}\}$, $a \in LA$ and $x, y \in \mathcal{P}$ defined as follows:

$$\begin{array}{lcl}
M(\varepsilon) & = & \varnothing, \\
M(\delta) & = & \varnothing, \\
M(a) & = & \begin{cases} \{a\} & \text{if } a \in LA_{msg}, \\ \varnothing & \text{otherwise,} \end{cases} \\
M(x \otimes y) & = & M(x) \cup M(y), \\
M(x^{\odot}) & = & M(x), \\
M(\rho_f(x)) & = & \{f(a) \mid a \in M(x)\}, \\
M(\langle X \mid E \rangle) & = & \langle X_M \mid E_M \rangle,
\end{array}$$

where E_M is defined as follows:

$$E_M = \{X_M = M(s_X) \mid X = s_X \in E\}.$$

The mapping $O : \mathcal{P} \rightarrow I\!P(LA)$ is for $\otimes \in \{\mp, \circ^S, \|^S \mid S \subseteq LA \times IN \times LA\}$, $\odot \in \{^*, ^\infty, ^{[m,n]}\} \mid m, n \in IN \cup \{\infty\}\}$, $a \in A$, $e \in \mathcal{L}(\langle\text{event name}\rangle)$, $O \subseteq \mathcal{AOR}$ and $x, y \in \mathcal{P}$ defined as follows:

$$\begin{array}{lcl}
O(\varepsilon) & = & \varnothing, \\
O(\delta) & = & \varnothing, \\
O(a_e^O) & = & \begin{cases} \{a_e^O\} & \text{if } O \neq \varnothing, \\ \varnothing & \text{otherwise,} \end{cases} \\
O(a_e) & = & \varnothing, \\
O(a) & = & \varnothing, \\
O(x \otimes y) & = & O(x) \cup O(y), \\
O(x^{\odot}) & = & O(x), \\
O(\rho_f(x)) & = & \{f(a) \mid a \in O(x)\}, \\
O(\langle X \mid E \rangle) & = & \langle X_O \mid E_O \rangle,
\end{array}$$

where E_O is defined as follows:

$$E_O = \{X_O = O(s_X) \mid X = s_X \in E\}.$$

Observe that $M(x) = \alpha(x) \cap LA_{msg}$. Note that this only gives the desired result if the renaming mapping f is such that for all $a \in LA$: $a \in LA_{msg}$ if and only if $f(a) \in LA_{msg}$.

If two MSC fragments are composed vertically or horizontally it can be the case that one of them contains a message output event and the other a corresponding message input event. In that case the ordering requirement that the message output event precedes the message input event must be taken into account. This is achieved by finding the pairs of atomic actions that refer to a message output or input event (using the mapping M and the relation $\circ\!\!\rightarrow\!\!\circ$). Such a pair then gives rise to an ordering requirement. Similarly if the MSC fragments contain corresponding ordered events this also gives rise to an ordering requirement. The mappings $MsgReq$ and $OrdReq$ are used to obtain the ordering requirements that must be taken into acount when two MSC fragments are composed due to the requirement that an output precedes the corresponding input and due to causal order relations between orderable events. The mapping $CrReq$ is used to enforce the ordering of all events on a created instance after the execution of the create event.

Definition B.6.6.3 The mapping $MsgReq : \mathcal{P} \times \mathcal{P} \rightarrow I\!P(LA_{out} \times LA_{in})$ is for $x, y \in \mathcal{P}$ defined as follows:

$$\begin{array}{rcl} MsgReq(x,y) & = & \{o \mapsto i \mid o \circ\!\!\rightarrow\!\!\circ i \wedge o \in M(x) \wedge i \in M(y)\} \\ & \cup & \{o \mapsto i \mid o \circ\!\!\rightarrow\!\!\circ i \wedge o \in M(y) \wedge i \in M(x)\}. \end{array}$$

The mapping $OrdReq : \mathcal{P} \times \mathcal{P} \rightarrow I\!P(LA \times LA)$ is for $x, y \in \mathcal{P}$ defined as follows:

$$\begin{array}{rcl} OrdReq(x,y) & = & \{s \mapsto d \mid s \circ\!\!\rightarrow\!\!-\!\!\circ d \wedge s \in O(x) \wedge d \in O(y)\} \\ & \cup & \{s \mapsto d \mid s \circ\!\!\rightarrow\!\!-\!\!\circ d \wedge s \in O(y) \wedge d \in O(x)\}. \end{array}$$

The mapping $CrReq : \mathcal{P} \times \mathcal{P} \rightarrow I\!P(LA \times LA)$ is for $x, y \in \mathcal{P}$ defined as follows:

$$\begin{array}{rcl} CrReq(x,y) & = & \{c \mapsto a \mid c \hookrightarrow a \wedge c \in \alpha(x) \wedge a \in \alpha(y)\} \\ & \cup & \{c \mapsto a \mid c \hookrightarrow a \wedge c \in \alpha(y) \wedge a \in \alpha(x)\}. \end{array}$$

If the connection of a message output event and a message input event is established via a gate it is necessary to change the atomic actions in such a way that the atomic action for the message output event is updated with the receiver instance name and the atomic action for the message input event is updated with the sender instance name. Before the connection was established these names were not known and therefore indicated by _. Given two processes x and y the mapping $f(x,y)$ associates with every atomic action a possibly renamed atomic action. Note that for output events this renaming only applies to the receiver instance part and for input events only to the sender instance part.

Definition B.6.6.4 Let $x, y \in \mathcal{P}$. Then, the mapping $f(x,y) : LA \rightarrow LA$ is for $i, j \in \mathcal{L}(\langle\text{instance name}\rangle)$, $m \in \mathcal{L}(\langle\text{message name}\rangle)$, $G \in \mathcal{AMG}$, $S \subseteq \mathcal{AOR}$, $a \in LA$ and $e \in \mathcal{L}(\langle\text{event name}\rangle)$ defined as follows:

$$\begin{array}{lcll} f(x,y)(out(i,G,\_,m)^S_e) & = & out(i,G,j,m)^S_e & \text{if } out(i,G,\_,m)^S_e \in M(x) \wedge in(\_,G,j,m)^{S'}_{e'} \in M(y) \\ & & & \text{or } out(i,G,\_,m)^S_e \in M(y) \wedge in(\_,G,j,m)^{S'}_{e'} \in M(x), \\ f(x,y)(in(\_,G,j,m)^S_e) & = & in(i,G,j,m)^S_e & \text{if } out(i,G,\_,m)^{S'}_{e'} \in M(x) \wedge in(\_,G,j,m)^S_e \in M(y) \\ & & & \text{or } out(i,G,\_,m)^{S'}_{e'} \in M(y) \wedge in(\_,G,j,m)^S_e \in M(x), \\ f(x,y)(a) & = & a & \text{otherwise.} \end{array}$$

Note that the $f(x,y)$ and the other renaming mappings used in this section are not bijective by definition. However, in all situations that can occur they are bijective due to the severe uniqueness requirements on MSC.

Definition B.6.6.5 For $x, y \in \mathcal{P}$

$$\begin{array}{rcl} x \bullet y & = & \rho_{f(x,y)}(x \circ^{MsgReq(x,y) \cup OrdReq(x,y) \cup CrReq(x,y)} y), \\ x \blacksquare y & = & \rho_{f(x,y)}(x \parallel^{MsgReq(x,y) \cup OrdReq(x,y) \cup CrReq(x,y)} y). \end{array}$$

B.6.7 Semantics of coregions

A coregion contains a number (possibly zero) of orderable events. These events are defined on the same instance, but are nevertheless not ordered for that reason. It is however possible that a coregion contains both the output and the input of a message or both events involved in a general ordering. The semantics of a coregion is thus the horizontal composition of the semantics of its events.

Definition B.6.7.1 Let $i \in \mathcal{L}(\langle\text{instance name}\rangle)$. Then, for $e \in \mathcal{L}(\langle\text{orderable event}\rangle)$ and $coevents \in \mathcal{L}(\langle\text{coevent list}\rangle)$

$$\begin{array}{lcl} [\![\textbf{concurrent; endconcurrent}]\!]_i & = & \varepsilon, \\ [\![\textbf{concurrent; } e \textbf{ ; } coevents \textbf{ endconcurrent}]\!]_i & = & [\![e]\!]_i \blacksquare [\![\textbf{concurrent; } coevents \textbf{ endconcurrent}]\!]_i. \end{array}$$

Example B.6.7.2 Consider a coregion on instance i which contains the input of message m, a local action a and the output of message m. Then, the semantics of this coregion is given by

$$out(i, \_, i, m) \parallel^{R_2} (action(i, a) \parallel^{R_1} out(i, \_, i, m)),$$

where $R_1 = \varnothing$ and $R_2 = \{out(i, \_, i, m) \mapsto in(i, \_, i, m)\}$.

B.6.8 Semantics of MSC bodies

An MSC body is a possibly empty list of event definitions. As explained before such a list of event definitions is interpreted as a list of MSC fragments that are composed vertically.

Definition B.6.8.1 For $eventdef \in \mathcal{L}(\langle\text{event definition}\rangle)$ and $mscbody \in \mathcal{L}(\langle\text{mscbody}\rangle)$

$$\begin{array}{lcl} [\![]\!] & = & \varepsilon, \\ [\![eventdef\ mscbody]\!] & = & [\![eventdef]\!] \bullet [\![mscbody]\!]. \end{array}$$

In composing an event definition with an MSC body it can be the case that gates are connected.

There are two types of event definitions that are considered in this section: single instance events and multi-instance events. Single instance events are instance events that are defined on one instance. In order to associate an atomic action with the defining instance as a parameter to these single instance events the defining instance(s) are determined and the semantic mapping is labeled with it. Multi instance events are events that are defined on a non-empty set of instances. There is no use for labeling the semantic mapping with these instances as in any relevant case the instances appear again in the textual description of the multi instance event.

Definition B.6.8.2
For $i \in \mathcal{L}(\langle\text{instance name}\rangle)$, $ilist \in \mathcal{L}(\langle\text{instance name list}\rangle)$, $instanceevent \in \mathcal{L}(\langle\text{instance event}\rangle)$ and $multiinstanceevent \in \mathcal{L}(\langle\text{multi instance event}\rangle)$

$$\begin{array}{lcl} [\![i \textbf{ : } instancevent\textbf{;}]\!] & = & [\![instanceevent]\!]_i, \\ [\![ilist \textbf{ : } multiinstanceevent\textbf{;}]\!] & = & [\![multiinstanceevent]\!]. \end{array}$$

Example B.6.8.3 (Simple communication) Consider the MSC A with two instances i and j and one message m from instance i to instance j. There are two event-oriented textual representations for this MSC:

msc A;
i : **out** m **to** j;
j : **in** m **from** i;
endmsc;

and

msc A;
j : **in** m **from** i;
i : **out** m **to** j;
endmsc;

Using the first textual representation the recursive equation

$$\overline{A} = out(i, \_, j, m) \circ^{out(i,\_,j,m) \mapsto in(i,\_,j,m)} in(i, \_, j, m)$$

is obtained and using the second textual representation the recursive equation

$$\overline{A} = in(i, \_, j, m) \circ^{out(i,\_,j,m) \mapsto in(i,\_,j,m)} out(i, \_, j, m)$$

is obtained. The semantics of the MSC is in both cases given by $\overline{A}$. Operationally the first can be depicted as

$$\overline{A} \xrightarrow{out(i,\_,j,m)} \varepsilon \circ^{out(i,\_,j,m) \overset{1}{\mapsto} in(i,\_,j,m)} in(i, \_, j, m) \xrightarrow{in(i,\_,j,m)} \varepsilon \circ^{out(i,\_,j,m) \mapsto in(i,\_,j,m)} \varepsilon \longrightarrow$$

and the second as

$$\overline{A} \xrightarrow{out(i,\_,j,m)} in(i, \_, j, m) \circ^{out(i,\_,j,m) \overset{1}{\mapsto} in(i,\_,j,m)} \varepsilon \xrightarrow{in(i,\_,j,m)} \varepsilon \circ^{out(i,\_,j,m) \mapsto in(i,\_,j,m)} \varepsilon \longrightarrow$$

Observe that in both cases the same traces can be performed.

Example B.6.8.4 (General ordering) Consider the MSC from Figure B.50. Suppose that this MSC is tex-

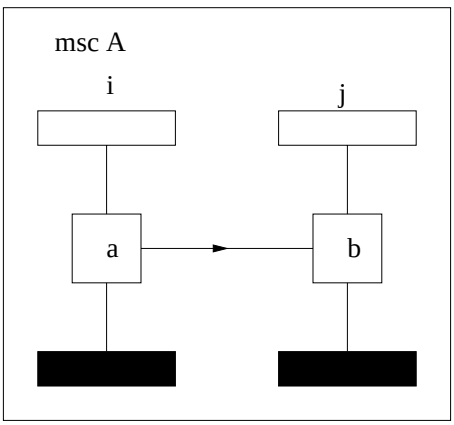


Figure B.50: MSC with a general ordering

tually represented by

msc A;
i : l1 **action** a **before** l2**;**
j : l2 **action** b **after** l1**;**
endmsc;

This MSC consists of two MSC fragments. These fragments are semantically represented by

$$action(i, a)_{l1}^{\{l1 \mapsto l2\}}$$

and

$$action(j, b)_{l2}^{\{l1 \mapsto l2\}}.$$

Observe that

$$O(action(i, a)_{l1}^{\{l1 \mapsto l2\}}) = \{action(i, a)_{l1}^{\{l1 \mapsto l_2\}}\}$$

and

$$O(action(j,b)_{l2}^{\{l1\mapsto l2\}}) = \{action(j,b)_{l2}^{\{l1\mapsto l_2\}}\}$$

Then, the following set of ordering requirements is obtained:

$$R = \{action(i,a)_{l1}^{\{l1\mapsto l2\}} \mapsto action(j,b)_{l2}^{\{l1\mapsto l2\}}\}$$

Thus, the expression representing the semantics of the MSC, is the following:

$$action(i,a)_{l1}^{\{l1\mapsto l2\}} \circ^R action(j,b)_{l2}^{\{l1\mapsto l2\}}$$

B.6.9 Semantics of MSC reference expressions

Textually an MSC reference expression consists of a textual formula containing MSC names and operators, a MSC reference identification and a reference gate interface. The semantics of the textual formula itself is rather easy as a semantical equivalent has been defined for each of the composition operators that can occur in this formula.

Definition B.6.9.1
For $mscrefexpr \in \mathcal{L}(\langle\text{msc ref expr}\rangle)$, $par \in \mathcal{L}(\langle\text{msc ref par expr}\rangle)$, $seq \in \mathcal{L}(\langle\text{msc ref seq expr}\rangle)$, $loop \in \mathcal{L}(\langle\text{msc ref loop expr}\rangle)$, $b \in \mathcal{L}(\langle\text{expr body}\rangle)$, $m, n \in IN \cup \{\infty\}$ and $mscname \in \mathcal{L}(\langle\text{msc name}\rangle)$,

$$\begin{array}{lcl}
[\![par \textbf{ alt } mscrefexpr]\!] & = & [\![par]\!] \mp [\![mscrefexpr]\!], \\
[\![seq \textbf{ par } par]\!] & = & [\![seq]\!] \parallel [\![par]\!], \\
[\![loop \textbf{ seq } seq]\!] & = & [\![loop]\!] \circ [\![seq]\!], \\
[\![\textbf{loop } \langle m\textbf{,}n\rangle \ b]\!] & = & [\![b]\!]^{[m,n]}, \\
 & & \\
[\![\textbf{empty}]\!] & = & \varepsilon, \\
[\![mscname]\!] & = & \overline{mscname}, \\
[\![\textbf{(}mscrefexpr\textbf{)}]\!] & = & [\![mscrefexpr]\!].
\end{array}$$

Example B.6.9.2 The semantics of the textual formula

reference (A **alt empty**) **par** B **seq** C

is given by the process $(\overline{A} \mp \varepsilon) \parallel \overline{B} \circ \overline{C}$.

Example B.6.9.3 The semantics of the textual formula

reference loop <5,3> A **seq** B

is given by the process $(\overline{A})^{[5,3]} \circ \overline{B}$ which cannot perform any events from MSC *A*.

If gates of an MSC reference expression are connected on the outside, the gate definitions in the MSCs referenced by the textual formula become actual gates. The semantics of the textual formula contains these gate definitions as the via part of message output events and message input events and as labels of the orderings with which atomic actions can be labelled. For message gates three different types of connection can exist.

1) A gate can be connected to the environment of the enclosing MSC fragment.

2) A gate can be connected to an instance of the enclosing MSC fragment.

3) A gate can be connected to an MSC reference expression or inline expression of the enclosing MSC fragment.

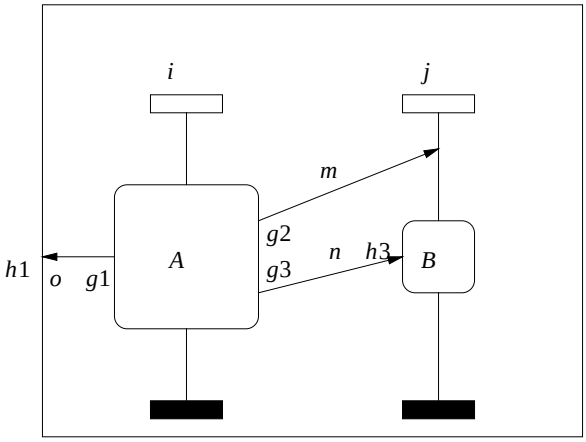


Figure B.51: Three different situations

These three situations are depicted in Figure B.51 for the MSC reference expression A.

In the semantics of the textual formula A the gates $g1$, $g2$ and $g3$ appear as $env(g1)$, $env(g2)$ and $env(g3)$ respectively. In the context of the MSC which contains this MSC reference expression these gates are not necessarily connections to the environment anymore. Only gate $g1$ is connected to the environment (again). In order to indicate this situation we replace all occurrences of $env(g1)$, $env(g2)$ and $env(g3)$ by more appropriate and convenient gate names. This renaming is based on the information that is available in the reference gate interface.

1) The gate with name $g1$ is connected externally to the environment via a gate with name $h1$. Therefore, all occurrences of $env(g1)$ in the semantics of A are replaced by $env(h1)$.

2) The gate with name $g2$ is connected externally to instance j by means of a message arrow. The intuition is that the output of message m in A is received by instance j. Thus, this communication will become internal. This is part of the reason why $env(g2)$ is replaced by $(l, g2)$. Another reason is that we must be able to distinguish the actual gates of references to an MSC in different MSC reference expressions. As the MSC reference identification is unique the combination of the MSC reference identification and the gate name is a nice name for the conceptual gate. Looking at the semantics of the message input event on instance j we find that it also has a via part $(l, g2)$. So additionally, but on purpose, we have created the situation in which we can establish which output and input event together make one communication.

3) For similar reasons the occurrences of $env(g3)$ are replaced by $((l, g3), (l', h3))$ where l' is the MSC reference identification of the MSC reference expression on instance j. The occurrences of $env(h3)$ in the semantics of this second MSC reference expression are also replaced by $((l, g3), (l', h3))$. This again, gives us a nice way to establish correspondence of the message output event and the message input event.

The information needed for the renamings discussed above is available in the reference gate interface. For the example MSC from Figure B.51 it contains the entries: "**gate** g1 **out** o **to env via** h1", "**gate** g2 **out** m **to** j" and "**gate** g3 **out** m **to reference** l' **via** h3".

The mapping G that is defined in the following definition abstracts from the textual representation of the reference gate interface and turns it into a set of pairs of connections. The gates of the MSC reference expression are indicated by a pair consisting of the MSC reference identification and the original gate name. A connection with a gate h in the environment is indicated by $env(h)$, a connection with an instance j by j and a connection with an MSC reference expression or an inline expression by its identification and the gate used on it. The pairs are ordered such that an arrow is drawn from the first 'gate' to the second 'gate'.

Definition B.6.9.4 (Abstract Message Address and Abstract Gate Interface) The set $\mathcal{AMA}$ is defined as follows:

$$\begin{array}{lcl} \mathcal{AMA} & = & \mathcal{L}(\langle\text{instance name}\rangle) \\ & \cup & \{env(g) \mid g \in \mathcal{L}(\langle\text{gate name}\rangle)\} \\ & \cup & \mathcal{L}(\langle\text{reference identification}\rangle) \times \mathcal{L}(\langle\text{gate name}\rangle \\ & \cup & (\mathcal{L}(\langle\text{ref name}\rangle) \times \mathcal{L}(\langle\text{gate name}\rangle))^2. \end{array}$$

The set $\mathcal{AGI}$ is defined as follows:

$$\mathcal{AGI} = I\!P((\mathcal{AMA} \cup \mathcal{AOD}) \times (\mathcal{AMA} \cup \mathcal{AOD}))$$

The mapping S associates with an output or input address of a message event an abstract message address.

Definition B.6.9.5
The mapping $S : \mathcal{L}(\langle\text{output address}\rangle) \cup \mathcal{L}(\langle\text{input address}\rangle) \to \mathcal{AMA}$ is for $i \in \mathcal{L}(\langle\text{instance name}\rangle)$, $g \in \mathcal{L}(\langle\text{gate name}\rangle)$ and $l \in \mathcal{L}(\langle\text{ref name}\rangle)$ defined by

$$\begin{array}{lcl} S(i) & = & i \\ S(\textbf{env via } g) & = & env(g) \\ S(\textbf{reference } l \textbf{ via } g) & = & (l, g) \\ S(\textbf{inline } l \textbf{ via } g) & = & (l, g) \end{array}$$

The mapping G associates with a reference gate interface an abstract gate interface.

Definition B.6.9.6 Let $l \in \mathcal{L}(\langle\text{ref name}\rangle)$. The mapping $G_l : \mathcal{L}(\langle\text{reference gate interface}\rangle) \to \mathcal{AGI}$ is for $refgate \in \mathcal{L}(\langle\text{ref gate}\rangle)$ and $gates \in \mathcal{L}(\langle\text{reference gate interface}\rangle)$ defined inductively by

$$\begin{array}{lcl} G_l() & = & \varnothing \\ G_l(\textbf{; gate } refgate\ gates) & = & \{G_l(refgate)\} \cup G_l(gates) \end{array}$$

Let $l \in \mathcal{L}(\langle\text{ref name}\rangle)$. The mapping $G_l : \mathcal{L}(\langle\text{ref gate}\rangle) \to \mathcal{AGI}$ is for $g \in \mathcal{L}(\langle\text{gate name}\rangle)$, $m \in \mathcal{L}(\langle\text{message name}\rangle)$, $a \in \mathcal{L}(\langle\text{output address}\rangle) \cup \mathcal{L}(\langle\text{input address}\rangle)$ and $d \in \mathcal{L}(\langle\text{order dest}\rangle)$ defined by

$$\begin{array}{lcl} G_l(g \textbf{ out } m \textbf{ to } a) & = & ((l, g), S(a)) \\ G_l(g \textbf{ in } m \textbf{ from } a) & = & (S(a), (l, g)) \\ G_l(g \textbf{ before } d) & = & ((l, g), S(d)) \\ G_l(g \textbf{ after } d) & = & (S(d), (l, g)) \end{array}$$

In the following definition a mapping *via* is defined. This mapping implements the renaming of the gate definitions of the referenced MSCs into actual gates following the lines explained before. For *via* to be well-defined it is necessary that there are no two gate definitions with the same gate name, not even if they have another direction. It is also necessary that there are no two different external connections for a given gate g on the MSC reference expression. The mapping *via* is extended to the ordering sets with which the atomic actions are labelled and to the atomic actions in the obvious way.

Definition B.6.9.7 Let $l \in \mathcal{L}(\langle\text{ref name}\rangle$ and let $gates \subseteq \mathcal{AGI}$. The mapping $via(l, gates) : (\mathcal{AMG} \cup \mathcal{AOD}) \to (\mathcal{AMG} \cup \mathcal{AOD})$ is for $g \in \mathcal{L}(\langle\text{gate name}\rangle)$ and $G \in \mathcal{AMG} \cup \mathcal{AOD}$ defined as follows:

$$via(l, gates)(env(g)) = \begin{cases} env(h) & \text{if } ((l, g), env(h)) \in gates \\ & \text{or } (env(h), (l, g)) \in gates \\ (l, g) & \text{if } ((l, g), j) \in gates \\ & \text{or } (i, (l, g)) \in gates \\ ((l, g), (l', g')) & \text{if } ((l, g), (l', g')) \in gates \\ ((l', g'), (l, g)) & \text{if } ((l', g'), (l, g)) \in gates \\ env(g) & \text{otherwise} \end{cases}$$

$$via(l, gates)(G) = G \quad \text{otherwise}$$

Let $l \in \mathcal{L}(\langle\text{ref name}\rangle$ and let $gates \subseteq \mathcal{AGI}$. The mapping $via(l, gates) : LA \to LA$ is for $i, j \in \mathcal{L}(\langle\text{instance name}\rangle)$, $g \in \mathcal{L}(\langle\text{gate name}\rangle)$, $m \in \mathcal{L}(\langle\text{message name}\rangle)$, $O \subseteq \mathcal{AOD} \times \mathcal{AOD}$, $a \in A$ and $e \in \mathcal{L}(\langle\text{event name}\rangle)$ defined as follows:

$$\begin{array}{lcl} via(l, gates)(out(i, env(g), j, m)_e^O) & = & out(i, via(l,gates)(env(g)), j, m)_e^{via(l,gates)(O)} \\ via(l, gates)(in(i, env(g), j, m)_e^O) & = & in(i, via(l,gates)(env(g)), j, m)_e^{via(l,gates)(O)} \\ via(l, gates)(a_e^O) & = & a_e^{via(l,gates)(O)} \text{otherwise} \end{array}$$

Let $l \in \mathcal{L}(\langle\text{ref name}\rangle$ and let $gates \subseteq \mathcal{AGI}$. The mapping $via(l, gates) : I\!P((\mathcal{AMG} \cup \mathcal{AOD}) \times (\mathcal{AMG} \cup \mathcal{AOD})) \to I\!P((\mathcal{AMG} \cup \mathcal{AOD}) \times (\mathcal{AMG} \cup \mathcal{AOD}))$ is for $O \subseteq \mathcal{AOR}$ defined as follows:

$$via(l, gates)(O) \quad = \quad \{(via(l, gates)(g_1), via(l, gates)(g_2)) \mid \exists_{g_1, g_2 \in \mathcal{AOD}}(g_1, g_2) \in O\}$$

Using the above definitions the semantics of the MSC reference expression can be described by

$$\rho_{via(l, G_l(gates))}([\![mscrefexpr]\!]_I)$$

where l is the MSC reference identification, $gates$ is the reference gate interface and $mscrefexpr$ is the textual formula. However, it is possible that two gates of the MSC reference expression are connected. Therefore, an ordering requirement must be added to the semantics and, if this is a connection between message gates, atomic actions have to be renamed. The mapping $g(x)$ defined below gives the necessary renaming and the mapping $R(x)$ defines the (not yet renamed) ordering requirements. The process x represents the semantics of the MSC reference expression including the renaming according to $via(l, G_l(gates))$.

Definition B.6.9.8 The mapping $R : \mathcal{P} \to I\!P(LA \times LA)$ is for $x \in \mathcal{P}$ defined as follows:

$$\begin{array}{lcl} R(x) & = & \{o \mapsto i \mid o \circ\!\!\to \circ i \wedge o, i \in M(x)\} \\ & \cup & \{s \mapsto d \mid s \circ\!\!\to\!\!-\circ d \wedge s, d \in M(x)\} \end{array}$$

Definition B.6.9.9 Let $x \in \mathcal{P}$. The mapping $g(x) : LA \to LA$ is for $i, j \in \mathcal{L}(\langle\text{instance name}\rangle)$, $m \in \mathcal{L}(\langle\text{message name}\rangle)$, $l \in \mathcal{L}(\langle\text{ref name}\rangle)$, $g, h \in \mathcal{L}(\langle\text{gate name}\rangle)$, $O \subseteq \mathcal{AOR}$, $e \in \mathcal{L}(\langle\text{event name}\rangle)$ and $a \in LA$ defined by:

$$\begin{array}{lcl} g(x)(out(i, ((l,g),(l,h)), \_, m)_e^O) & = & out(i, ((l,g),(l,h)), j, m)_e^O \\ & & \text{if } out(i, ((l,g),(l,h)), \_, m)_e^O, in(\_, ((l,g),(l,h)), j, m)_{e'}^{O'} \in M(x) \\ g(x)(in(\_, ((l,g),(l,h)), j, m)_e^O) & = & in(i, ((l,g),(l,h)), j, m)_e^O \\ & & \text{if } out(i, ((l,g),(l,h)), \_, m)_{e'}^{O'} in(\_, ((l,g),(l,h)), j, m)_e^O \in M(x) \\ g(x)(a) & = & a \quad \text{otherwise} \end{array}$$

Definition B.6.9.10
For $l \in \mathcal{L}(\langle\text{ref name}\rangle$, $mscrefexpr \in \mathcal{L}(\langle\text{msc ref expr}\rangle)$ and $gates \in \mathcal{L}(\langle\text{reference gate interface}\rangle)$

$$[\![\textbf{reference } l : mscrefexpr\ gates]\!] \quad = \quad \rho_g(\rho_v([\![mscrefexpr]\!]) \circ^R \varepsilon),$$

where $v = via(l, G_l(gates))$, $g = g(\rho_v([\![mscrefexpr]\!]))$ and $R = R(\rho_v([\![mscrefexpr]\!]))$.

B.6.10 Semantics of inline expressions

The semantics of inline expressions is easily obtained from the semantics of the arguments of an inline expression by combining them by means of the semantical equivalent of the operation indicated in the inline expression. The operation indicated with the keyword **alt** is interpreted by the operator delayed choice $\mp$, the operation indicated by **par** is interpreted as delayed parallel composition $\|$ and the operation **loop**<m,n> by the operator $^{[m,n]}$.

Define the gates of an inline expression and the gates of an inline expression that connect gates from the inline expression with gates from the inline expression.

Definition B.6.10.1 (External connections of inline expressions) Let $l \in \mathcal{L}(\langle\text{ref name}\rangle)$. The mapping $G_l : \mathcal{L}(\langle\text{alt list}\rangle) \to \mathcal{AGI}$ is for $gates \in \mathcal{L}(\langle\text{inline gate interface}\rangle)$, $b \in \mathcal{L}(\langle\text{msc body}\rangle)$ and $altlist \in \mathcal{L}(\langle\text{alt list}\rangle)$ defined inductively by:

$$\begin{array}{lcl} G_l(gates\ b) & = & G_l(gates), \\ G_l(gates\ b\ \textbf{alt ;}\ altlist) & = & G_l(gates) \cup G_l(altlist). \end{array}$$

Let $l \in \mathcal{L}(\langle\text{ref name}\rangle)$. The mapping $G_l : \mathcal{L}(\langle\text{par list}\rangle) \to \mathcal{AGI}$ is for $gates \in \mathcal{L}(\langle\text{inline gate interface}\rangle)$, $b \in \mathcal{L}(\langle\text{msc body}\rangle)$ and $parlist \in \mathcal{L}(\langle\text{par list}\rangle)$ defined inductively by:

$$\begin{array}{lcl} G_l(gates\ b) & = & G_l(gates), \\ G_l(gates\ b\ \textbf{par ;}\ parlist) & = & G_l(gates) \cup G_l(parlist). \end{array}$$

Let $l \in \mathcal{L}(\langle\text{ref name}\rangle)$. The mapping $G_l : \mathcal{L}(\langle\text{inline gate interface}\rangle) \to \mathcal{AGI}$ is for $igate \in \mathcal{L}(\langle\text{inline gate}\rangle)$ and $gates \in \mathcal{L}(\langle\text{inline gate interface}\rangle)$ defined as follows:

$$\begin{array}{lcl} G_l() & = & \varnothing, \\ G_l(\textbf{gate}\ igate\ \textbf{;}\ gates) & = & G_l(inlinegate) \cup G_l(gates). \end{array}$$

Let $l \in \mathcal{L}(\langle\text{ref name}\rangle)$. The mapping $G_l : \mathcal{L}(\langle\text{inline gate}\rangle) \to \mathcal{AGI}$ is for $g \in \mathcal{L}(\langle\text{gate name}\rangle)$, $m, m' \in \mathcal{L}(\langle\text{message name}\rangle)$, $s \in \mathcal{L}(\langle\text{output address}\rangle)$, $d \in \mathcal{L}(\langle\text{input address}\rangle)$ and $o, o' \in \mathcal{L}(\langle\text{order dest}\rangle)$ defined as follows:

$$\begin{array}{lcl} G_l(g\ \textbf{in}\ m\ \textbf{from}\ s\ \textbf{external out}\ m'\ \textbf{to}\ d) & = & \{((l, g), S(d))\}, \\ G_l(g\ \textbf{out}\ m\ \textbf{to}\ d\ \textbf{external in}\ m'\ \textbf{from}\ s) & = & \{(S(s), (l, g))\}, \\ G_l(g\ \textbf{after}\ o\ \textbf{external before}\ o') & = & \{((l, g), S(o'))\}, \\ G_l(g\ \textbf{before}\ o\ \textbf{external after}\ o') & = & \{(S(o'), (l, g))\}. \end{array}$$

Please note that the recommendation allows the use of different message names in the internal and external connection of a gate. A static requirement that forbids this should be defined.

Definition B.6.10.2 (Inline loop expression)
For $m, n \in IN \cup \{\infty\}$, $l \in \mathcal{L}(\langle\text{ref name}\rangle$, $gates \in \mathcal{L}(\langle\text{inline gate interface}\rangle)$ and $b \in \mathcal{L}(\langle\text{msc body}\rangle)$

$$[\![\textbf{loop}\ \langle m,n\rangle\ \textbf{begin}\ l\ \textbf{;}\ gates\ b\ \textbf{loop end}]\!] \quad = \quad \rho_g(\rho_v([\![b]\!]^{[m,n]}) \circ^R \varepsilon),$$

where $v = via(l, G_l(gates))$, $g = g(\rho_v([\![b]\!]^{[m,n]}))$ and $R = R(\rho_v([\![b]\!]^{[m,n]}))$.

Definition B.6.10.3 (Inline alternative expression) For $l \in \mathcal{L}(\langle\text{ref name}\rangle)$, $altlist \in \mathcal{L}(\langle\text{alt list}\rangle)$, $gates \in \mathcal{L}(\langle\text{inline gate interface}\rangle)$ and $b \in \mathcal{L}(\langle\text{msc body}\rangle)$

$$[\![\textbf{alt begin}\ l\ \textbf{;}\ altlist\ \textbf{alt end}]\!] \quad = \quad \rho_g(\rho_v([\![altlist]\!]) \circ^R \varepsilon),$$

$$\begin{array}{lcl} [\![gates\ b]\!] & = & [\![b]\!], \\ [\![gates\ b\ \textbf{alt ;}\ altlist]\!] & = & [\![b]\!] \mp [\![altlist]\!], \end{array}$$

where $v = via(l, G_l(altlist))$, $g = g(\rho_v([\![altlist]\!]))$ and $R = R(\rho_v([\![altlist]\!]))$.

Definition B.6.10.4 (Inline parallel expression) For $l \in \mathcal{L}(\langle\text{ref name}\rangle)$, $parlist \in \mathcal{L}(\langle\text{par list}\rangle)$, $gates \in \mathcal{L}(\langle\text{inline gate interface}\rangle)$ and $b \in \mathcal{L}(\langle\text{msc body}\rangle)$

$$[\![\textbf{par begin}\ l\ \textbf{;}\ parlist\ \textbf{par end}]\!] \quad = \quad \rho_g(\rho_v([\![parlist]\!]) \circ^R \varepsilon),$$

$$\begin{array}{lcl} [\![gates\ b]\!] & = & [\![b]\!], \\ [\![gates\ b\ \textbf{par ;}\ parlist]\!] & = & [\![b]\!] \parallel [\![parlist]\!], \end{array}$$

where $v = via(l, G_l(parlist))$, $g = g(\rho_v([\![parlist]\!]))$ and $R = R(\rho_v([\![parlist]\!]))$.

B.6.11 Semantics of High-level Message Sequence Charts

Textually an HMSC is described by associating a label with every node except the start node. The start node is describes first in the textual syntax by simply listing its successor nodes in a label name list. Then all other nodes are described. Such a description consists of the label name associated with the node followed by a description of the type of the node and a label name list representing the label names of the successor nodes.

If a node has successor nodes then these are interpreted as alternative vertical compositions. For example if a node labeled l has two successor nodes labeled l_1 and l_2 this means that the node l is vertically composed with either node l_1 or l_2.

Semantically, HMSCs are dealt with by associating a recursion variable $\overline{l}$ with a node labeled l. Since the start node of an HMSC does not have a label the name of the HMSC is used as a recursion variable for this node. For each of the recursion variables introduced in this way a recursive equation is determined as follows. The recursive equation associated with a node labeled l is of the form $\overline{l} = C \circ (\overline{l_1} \mp \cdots \mp \overline{l_N})$ where C represents the semantics of the node with label l and $l_1, \cdots l_N$ are the labels associated with the successor nodes of the node labeled with l. The recursive equation associated with the recursion variable introduced for the start node, say $\overline{mscname}$, is $\overline{mscname} = \overline{l_1} \mp \cdots \mp \overline{l_N}$ where $l_1, \cdots, l_N$ are the labels of the successor nodes of the start node.

The set of all recursive equations of an HMSC is obtained by applying the mapping Eqs given in Definition B.6.11.1. The mappings $Start$ and Suc used in this definition are given in Definition B.6.11.2.

Definition B.6.11.1 Then, for $mscname \in \mathcal{L}(\langle\text{msc name}\rangle)$, $mscexpr \in \mathcal{L}(\langle\text{msc expression}\rangle)$, $labellist \in \mathcal{L}(\langle\text{label name list}\rangle)$, $nodeexprlist \in \mathcal{L}(\langle\text{node expression list}\rangle)$, $nodeexpr \in \mathcal{L}(\langle\text{node expression}\rangle)$, $l \in \mathcal{L}(\langle\text{label name}\rangle)$, $node \in \mathcal{L}(\langle\text{node}\rangle)$, $mscrefexpr \in \mathcal{L}(\langle\text{msc ref expr}\rangle)$, $parexpr \in \mathcal{L}(\langle\text{par expression}\rangle)$ and $clist \in \mathcal{L}(\langle\text{condition name list}\rangle)$

$$
\begin{array}{lcl}
Eqs(\textbf{msc}\ mscname\textbf{;}\ \textbf{expr}\ mscexpr\ \textbf{endmsc;}) & = & \{\overline{mscname} = Start(mscexpr)\} \cup Eqs(mscexpr) \\
\\
Eqs(labellist\textbf{;}\ nodeexprlist) & = & Eqs(nodeexprlist) \\
\\
Eqs() & = & \varnothing, \\
Eqs(nodeexpr\ nodeexprlist) & = & Eqs(nodeexpr) \cup Eqs(modeexprlist), \\
\\
Eqs(l\textbf{:}\ \textbf{end;}) & = & \{\overline{l} = \varepsilon\} \\
Eqs(l\textbf{:}\ node\ \textbf{seq (}labellist\textbf{);}) & = & \{\overline{l} = [\![node]\!] \circ Suc(labellist)\} \cup Eqs(node) \\
\\
Eqs(\textbf{empty}) & = & \varnothing \\
Eqs(mscname) & = & \varnothing \\
Eqs(parexpr) & = & Eqs(parexpr) \\
Eqs(\textbf{condition}\ clist) & = & \varnothing \\
Eqs(\textbf{connect}) & = & \varnothing \\
Eqs(\textbf{(}mscrefexpr\textbf{)}) & = & \varnothing \\
\\
Eqs(\textbf{expr}\ mscexpr\ \textbf{endexpr}) & = & Eqs(mscexpr), \\
Eqs(\textbf{expr}\ mscexpr\ \textbf{endexpr par}\ parexpr) & = & Eqs(mscexpr) \cup Eqs(parexprlist).
\end{array}
$$

The semantics of a node in an HMSC depends on the type of node. Start nodes, condition nodes, connector nodes and end nodes do not describe the execution of events. Therefore their semantics is given by the empty process ε. An MSC reference node describes the composition of a number of MSCs by means of a textual formula and a parallel frame node describes the parallel composition of a number of sub-HMSCs. The semantics of one such sub-HMSC is given by the delayed choice of the recursion variables associated with the successor nodes of the start node of the sub-HMSC. This is necessary as the start node does not have a name. For this purpose the mapping $Start$ is given in Definition B.6.11.2.

Definition B.6.11.2 The mapping $Start : \mathcal{L}(\langle\text{msc expression}\rangle) \to \mathcal{P}$ is, for $list \in \mathcal{L}(\langle\text{label name list}\rangle)$ and $nodeexprlist \in \mathcal{L}(\langle\text{node expression list}\rangle)$, defined by

$$Start(list\,\mathbf{;}\,nodeexprlist) \quad = \quad Suc(list)$$

and the mapping $Suc : \mathcal{L}(\langle\text{label name list}\rangle) \to \mathcal{P}$ is, for $l \in \mathcal{L}(\langle\text{label name}\rangle)$ and $list \in \mathcal{L}(\langle\text{label name list}\rangle)$ defined by

$$\begin{array}{lcl} Suc(l) & = & \overline{l}, \\ Suc(l\ \mathbf{alt}\ list) & = & \overline{l} \mp Succ(list). \end{array}$$

The semantics of a parallel frame node is then given by the delayed parallel composition of the semantics of the sub-HMSCs.

Definition B.6.11.3 (Semantics of a node)
For $mscrefexpr \in \mathcal{L}(\langle\text{msc ref expr}\rangle)$, $mscname \in \mathcal{L}(\langle\text{msc name}\rangle)$, $parexpr \in \mathcal{L}(\langle\text{par expression}\rangle)$, $clist \in \mathcal{L}(\langle\text{condition name list}\rangle)$ and $mscexpr \in \mathcal{L}(\langle\text{msc expression}\rangle)$,

$$\begin{array}{lcl} [\![\mathbf{empty}]\!] & = & \varepsilon \\ [\![mscname]\!] & = & \overline{mscname} \\ [\![parexpr]\!] & = & [\![parexpr]\!] \\ [\![\mathbf{condition}\ clist]\!] & = & \varepsilon \\ [\![\mathbf{connect}]\!] & = & \varepsilon \\ [\![\mathbf{(}mscrefexpr\mathbf{)}]\!] & = & [\![mscrefexpr]\!] \\ & & \\ [\![\mathbf{expr}\ mscexpr\ \mathbf{endexpr}]\!] & = & Start(mscexpr), \\ [\![\mathbf{expr}\ mscexpr\ \mathbf{endexpr\ par}\ parexpr]\!] & = & Start(mscexpr) \parallel [\![parexpr]\!]. \end{array}$$

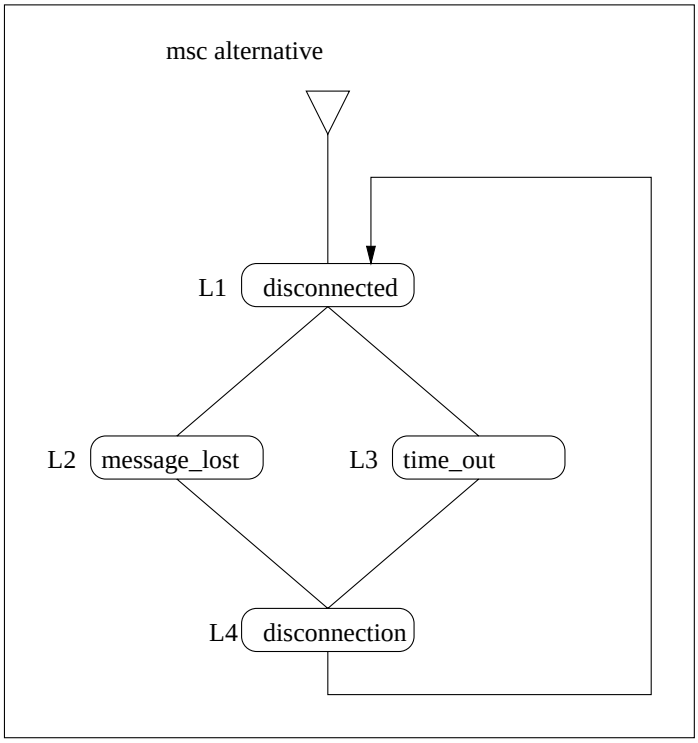
When a recursive specification is described often the curly brackets are omitted.

Example B.6.11.4 Consider the HMSC in Figure B.52. Besides the HMSC also the labels associated with each node and the textual syntax of the HMSC are presented in the figure. With this HMSC the following recursive equations are associated:

$$\begin{array}{lcl} \overline{alternative} & = & \overline{L1} \\ \overline{L1} & = & \overline{disconnected} \circ (\overline{L2} \mp \overline{L3}) \\ \overline{L2} & = & \overline{message\_lost} \circ \overline{L4} \\ \overline{L3} & = & \overline{time\_out} \circ \overline{L4} \\ \overline{L4} & = & \overline{disconnection} \circ \overline{L1}. \end{array}$$

Example B.6.11.5 Consider the *HMSC* shown both in graphical and textual form in Figure B.53. The graphical version of the *HMSC* is annotated by the label names used for the description of the nodes in the textual representation. Applying the semantics to the textual representation results in

$$\begin{array}{lcll} \langle \overline{par\_HMSC} \mid & \overline{par\_HMSC} & = & \overline{L_1}, \\ & \overline{L_1} & = & (\overline{L_2} \parallel \overline{L_4}) \circ \overline{L_6}, \\ & \overline{L_2} & = & \overline{CR} \circ \overline{L_3}, \\ & \overline{L_3} & = & \varepsilon, \\ & \overline{L_4} & = & \overline{DR} \circ \overline{L_5}, \\ & \overline{L_5} & = & \varepsilon, \\ & \overline{L_6} & = & \varepsilon\rangle. \end{array}$$




```
msc alternative;
expr L1;
   L1: disconnected seq (L2 alt L3);
   L2: message_lost seq (L4);
   L3: time_out seq (L4);
   L4: disconnection seq (L1);
endmsc;
```

Figure B.52: HMSC with a loop

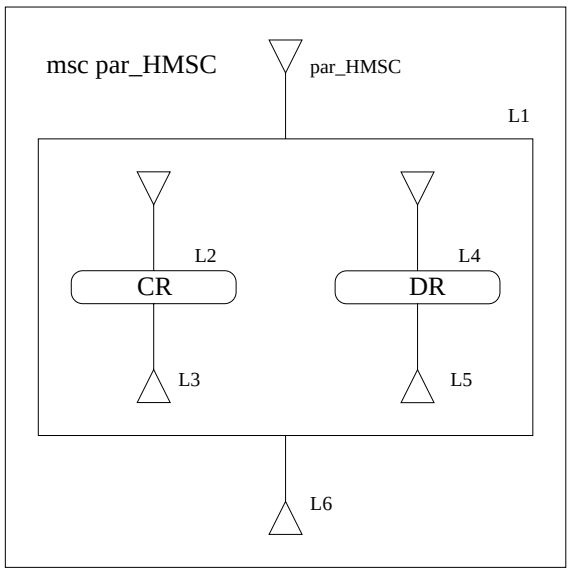


Figure B.53: HMSC with a parallel frame

References

[BM94] J. C. M. Baeten and S. Mauw. Delayed choice: an operator for joining Message Sequence Charts. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII*, IFIP Transactions C, Proceedings 7^{th} International Conference on Formal Description Techniques, pages 340–354. Chapman-Hall, 1994.

[BV95] J. C. M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 149–268. Oxford University Press, 1995.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[IT96a] ITU-TS. *ITU-TS Recommendation Z.120 Annex C: Static Semantics of Message Sequence Charts*. ITU-TS, Geneva, 1996.

[IT96b] ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, October 1996.

[MR97] S. Mauw and M.A. Reniers. High-level Message Sequence Charts. In A. Cavalli and A. Sarma, editors, *SDL'97: Time for Testing - SDL, MSC and Trends*, Proceedings of the Eighth SDL Forum, pages 291–306. Elsevier Science Publishers B.V., 1997.

[MR98] S. Mauw and M.A. Reniers. Operational semantics for MSC96. *Computer Networks and ISDN Systems*, 1998. To appear.

[Par81] D. M. R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proceedings 5^{th} GI Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[Ren95] M. A. Reniers. Syntax requirements of Message Sequence Charts. In R. Bræk and A. Sarma, editors, *SDL'95 with MSC in CASE*, Proceedings of the Seventh SDL Forum, pages 63–74, Amsterdam, 1995. Oslo, North-Holland.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	**Languages and general software aspects for telecommunication systems**

Geneva, 2002