

International Telecommunication Union

**ITU-T**

TELECOMMUNICATION  
STANDARDIZATION SECTOR  
OF ITU

**Z.120**

(02/2011)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE  
ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

---

## **Message Sequence Chart (MSC)**

Recommendation ITU-T Z.120



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
<b>Message Sequence Chart (MSC)</b>	<b>Z.120–Z.129</b>
User Requirements Notation (URN)	Z.150–Z.159
Testing and Test Control Notation (TTCN)	Z.160–Z.179
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-machine interfaces for the management of telecommunications networks	Z.360–Z.379
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	
Processing environment architectures	Z.600–Z.609

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

# Recommendation ITU-T Z.120

## Message Sequence Chart (MSC)

### Summary

#### Scope/objective

The purpose of recommending MSC (Message Sequence Chart) is to provide a trace language for the specification and description of the communication behaviour of system components and their environment by means of message interchange. Since in MSCs the communication behaviour is presented in a very intuitive and transparent manner, particularly in the graphical representation, the MSC language is easy to learn, use and interpret. In connection with other languages it can be used to support methodologies for system specification, design, simulation, testing, and documentation.

#### Coverage

This Recommendation presents a syntax definition for Message Sequence Charts in textual and graphical representation. An informal semantics description is provided.

#### Application

MSC is widely applicable. It is not tailored for one single application domain. An important area of application for MSC is an overview specification of the communication behaviour for real time systems, in particular telecommunication switching systems. By means of MSCs, selected system traces, primarily "standard" cases, may be specified. Non-standard cases covering exceptional behaviour may be built on them. Thereby, MSCs may be used for requirement specification, interface specification, simulation and validation, test case specification and documentation of real time systems. MSC may be employed in connection with other specification languages, in particular SDL. In this context, MSCs also provide a basis for the design of SDL-systems.

#### Status/Stability

MSC is stable. This Recommendation is a maintenance update of the 2004 version correcting a number of errors and consolidating Appendix I into the main text. Otherwise it is intended to be the same as the 2004 version, which was a natural continuation of its 1999 version, refining concepts including:

- extended data interface, and references to default SDL interface, defined in ITU-T Z.121;
- uni-directional time constraints;
- in-line high level expressions.

#### Associated work

- Recommendation ITU-T Q.65 (2000), *The unified functional methodology for the characterization of services and network capabilities including alternative object oriented techniques*.
- Recommendation ITU-T X.210 (1993) | ISO/IEC 10731:1994, *Information technology – Open Systems Interconnection – Basic Reference model: Conventions for the definition of OSI services*.
- Recommendation ITU-T X.292 (2002), *OSI conformance testing methodology and framework for protocol Recommendations for ITU-T applications – The tree and tabular combined notation (TTCN)*.
- Recommendation ITU-T Z.100 (2002), *Specification and Description Language (SDL)*.
- Recommendation ITU-T Z.121 (2003), *Specification and Description Language (SDL) data binding to Message Sequence Charts (MSC)*.
- UML 2.0, OMG 2003.

## History

Edition	Recommendation	Approval	Study Group
1.0	ITU-T Z.120	1993-03-12	X
1.1	ITU-T Z.120 Annex B	1995-03-06	10
2.0	ITU-T Z.120	1996-10-18	10
2.1	ITU-T Z.120 Annex C	1996-10-18	10
2.2	ITU-T Z.120 Annex B	1998-04-01	10
3.0	ITU-T Z.120	1999-11-19	10
3.1	ITU-T Z.120 (1999) Cor. 1	2001-12-14	10
4.0	ITU-T Z.120	2004-04-29	17
4.1	ITU-T Z.120 (2004) Amend.1	2008-09-19	17
4.2	ITU-T Z.120 (2004) Amend.2	2009-09-25	17
5.0	ITU-T Z.120	2011-02-13	17

## FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). 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 Assembly (WTSA), 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 WTSA 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.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

## 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 not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2011

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

## Table of Contents

	<b>Page</b>
1	Introduction ..... 1
1.2	Objectives of MSC ..... 1
1.2	Organization of the Recommendation ..... 1
1.3	Meta-language for textual grammar ..... 2
1.4	Meta-language for graphical grammar ..... 3
2	General Rules ..... 6
2.1	Lexical Rules ..... 6
2.2	Visibility and Naming Rules ..... 12
2.3	Comment ..... 13
2.4	Drawing Rules ..... 14
2.5	Paging of MSCs ..... 15
3	Message Sequence Chart document ..... 15
4	Basic MSC ..... 17
4.1	Message Sequence Chart ..... 17
4.2	Instance ..... 22
4.3	Message ..... 24
4.4	Control Flow ..... 27
4.5	Environment and gates ..... 32
4.6	General ordering ..... 39
4.7	Condition ..... 41
4.8	Timer ..... 43
4.9	Action ..... 46
4.10	Instance creation ..... 47
4.11	Instance stop ..... 48
5	Data concepts ..... 48
5.1	Introduction ..... 48
5.2	Syntax interface to external data languages ..... 48
5.3	Semantic interface to external data languages ..... 51
5.4	Declaring data ..... 53
5.5	Static data ..... 54
5.6	Dynamic data ..... 55
5.7	Bindings ..... 56
5.8	Data in message and timer parameters ..... 58
5.9	Data in instance creation parameters ..... 60
5.10	Data in action boxes ..... 60
5.11	Required data types ..... 61

	<b>Page</b>
6	Time concepts..... 61
6.1	Timed semantics..... 62
6.2	Relative timing..... 62
6.3	Absolute timing..... 62
6.4	Time domain..... 63
6.5	Static and dynamic time variables..... 63
6.6	Time offset..... 63
6.7	Time points, measurements, and intervals..... 63
6.8	Time points..... 63
6.9	Measurements..... 64
6.10	Time interval..... 64
7	Structural concepts..... 67
7.1	Coregion..... 68
7.2	Inline expression..... 69
7.3	MSC reference..... 73
7.4	Instance decomposition..... 77
7.5	High-level MSC (HMSC)..... 85
8	Message Sequence Chart Document..... 90
8.1	MSC Documents..... 90
8.2	Instance decomposition..... 91
8.3	Instance inheritance..... 93
9	Simple Message Sequence Charts..... 94
9.1	Basic MSC..... 94
9.2	Message overtaking..... 95
9.3	MSC basic concepts..... 96
9.4	MSC-composition through labelled conditions..... 97
9.5	MSC with time supervision..... 99
9.6	MSC with message loss..... 101
9.7	Local conditions..... 101
10	Data..... 103
11	Time..... 106
12	Creating and terminating processes..... 110
13	Coregion..... 110
14	General ordering..... 111
14.1	Generalized ordering within a coregion..... 111
14.2	Generalized ordering between different instances..... 112

	<b>Page</b>
15	Inline expressions ..... 113
15.1	Inline expression with alternative composition..... 113
15.2	Inline expression with gates ..... 115
15.3	Inline expression with parallel composition..... 116
16	MSC references ..... 117
16.1	MSC reference..... 117
16.2	MSC reference with gate ..... 119
17	High-level MSC (HMSC)..... 119
17.1	High-level MSC with free loop ..... 119
17.2	High-level MSC with loop ..... 120
17.3	High-level MSC with alternative composition..... 121
17.4	High-level MSC with parallel composition..... 123
Appendix I – Application of MSC ..... 125	
I.1	Introduction ..... 125
I.2	Problems ..... 125
I.3	General undecidable results..... 129
I.4	Syntactical description of MSC subclasses ..... 129
I.5	Summary of results..... 134
I.6	Recommendations ..... 135
Bibliography..... 136	



# Recommendation ITU-T Z.120

## Message Sequence Chart (MSC)

### 1 Introduction

#### 1.2 Objectives of MSC

Message Sequence Charts (MSC) is a language to describe the interaction between a number of independent message-passing instances. The main characteristics of the MSC language are the following.

- MSC is a scenario language. An MSC describes the order in which communications and other events take place. Additionally, it allows for expressing restrictions on transmitted data values and on the timing of events.
- MSC supports complete and incomplete specifications. It has the possibility to describe incomplete behaviours used in early analysis and for documentation purposes.
- MSC is a graphical language. The two-dimensional diagrams give overview of the behaviour of communicating instances. The textual form of MSC is mainly intended for exchange between tools and as a base for automatic formal analysis.
- MSC is a formal language. The definition of the language is given in natural language as well as in a formal notation.
- MSC is a practical language. MSC is used throughout the engineering process. Its use ranges from domain analysis and idea generation via the requirements capture and design phases to testing. MSC is used in slightly different ways in the various phases, and it is important that MSC has formal expressive power as well as intuitive appearance.
- MSC is widely applicable. It is not tailored for one single application domain.
- MSC supports structured design. Simple scenarios (described by Basic Message Sequence Charts) can be combined to form more complete specifications by means of High-level Message Sequence Charts. MSCs are gathered in an MSC document. A modular design of scenarios is supported by mechanisms for decomposition and reuse.
- MSC is often used in conjunction with other methods and languages. Its formal definition enables formal and automated validation of an MSC with respect to a model described in a different language. MSC can, for example be used in combination with SDL and TTCN.
- The usual interpretation of a scenario specified in an MSC is that the actual implementation should at least exhibit the behaviour expressed in the scenario. Alternative interpretations are also possible. An MSC can, for example, be used to specify disallowed scenarios.

#### 1.2 Organization of the Recommendation

The document is structured in the following manner: In 2, general rules concerning syntax, drawing and paging are outlined. In 3, a definition for the Message Sequence Chart document is provided. Section 4 contains the definition of Message Sequence Charts and the basic constituents, i.e., instance, message, general ordering, condition, timer, action, instance creation and termination. Section 5 contains the data concepts and Section 6 defines the concepts for Time. In 7, higher level concepts concerning structuring and modularisation are introduced. These concepts support a top down specification and permit a refinement of individual instances by means of coregion (7.1) and instance decomposition (7.4). Inline expressions are defined in 7.2 and MSC references 7.3. High level MSC (7.4) – permits MSC composition and reusability of (parts of) MSCs on different levels. In 8, examples are provided for all MSC-constructs. Annex A contains an index for the <keyword>s and non-terminals.

### 1.3 Meta-language for textual grammar

In the Backus-Naur Form (BNF) a terminal symbol is either indicated by not enclosing it within angle brackets (that is the less-than sign and greater-than sign, < and >) or it is one of the two representations <name> and <character string>.

The angle brackets and enclosed word(s) are either a non-terminal symbol or one of the two terminals <character string> or <name>. Syntactic categories are the non-terminals indicated by one or more words enclosed between angle brackets. For each non-terminal symbol, a production rule is given either in concrete textual grammar or in concrete graphical grammar. For example

```
<instance parameter decl> ::=
    inst <instance parm decl list> <end>
```

A production rule for a non-terminal symbol consists of the non-terminal symbol at the left-hand side of the symbol ::=, and one or more constructs, consisting of non-terminal and/or terminal symbol(s) at the right-hand side. For example, <instance parameter decl> and <instance parm decl list> and <end> in the example above are non-terminals; **inst** is a terminal symbol.

Sometimes a symbol includes an underlined part. This underlined part stresses a semantic aspect of that symbol, e.g., <msc name> is syntactically identical to <name>, but semantically it requires the name to be a Message Sequence Chart name.

At the right-hand side of the ::= symbol several alternative productions for the non-terminal can be given, separated by vertical bars (|). For example,

```
<incomplete message area> ::=
    <lost message area>
    | <found message area>
```

expresses that an <incomplete message area> is either a <lost message area> or a <found message area>.

Syntactic elements may be grouped together by using curly brackets ({ and }). A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements. For example,

```
<using clause> ::=
    { using <instance kind> <end> }*
```

Repetition of curly bracketed groups is indicated by an asterisk (\*) or plus sign (+). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus sign indicates that the group must be present and can be further repeated any number of times. The example above expresses that a <using clause> may be empty but may also contain any number of occurrences of "**using** <instance kind> <end>".

If syntactic elements are grouped using square brackets ([ and ]), then the group is optional. For example

```
<identifier> ::=
    [ <qualifier> ] <name>
```

expresses that an <identifier> may, but need not, contain <qualifier>.

The meta-language operators have the following precedence order. The operator that binds the weakest is the alternative operator "|". Then follows the left-to-right sequencing. Operators "+" and "\*" bind stronger than the sequencing, and finally the brackets "[ .... ]" and "{ ... }" bind the strongest.

## 1.4 Meta-language for graphical grammar

The meta-language for the graphical grammar is based on a few constructs which are described informally below. The graphical syntax is not precise enough to describe the graphics such that there are no graphical variations. Small variations on the actual shapes of the graphical terminal symbols are allowed. These include, for instance, shading of the filled symbols, the shape of an arrow head and the relative size of graphical elements. Whenever necessary the graphical syntax will be supplemented with informal explanation of the appearance of the constructions.

The meta-language consists of a BNF-like notation with the special meta-constructions: *contains*, *is followed by*, *is associated with*, *is attached to*, *above* and *set*. These constructs behave like normal BNF production rules, but additionally they imply some logical or geometrical relation between the arguments. The *is attached to* construct behaves somewhat differently as explained below. The left-hand side of all constructs except *above* must be a symbol. A symbol is a non-terminal that produces in every production sequence exactly one graphical terminal. We will consider a symbol that *is attached to* other areas or that *is associated with* a text string as a symbol too. The explanation is informal and the meta-language does not precisely describe the geometrical dependencies.

In the following text <area1> and <area2> and <area> are used to denote any syntax non-terminal with a name ending in "area". <non terminal> is used to represent an arbitrary non-terminal.

### contains

"<area1> *contains* <area2>" means that <area2> is contained within <area1>, in principle geometrically, but also logically.

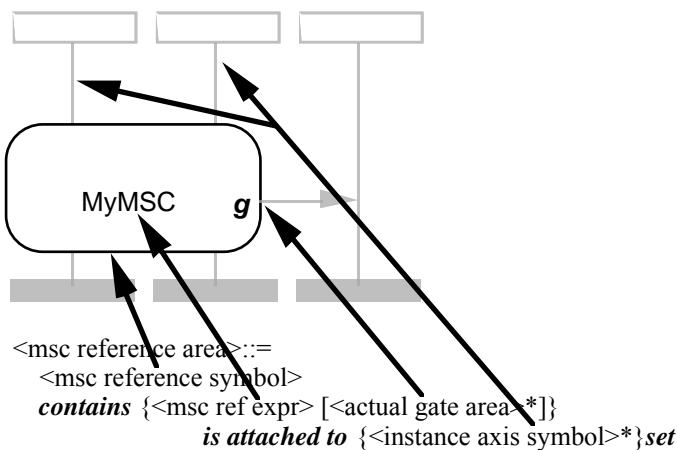
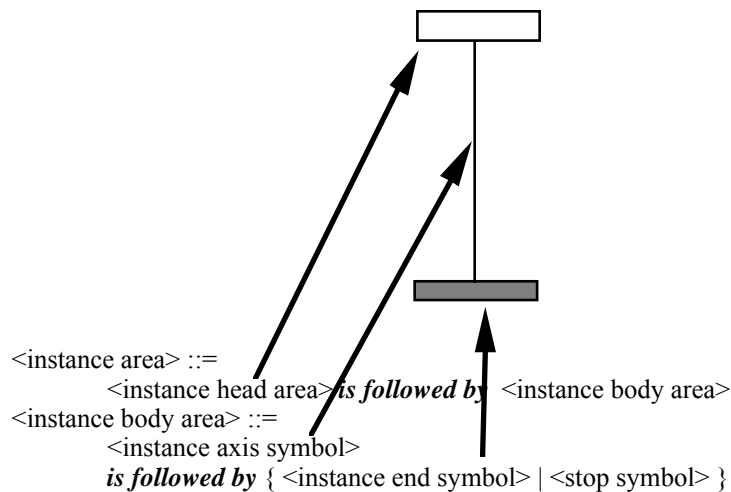


Figure 1 – Example for 'contains'

### is followed by

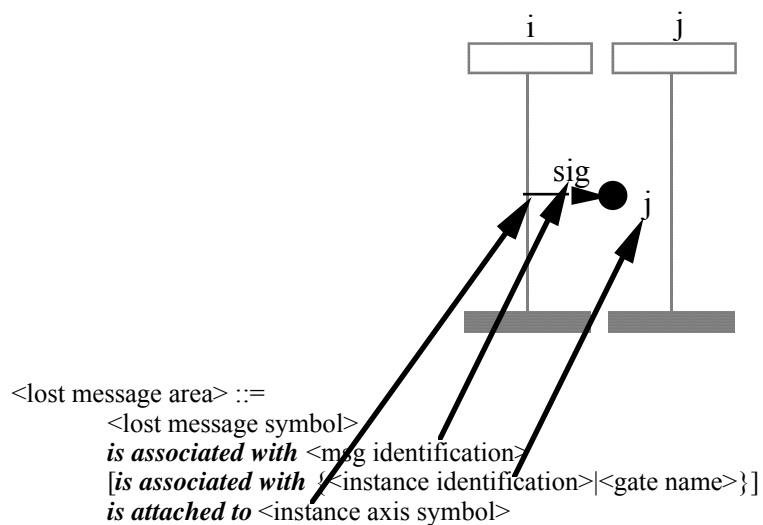
"<area1> *is followed by* <area2>" means that <area2> is logically and geometrically related to <area1>.



**Figure 2 – Example for 'is followed by'**

### is associated with

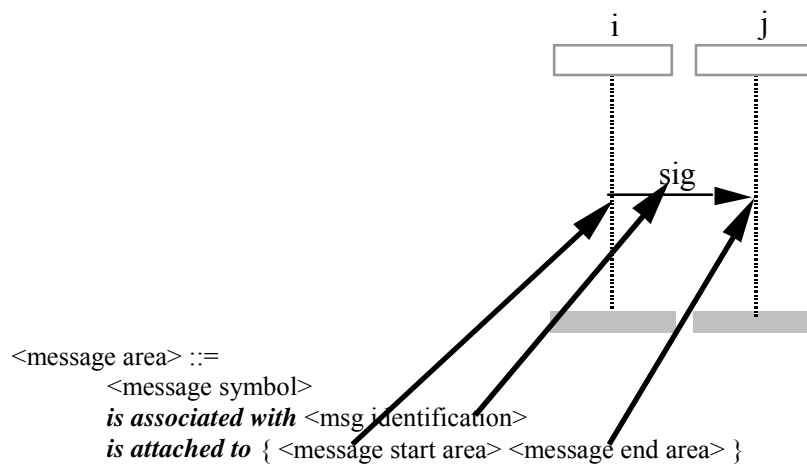
"<area1> *is associated with* <area2>" means that <area2> will expand to a text string which is affiliated with <area1>. There is no closer geometrical association between the areas than the idea that they should be seen as associated with each other.



**Figure 3 – Example for 'is associated with'**

### is attached to

"<area1> *is attached to* <area2>" is not like a normal BNF production rule and its usage is restricted. <area2> must be a symbol or a set of symbols. The meaning of production rule "<non terminal> ::= <area1> *is attached to* <area2>" is that, if a <non terminal> is expanded using this rule, only the symbol <area1> is produced. Instead of also producing <area2>, an occurrence of <area2> is identified which is produced by one of the other productions. The result of the *is attached to* construct is a logical and geometrical relation between <area1> and the symbol <area2> from the implied occurrence. In case the right-hand side is a set of symbols, there is a relation between <area1> and all elements of the set.



**Figure 4 – Example for 'is attached to'**

Notice that if symbol A *is attached to* symbol B, then symbol B is also *attached to* symbol A. Henceforth, an attachment between two symbols is always defined twice in the grammar. We follow one of the non-terminals above to see this:

```

<message end area> ::=
  <message in area> | <actual in gate area>
  | <def out gate area> | <inline gate area>
<message in area> ::=
  <message in symbol>
  is attached to <instance axis symbol>
  is attached to <message symbol>
<message in symbol> ::=
  <void symbol>

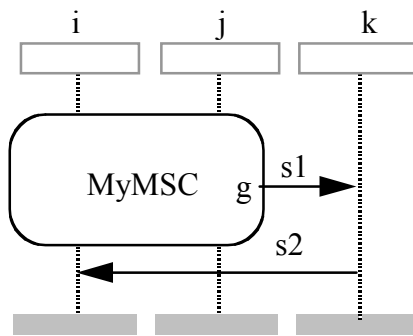
```

Thus a <message symbol> *is attached to* a <message in symbol> and vice versa.

The "*is attached to*" construct may be specialized by prefixing or appending indications to where the construct is attached. The modifiers may be *top*, *bottom*, *left*, *right* or any combination of these. This means that we may have constructs like "<area1> *bottom is attached to top or right* <area2>" meaning that the bottom part of <area1> is attached logically and geometrically to the top or right side of <area2>.

#### **above**

"<area1> *above* <area2>" means that <area1> and <area2> are logically ordered. This is geometrically expressed by ordering <area1> and <area2> vertically. If two <area>s have the same vertical coordinate then no ordering between them is defined, except that an output of a message must occur before an input.



<event layer> ::=  
 <event area> | <event area> **above** <event layer>

**Figure 5 – Example for 'above'**

Figure 5 describes a sequence of events. When the events are on different instances, the geometrical vertical coordinate is of no semantical significance.

Figure 5 describes the following sequence:

i,j: **reference** mr1: MyMSC **gate** g **output** s1 **to** k;  
 k: **input** s1 **from** mr1 **via** g;  
 k: **output** s2 **to** i;  
 i: **input** s2 **from** k;

## set

The **set** metasymbol is a postfix operator operating on the immediately preceding syntactic elements within curly brackets, and indicating a (unordered) set of items. Each item may be any group of syntactic elements, in which case it must be expanded before applying the **set** metasymbol.

<text layer> ::=  
 {<text area>\*} **set**

is a set of zero or more <text area>s.

<msc body area> ::=  
 { <instance layer> <text layer> <gate def layer>  
 <event layer> <connector layer> } **set**

is an unordered set of the elements within the curly brackets.

## 2 General Rules

### 2.1 Lexical Rules

Lexical rules define lexical units. Lexical units are the terminal symbols of the *Concrete textual syntax*.

<lexical unit> ::=  
 | <name>  
 | <character string>  
 | <special>  
 | <composite special>  
 | <note>  
 | <qualifier>  
 | <national>  
 | <misc>  
 | <keyword>

```

<alphanumeric> ::=
    <letter>
    | <decimal digit>
    | <national>
<letter> ::=
    A | B | C | D | E | F | G | H | I | J | K | L | M
    | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
    | a | b | c | d | e | f | g | h | i | j | k | l | m
    | n | o | p | q | r | s | t | u | v | w | x | y | z
<decimal digit> ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<national> ::=
    ` | ¨ | \
    | <left curly bracket>
    | <vertical line>
    | <right curly bracket>
    | <overline>
    | <upward arrow head>
<left square bracket> ::=
    [
<right square bracket> ::=
    ]
<left curly bracket> ::=
    {
<vertical line> ::=
    |
<right curly bracket> ::=
    }
<left open> ::=
    (
<left closed> ::=
    <left square bracket>
<right open> ::=
    )
<right closed> ::=
    <right square bracket>
<abs time mark> ::=
    @
<rel time mark> ::=
    &
<overline> ::=
    ~
<upward arrow head> ::=
    ^
<full stop> ::=
    .
<underline> ::=
    -
<left angular bracket> ::=
    <
<right angular bracket> ::=
    >

```

```

<character string> ::=
    <apostrophe>{<alphanumeric>
    | <other character>
    | <special>
    | <full stop>
    | <underline>
    | <space>
    | <apostrophe><apostrophe>}* <apostrophe>

<text> ::=
    { <alphanumeric>
    | <other character>
    | <special>
    | <full stop>
    | <underline>
    | <space>
    | <apostrophe> }*

<misc> ::=
    <other character>
    | <apostrophe>

<apostrophe> ::=
    '

<other character> ::=
    ? | % | + | - | ! | / | * | " | =

<special> ::=
    <abs time mark> | <rel time mark>
    | <left open> | <right open>
    | <left closed> | <right closed>
    | <left angular bracket> | <right angular bracket>
    | # | , | ; | :

<composite special> ::=
    <qualifier left> | <qualifier right>

<qualifier left> ::=
    <<

<qualifier right> ::=
    >>

<note> ::=
    /* <text> */

<qualifier> ::=
    <qualifier left> <text> <qualifier right>

```

The text in a qualifier must not contain '<<' or '>>'.

```

<name> ::=
    {<letter> | <decimal digit> | <underline> | <full stop>}+

```



<keyword> ::=

**action**  
**after**  
**all**  
**alt**  
**as**  
**before**  
**begin**  
**bottom**  
**call**  
**comment**  
**concurrent**  
**condition**  
**connect**  
**create**  
**data**  
**decomposed**  
**def**  
**empty**  
**end**  
**endconcurrent**  
**endexpr**  
**endinstance**  
**endmethod**  
**endmsc**  
**endmscdocument**  
**endsuspension**  
**env**  
**equalpar**  
**escape**  
**exc**  
**external**  
**final**  
**finalized**  
**found**  
**from**  
**gate**  
**in**  
**inf**  
**inherits**  
**initial**  
**inline**  
**inst**  
**instance**  
**int\_boundary**  
**label**  
**language**  
**loop**  
**lost**  
**method**  
**msc**  
**mscdocument**  
**msg**  
**nestable**  
**nonnestable**  
**offset**  
**opt**  
**origin**  
**otherwise**  
**out**  
**par**  
**parenthesis**  
**receive**

	<b>redefined</b>
	<b>reference</b>
	<b>related</b>
	<b>replyin</b>
	<b>replyout</b>
	<b>seq</b>
	<b>shared</b>
	<b>starttimer</b>
	<b>stop</b>
	<b>stoptimer</b>
	<b>suspension</b>
	<b>text</b>
	<b>time</b>
	<b>timeout</b>
	<b>timer</b>
	<b>to</b>
	<b>top</b>
	<b>undef</b>
	<b>using</b>
	<b>utilities</b>
	<b>variables</b>
	<b>via</b>
	<b>virtual</b>
	<b>when</b>
	<b>wildcards</b>

<space> ::=

ITU-T Alphabet No. 5 character for a space

The <national> characters are represented above as in [b-ITU-T T.50]. The responsibility for defining the national representations of these characters lies with national standardisation bodies.

Control characters are defined as in [b-ITU-T T.50]. A sequence of control characters may appear where a <space> may appear, and has the same meaning as a <space>. The <space> represents the ITU-T T.50 character for a space. Any number of <space>s may be inserted before or after any <lexical unit>. Inserted <space>s or <note>s have no syntactic relevance, but sometimes a <space> or a <note> is needed to separate one <lexical unit> from another.

An occurrence of a control character is not significant in <note>. A control character cannot appear in character string literals if its presence is significant.

In all <lexical unit>s upper case and lower case letters are distinct, which means that AB, ab, Ab and aB are four different lexical units.

A <lexical unit> is terminated by the first character which cannot be part of the <lexical unit> according to the syntax specified above. When an <underline> character is followed by one or more <space>s, all of these characters (including the <underline>) are ignored, e.g., A\_ B denotes the same <name> as AB. This use of <underline> allows <lexical unit>s to be split over more than one line.

When the character / is immediately followed by the character \* outside of a <note>, it starts a <note>. The character \* immediately followed by the character / in a <note> always terminates the <note>. A <note> may be inserted before or after any <lexical unit>.

A data string is a string of characters that will be analyzed by an analyzer external to MSC. The MSC lexical analyzer needs to return the data string as one token that will in turn be given to the external analyzer.

Since it would be preferable that the data string can be recognized from MSC and from the data language without a number of extra delimiting characters, concepts are defined in the MSC grammar to match most data languages directly.

Three concepts of parenthesis are defined:

- normal parenthesis that may be nested meaning that parentheses may have parentheses nested inside,
- non-nestable parentheses where everything inside the parenthesis is considered irrelevant; typical of comment delimiters,
- symmetrical, or equal, parenthesis where the opening and the closing parenthesis characters are the same, as typical of string and character quotes.

The actual parenthesis delimiters are declared inside `<parenthesis declaration>` found at `<document head>` of the MSC document.

To make the approach applicable in all circumstances a global escape can also be declared in `<parenthesis declaration>` to describe when parenthesis stings are not to be considered as parenthesis.

```

<string> ::=
    <pure data string>

<pure data string> ::=
    <non parenthesis> [<parenthesis>] [<pure data string>]

<parenthesis> ::=
    <nestable par> | <equal par> | <non nestable par>

<nestable par> ::=
    <open par> <pure data string> <close par>

```

The `<open par>` and `<close par>` must be corresponding parenthesis strings defined as a nestable parenthesis pair in the MSC document heading defined in 5.2.

```

<non parenthesis> ::=
    {<non par non escape> |
    <escapechar> {<escapechar> | <open par> | <close par> | <equal par> } }*

<non par non escape> ::=
    character string containing no escape char and no parenthesis delimiters and
    not the possible terminators 00000000=00=0or comment

<equal par> ::=
    <equal par> <unmatched string> <equal par>

```

The two `<equal par>` matches must be identical and defined as an equal parenthesis string in the MSC document heading.

```

<unmatched string> ::=
    <non parenthesis> [<nestable par>] [<unmatched string>]

<non nestable par> ::=
    <open par> <text> <close par>

```

The `<open par>` and `<close par>` must be corresponding parenthesis strings defined as a non-nestable parenthesis pair in the MSC document heading.

If the `<string>` needs to contain the characters that would otherwise terminate a `<non par non escape>`, then either the characters must be escaped or the string be enclosed in matching parenthesis.

## 2.2 Visibility and Naming Rules

Entities are identified and referred to by means of associated names. Entities are grouped into entity classes to allow flexible naming rules. The following entity classes exist:

- a) MSC document
- b) MSC
- c) instance
- d) condition
- e) timer
- f) message
- g) gate
- h) split time interval connector
- i) variable
- j) general name

The scope units are the whole system of MSC documents, one MSC document and one MSC.

The whole system of MSC documents is the scope for defining MSC document names (which are equivalent to instance kind names).

The MSC document is the scope for defining MSCs, instances, conditions, timers, messages, split time interval connectors, variables and general names.

The MSC is the scope for defining gates and MSC formal parameters. Formal parameters take precedence over entities defined in the MSC document.

Instance without explicit kind will take its name as its implicit kind.

Messages going in and out of decomposed instances must be redeclared with the same name and signature within the MSC document defining the decomposition. A message signature consists of the message parameters' types and order.

A gate is referenced from outside of its defining MSC, and only in a reference to that MSC.

In the textual notation it is sometimes necessary to introduce specific identifiers for individual objects that need no explicit naming in the graphical notation. An example is MSC references where the graphical notation can easily handle two or more occurrences of the same MSC reference while the textual notation must distinguish them by unique individual identifiers.

While lines and spatial proximity can be used to tie constructs together in a graphic description, a textual notation will need identifiers for the same purpose. In some cases, the graphic notation will also need identifiers on entities because a relation which is spatially far distributed shall be described. Such identifiers can be found on a number of constructs. The MSC reference contains an identifier, the gates may be identified by names. Such names are either contained in a symbol (like the MSC reference), spatially adjacent described in the graphical grammar as *is associated with* (like message gates on a diagram), or a special naming symbol containing the name, which is attached to the construct (like for instance event areas). When gates have no explicit name, an implicit name is constructed.

<general name area> ::=  
     <general name symbol> **contains** <name>  
     **is attached to** {<instance event area> | <inline expression area> |  
     <operand area>}  
 <general name symbol> ::=



The <general name symbol> may also be mirrored. The contained <name> is contained within the circle segment, but may extend beyond the symbol through the open end. The <general name area> is attached to other constructs through the straight line.

### 2.3 Comment

There are three kinds of comments.

Firstly there is the *note* which occurs only in the textual syntax (see lexical rules).

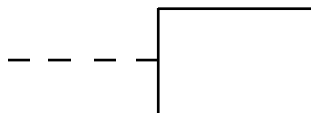
Secondly there is the *comment* which is a notation to represent informal explanations associated with symbols or text.

The *concrete textual syntax* of the comments is:

<end> ::=  
     [ <comment>];  
 <comment> ::=  
     **comment** <character string>

In the *concrete graphical grammar* the following syntax is used:

<comment area> ::=  
     <comment symbol> **contains** <text>  
 <comment symbol> ::=



A <comment symbol> can be attached to the graphical symbols.

Thirdly there is the *text* which may be used for the purpose of global comments.

Text in *textual grammar* is defined by:

<text definition> ::=  
     **text** <character string> <end>

In *graphical grammar* text is defined by:

<text area> ::=  
     <text symbol> **contains** <text>  
 <text symbol> ::=



## 2.4 Drawing Rules

The size of the graphical symbols can be chosen by the user. Symbol boundaries must not overlap or cross. An exception to this rule applies for the crossing or overlap of:

- a) a message symbol, reply symbol and general ordering symbol with message symbol, reply symbol, general ordering symbol, lines in timer symbols, createline symbol, instance axis symbol, method symbol, suspension symbol, coregion symbol, dashed line in comment symbol and condition symbol,
- b) lines in timer symbols with message symbol, reply symbol, general ordering symbol, lines in timer symbols, createline symbol, dashed line in comment symbol and condition symbol,
- c) a message symbol, reply symbol and general order symbol with the inline expression symbol or exc inline expression symbol when the in and out events of message symbol, reply symbol and general ordering symbol are not connected to the same instance axis,
- d) a createline symbol with message symbol, reply symbol, general ordering symbol, lines in timer symbols, instance axis symbol, method symbol, suspension symbol, coregion symbol and dashed line in comment symbol,
- e) a condition symbol with instance axis symbol, method symbol, coregion symbol, message symbol, reply symbol, general ordering symbol, lines in timer symbols,
- f) an inline expression symbol with instance axis symbol,
- g) a reference symbol with instance axis symbol and method symbol,
- h) an action symbol with instance axis symbol in line form, and the overlap of action symbol with instance axis symbol in column form,
- i) a method symbol, suspension symbol and coregion symbol with instance axis symbol,
- j) a method symbol with method symbol and suspension symbol,
- k) an hmsc line symbol with another hmsc line symbol.

There are two forms of the instance axis symbol and the coregion symbol: the single line form and the column form. It is not allowed to mix both forms within one instance except single line axis with column form coregions.

If a shared condition (see 4.7) crosses an instance which is not involved in this condition the instance axis is drawn through.

If a shared reference (see 7.3) crosses an instance which is not involved in this reference the instance axis is drawn through.

If a shared inline expression (see 7.2) crosses an instance which is not involved in this inline expression the instance axis may not be drawn through.

In case where the instance axis symbol has the column form, the vertical boundaries of the action symbol have to coincide with the column lines.

Message lines may be horizontal or with downward slope (with respect to the direction of the arrow), and they may be a connected sequence of straight line segments.

If an incoming event and an outgoing event are on the same point of an instance axis, then it is interpreted as if the incoming event is drawn above the outgoing event. A general ordering cannot be attached to this point. It is not allowed to draw two or more outgoing events on the same point. It is not allowed to draw two or more incoming events on the same point. The following events are incoming events: message input, found message, receive, found receive, reply-in, found reply-in and time-out. The following events are outgoing events: message output, lost message, call, lost call, reply-out, lost reply-out, timer start, timer stop and instance creation.

## 2.5 Paging of MSCs

MSCs can be partitioned over several pages. The partitioning may be both horizontal and vertical. The partitioning may alternatively be circumvented by means of MSC composition or instance decomposition (see 7.4).

If an MSC is partitioned vertically into several pages, then the <msc heading> is repeated on each page, but the instance end symbol may only appear on one page (on the "last" page for the instance in question). For each instance the <instance head area> must appear on the first page where the instance in question starts and must be repeated in dashed form on each of the following pages where it is continued.

If messages, timers, create statements or conditions are continued from one page to the next page then the entire text associated with the message, timer, create or condition must be present on the first page and all or part of the text may be repeated on the next page.

Page numbering must be included on the pages of an MSC in order to indicate the correct position of the pages. Pages must be numbered in pairs: "v-h" where "v" is the vertical page number and "h" the horizontal page number. Arabic numerals must be used for the vertical numbers and English upper case letters ('A' to 'Z') for the horizontal. If the range 'A'-'Z' is not sufficient then it is extended with 'AA' to 'AZ', 'BA' to 'BZ' etc.

## 3 Message Sequence Chart document

The Message Sequence Chart document defines an instance kind and the associated collection of message sequence charts, which again defines a set of traces. MSC documents can be described graphically or textually. Since the Message Sequence Chart document defines the instances used within the message sequence charts, Message Sequence Chart documents define the instance decomposition structures.

The Message Sequence Chart document header contains the document name and optionally, following the keyword **related to**, the identifier (pathname) of the document to which the MSCs refer. This document may be described in SDL, UML, TTCN, etc.

### Concrete textual grammar

```
<textual msc document> ::=
    <document head>
    <textual defining part> <textual utility part>
    <msc document body>

<document head> ::=
    mscdocument <instance kind> [ related to <sdl reference> ]
    [<inheritance>] <end>
    [<parenthesis declaration> ]
    <data definition>
    <using clause>
    <containing clause>
    <message decl clause>
    <timer decl clause>

<textual defining part> ::=
    <defining msc reference>*

<textual utility part> ::=
    utilities [<containing clause>] <defining msc reference>*

<defining msc reference> ::=
    reference [<virtuality>] <msc name>

<virtuality> ::=
    virtual | redefined | finalized
```

```

<using clause> ::=
    { using <instance kind> <end> } *

<containing clause> ::=
    { inst <instance item> } +

<instance item> ::=
    <instance name> [ : <instance kind> ] [ <inheritance> ]
    [ <decomposition> ]
    { <dynamic decl list> | <end> }

<inheritance> ::=
    inherits <instance kind>

<message decl clause> ::=
    { msg <message decl> <end> } *

<timer decl clause> ::=
    { timer <timer decl> <end> } *

<sdl reference> ::=
    <sdl document identifier>

<identifier> ::=
    [ <qualifier> ] <name>

<msc document body> ::=
    <message sequence chart> * endmscdocument <end>

```

The decomposed-clause in <instance item> does not apply to lists used in <document head>, but to decomposition of instances in high-level MSCs.

### Concrete graphical grammar

```

<msc document area> ::=
    <frame symbol> contains { <document head>
    is followed by <defining part area> is followed by <utility part area> }

<defining part area> ::=
    { { <defining msc reference area> * } set } is followed by <separator area>

<utility part area> ::=
    [ <containing area> is followed by ] { { <defining msc reference area> * } set

<containing area> ::=
    <containing clause>

<defining msc reference area> ::=
    <msc reference symbol>
    contains [ <virtuality> ] <msc name>

```

### Static requirements

Instances contained in the MSCs referenced from the defining part and utility part must only be chosen from the list of instances of the <containing clause> in the <document head> and the start of the utility part.

The optional containing clause of the utility part will include instances that are contained within the MSCs of the utility part, but not of the defining part.

Instances contained in the <containing clause> may be **decomposed** in some of the MSCs. Let the instance  $x$  contained in MSC  $m$  be **decomposed as**  $xm$ . Then the defining part of the MSC document for  $x$  must contain the MSC  $xm$ . All instances that are decomposed in MSCs must be declared as decomposed in the containing clauses.



When there is an **inherits**-clause inside the containing clause, and the inheriting instance kind is itself defined by an MSC document, there must be a corresponding **inherits**-clause of its <document head>.

The <virtuality>-clause of the defining MSC reference must correspond to the <msc heading> of the definition of the MSC.

A **redefined** or **finalized** MSC must have a corresponding MSC in the inherited instance kind which is not **finalized** in the inherited instance kind.

The environment messages of a **redefined** MSC must correspond exactly to those of the MSC it is redefining: no fewer or extra messages to/from the environment are permitted.

## Semantics

A Message Sequence Chart document is a collection of Message Sequence Charts, optionally referring to a corresponding SDL-document.

The MSC document defines an instance or instance kind. An instance has a collection of Message Sequence Charts associated. The MSCs are scenarios of interaction between the instances contained in the defined instance. The defining part (defining MSCs) defines the collection of MSC traces while the utility part contains only patterns (utility MSCs) reused by the defining part. Nevertheless it is allowed for the MSCs in the utility part to reference MSCs of the defining part.

The meaning of the optional **inherits**-clause is the same as

- 1) including all instances of the inherited kind into the containing clauses of the inheriting kind,
- 2) including all defining MSCs of the inherited kind into the defining part of the inheriting kind, and
- 3) including all utility MSCs of the inherited kind into the utility part of the inheriting kind,
- 4) replacing every **virtual** or **redefined** MSCs of the inherited kind with the corresponding **redefined** or **finalized** MSC of the inheriting kind.

The optional **using**-clause defines usage of MSC documents as libraries of MSCs. The meaning of the **using**-clause is to include the defining part of the library MSC in the utility part of the enclosing MSC document.

The instance names may be qualified by names of enclosing instances.

The keyword **decomposed** in the containing clause indicates that the instance is decomposed within at least one MSC.

## 4 Basic MSC

### 4.1 Message Sequence Chart

A Message Sequence Chart, which is normally abbreviated to MSC, describes the message flow between instances. One Message Sequence Chart describes a partial behaviour of a system. Although the name *Message Sequence Chart* obviously originates from its graphical representation, it is used both for the textual and the graphical representation.

An MSC is described either by instances and events or by an expression that relates MSC references without explicitly mentioning the instances (see 7.5).

## Concrete textual grammar

<message sequence chart> ::=  
    [<virtuality>] **msc** <msc head> { <msc> | <hmsc> } **endmsc** <end>

<msc> ::=  
    <msc body>

<msc head> ::=  
    <msc name>[<msc parameter decl>] [ <time offset> ]<end>  
    [ <msc inst interface> ] <msc gate interface>

<msc parameter decl> ::=  
    <left open> <msc parm decl list> <right open>

<msc parm decl list> ::=  
    <msc parm decl block> [ <end> <msc parm decl list> ]

<msc parm decl block> ::=  
    <data parameter decl>  
    | <instance parameter decl>  
    | <message parameter decl>  
    | <timer parameter decl>

<instance parameter decl> ::=  
    **inst** <instance parm decl list>

<instance parm decl list> ::=  
    <instance parameter name> [ : <instance kind> ] [ , <instance parm decl list> ]

<instance parameter name> ::=  
    <instance name>

<message parameter decl> ::=  
    **msg** <message parm decl list>

<message parm decl list> ::=  
    <message decl list>

<timer parameter decl> ::=  
    **timer** <timer parm decl list>

<timer parm decl list> ::=  
    <timer decl list>

<msc inst interface> ::=  
    <containing clause>

<instance kind> ::=  
    [ <kind denominator> ] <identifier>

<kind denominator> ::=  
    <name>

<msc gate interface> ::=  
    <msc gate def>\*

<msc gate def> ::=  
    **gate** { <msg gate> | <method call gate> | <reply gate> |  
    <create gate> | <order gate> } <end>

<msg gate> ::=  
    <def in gate> | <def out gate>

<method call gate> ::=  
    <def out call gate> | <def in call gate>

<reply gate> ::=  
    <def out reply gate> | <def in reply gate>

<create gate> ::=  
    <def create in gate> | <def create out gate>

```

<order gate> ::=
    <def order in gate> | <def order out gate>

<msc body> ::=
    <msc statement>*
    | <instance decl statement>*

<msc statement> ::=
    <text definition> | <event definition>

<event definition> ::=
    <instance name> : <instance event list>
    | <instance name list> : <multi instance event list>

<instance event list> ::=
    <start method> <instance event>* <end method>
    | <start suspension> <instance event>* <end suspension>
    | <start coregion> <instance event>* <end coregion>
    | <instance head statement> <instance event>*
      { <instance end statement> | <stop> }

<instance decl statement> ::=
    <instance head statement> { <orderable event> | <non orderable event> }*
    { <instance end statement> | <stop> }

<instance event> ::=
    <orderable event> | <non orderable event>

<orderable event> ::=
    [ label <event name> <end> ]
    { <message event> | <incomplete message event> |
      <method call event> | <incomplete method call event> | <create> |
      <timer statement> | <action> }
    [ before <order dest list> ] [ after <order dest list> ] <end>
    [ time <time dest list> <end> ]

```

The optional **label** <event name> ; is used when the event is generally ordered.

```

<order dest list> ::=
    <order dest> [ , <order dest list> ]

<time dest list> ::=
    [ <time dest> [ origin ] ] <time interval> [ , <time dest list> ]

<time dest> ::=
    <event name> | { top | bottom } { <reference identification> | <label name> }

```

The <time dest list> for orderable events is used to express time constraints. In the relative timing case, timing is given with respect to a previous event. In the absolute timing, the absolute time is assigned. In the case of time intervals between pairs of events, or events and regions such as in-line expressions and references, the keyword **origin** is used to indicate a directed constraint that starts with the statement in which it appears. The keyword can only appear at most once in the two events or regions to which the time constraint applies. That is, the constraint is either undirected or unidirectional.

The <gate name> in <order dest> refers to a <def order out gate>, <actual order in gate>, <inline order out gate> or <def order out gate>. The <event name> in <order dest> refers to an <orderable event>.

```

<non orderable event> ::=
    <shared condition> |
    <shared msc reference> |
    <shared inline expr>

<instance name list> ::=
    <instance name> { , <instance name> }* | all

```

<multi instance event list> ::=  
     <multi instance event> +  
 <multi instance event> ::=  
     <condition> | <msc reference> | <inline expr>

## Static requirements

The blocks of MSC parameter declarations given inside a <msc parm decl list>, viz variable, instance, message, and timer blocks, can be given in any order, but there can be at most one block of each type. For example, there cannot be two blocks of message parameters, even if separated by an instance declaration block.

For each <instance head statement> there must exactly one <instance end statement> or exactly one <stop> event. For each instance there must be no events before its <instance head statement> is defined. For each instance there must be no events after its <instance end statement>. For each instance there must be not more than one <instance head statement> and not more than one <instance end statement>.


The instances specified within the <msc> or <hmsc> must be a subset of the instances specified in the enclosing MSC document.

The <containing clause>, if present, must contain the same instances as specified within the <msc> or <hmsc>.

For <hmsc>s, decomposition of instances must be described in the decomposition-clause of the <containing clause> in the MSC heading.

The **label** <name>s used in a <time dest> of an <hmsc> must refer only to <timeable node>s.

## Concrete graphical grammar

<msc diagram> ::=  
     <simple msc diagram> | <hmsc diagram>  
 <simple msc diagram> ::=  
     <msc symbol>  
     *contains* <msc heading> <msc body area>  
 <hmsc diagram> ::=  
     <msc symbol>  
     *contains* {<msc heading> [<containing clause>]} <hmsc area>  
 <msc symbol> ::=  
     <frame symbol> *is attached to* { <def gate area>\* } *set*  
 <frame symbol> ::=  
       
 <msc heading> ::=  
     **msc** <msc name>[<msc parameter decl>] [ <time offset> ]  
 <msc body area> ::=  
     { <instance layer> <text layer> <gate def layer>  
     <event layer> <connector layer> } *set*  
 <instance layer> ::=  
     { <instance area>\* } *set*  
 <text layer> ::=  
     { <text area>\* } *set*  
 <gate def layer> ::=  
     { <def gate area>\* } *set*  
 <event layer> ::=  
     <event area> | <event area> *above* <event layer>

```

<connector layer> ::=
    { <message area>* | <incomplete message area>* |
      <method call area>* | <incomplete method call area>*
      <reply area>* | <incomplete reply area>* } set

<event area> ::=
    <instance event area>
    | <shared event area>
    | <create area>

<instance event area> ::=
    { <message event area>
      <method call event area>
      <reply event area>
      <timer area>
      <concurrent area>
      <method area>
      <suspension area>
      <action area> }
    [is followed by <general name area>]

<shared event area> ::=
    <condition area>
    | <msc reference area>
    | <inline expression area>

```

## Static requirements

All messages and timers with parameters must be declared in the enclosing MSC document. Instances used within the MSCs must also be defined in the enclosing MSC document.

The instance kinds of the instance parameters must be defined in the enclosing MSC document. Instance kinds are defined by the fact that instances of the specified kind are contained in the MSC document. When the instance kind is not present in the parameter declaration, this is equivalent to any kind.

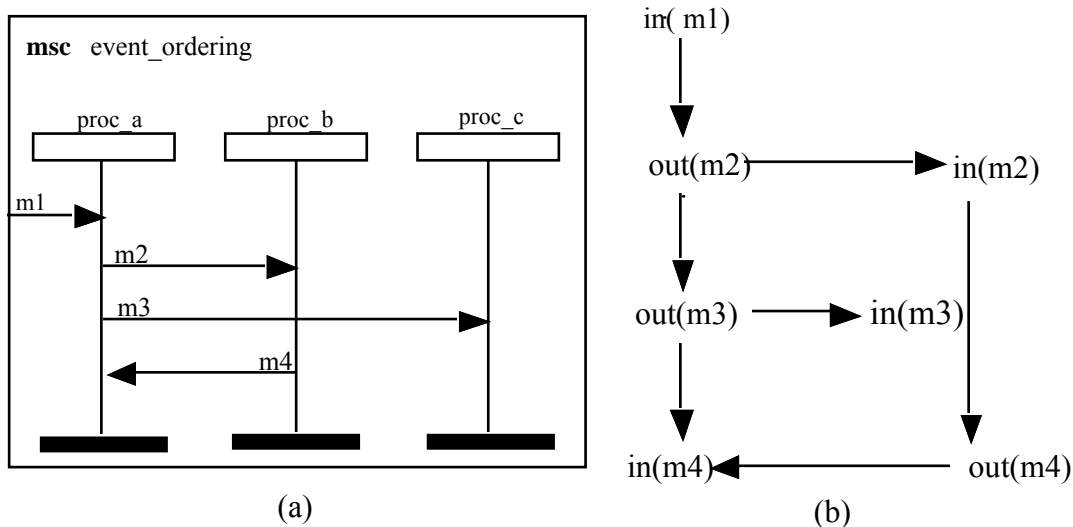
## Semantics

An MSC describes the communication between a number of system components, and between these components and the rest of the world, called environment. For each system component covered by an MSC there is an instance axis. The communication between system components is performed by means of messages. The sending and consumption of messages are two asynchronous events. It is assumed that the environment of an MSC is capable of receiving and sending messages from and to the Message Sequence Chart; no ordering of message events within the environment is assumed.

A global clock is assumed for one Message Sequence Chart. Along each instance axis the time is running from top to bottom, however, a proper time scale is not assumed. If no coregion or inline expression is introduced (see 7.1, 7.2) a total time ordering of events is assumed along each instance axis. Events of different instances are ordered via messages – a message must first be sent before it is consumed – or via the generalized ordering mechanism. With this generalized ordering mechanism "orderable events" on different instances (even in different MSCs) can be ordered explicitly. No other ordering is prescribed. A Message Sequence Chart therefore imposes a partial ordering on the set of events being contained. A binary relation which is transitive, antisymmetric and irreflexive is called partial order.

For the message inputs (labelled by  $in(m_i)$ ) and outputs (labelled by  $out(m_i)$ ) of the Message Sequence Chart in Figure 6(a) we derive the following ordering relation:  $out(m_2) < in(m_2)$ ,  $out(m_3) < in(m_3)$ ,  $out(m_4) < in(m_4)$ ,  $in(m_1) < out(m_2) < out(m_3) < in(m_4)$ ,  $in(m_2) < out(m_4)$  together with the transitive closure.

The partial ordering can be described in a minimal form (without an explicit representation of the transitive closure) by its connectivity graph (Figure 6(b)).



**Figure 6 – Message Sequence Chart and corresponding connectivity graph**

A formal semantics of MSCs based on process algebra is provided in Annex B. The semantics of an MSC can be related to the semantics of SDL by the notion of a reachability graph. Each sequentialization of an MSC describes a trace from one node to another node (or a set of nodes) of the reachability graph describing the behaviour of an SDL system specification. The reachability graph consists of nodes and edges. Nodes denote global system states. A global system state is determined by the values of the variables and the state of execution of each process and the contents of the message queues. The edges correspond to the events which are executed by the system e.g., the sending and the consumption of a message or the execution of a task. A sequentialization of an MSC denotes one total ordering of events compatible with the partial ordering defined by the MSC.

In the textual representation, the <event definition> is provided either by an <instance name> followed by the attached <instance event list> or by <instance name list> followed by an attached <multi instance event> whereby a colon symbol serves as separator. The nonterminal <instance event> denotes either an event which is attached to one single instance, e.g., <action> or a shared object e.g., <shared condition> whereby the keyword **shared** together with the instance list or the keyword **all** is used to denote the set of instances by which the condition is shared. A shared object may be represented alternatively by <multi instance event>. The different notations are introduced in order to facilitate an *instance oriented* description on the one hand and an *event oriented* description on the other hand. Both notations may be mixed arbitrarily.

The *instance oriented* description lists events in association with an instance.

Within the *event oriented* textual representation, the events may be listed in form of a possible execution trace and not ordered with respect to instances.

The <msc inst interface> provides a declaration of the instances i.e., <instance name> and optionally <instance kind>. The <msc gate interface> provides a definition of message and ordering gates contained in the MSC. Message gates define the connection points of messages with the environment. Optionally, gate names may be associated to gates.

## 4.2 Instance




A Message Sequence Chart is composed of interacting instances. An instance of an instance kind has the properties of this kind. Within the instance heading the instance kind name, e.g., process name, may be specified in addition to the instance name.

Within the instance body the ordering of events is specified.

## Concrete textual grammar

<instance head statement> ::=  
    **instance** [ <instance kind> ] [ <decomposition> ] <end>  
<instance end statement> ::=  
    **endinstance** <end>

## Concrete graphical grammar

<instance area> ::=  
    <instance fragment area> [ *is followed by* <instance area> ]  
<instance fragment area> ::=  
    <instance head area> *is followed by* <instance body area>  
<instance head area> ::=  
    <instance head symbol>  
    *is associated with* <instance heading>  
    [*is attached to* <createline symbol>  
    [*is attached to*  
    { {<int symbol> | <abs time symbol>}\* } *set*]]  
<instance heading> ::=  
    <instance name> [ [ : ] <instance kind> ] [ <decomposition> ]  
<instance head symbol> ::=  
      
<instance body area> ::=  
    <instance axis symbol>  
    *is followed by* { <instance end symbol> | <stop symbol> }  
<instance axis symbol> ::=  
    { <instance axis symbol1> | <instance axis symbol2> }  
    *is attached to* { <event area>\* } *set*  
<instance axis symbol1> ::=  
      
<instance axis symbol2> ::=  
      
<instance end symbol> ::=  
    

## Static Requirements

The <instance heading> may be placed above or inside of the <instance head symbol> or split such that the <instance name> is placed inside the <instance head symbol> whereas the <instance kind> and <decomposition> is placed above. If the <instance heading> is split, the colon symbol is optional.

All instance fragments with the same name must be placed directly under each other when occurring on the same page.

If an instance is created but not terminated inside a referenced MSC, then the instance head and axis must be included on the referring MSC, and no events may be placed between the head and the reference. This permits the continuation of the instance axis below the reference to be identified with the correct instance name. Conversely, if an instance is terminated (stopped) but not created within a referenced MSC, then the instance axis must be terminated on the referencing MSC, and no events may be placed between the reference and the stop symbol. These rules ensure that a reference always has the same number of instance axes attached below as it does above.

## Semantics

Within the Message Sequence Chart body the instances are defined. The instance end symbol determines the end of a description of the instance within an MSC. It does not describe the termination of the instance (see 4.11). Correspondingly, the instance head symbol determines the start of a description of the instance within an MSC. It does not describe the creation of the instance (see 4.10). All instance fragments with the same name constitute the same instance. Static instances of an MSC document are those instances that appear within its defining part but which are not dynamically created. The static instances are created when their enclosing instance is created – the enclosing instance (kind) being the name of the MSC document.

In the context of SDL an instance may refer to entities such as processes, blocks, or systems, and outside of SDL it may refer to any kind of entity. The <kind denominator> permits the user to name the kind of entity that the instance relates to, such as process, block, etc., and the <instance kind> identifier can be used to give the particular name for that entity class. Multiple instances can share the same <instance kind> identifier to denote that they are of the common kind.

The instance definition provides an event description for message inputs and message outputs, method calls and replies, actions, shared and local conditions, timer events, instance creation, instance stop. Outside of coregions (see 7.1) and inline expressions (see 7.2) a total ordering of events is assumed along each instance-axis. Within coregions no ordering of events is assumed if no further synchronization constructs in form of general order relations are prescribed.

### 4.3 Message

A message within an MSC is a relation between an output and an input. The output may come from either the environment (through a gate) or an instance, or be **found**; and an input is to either the environment (through a gate) or an instance or is **lost**.

A message exchanged between two instances can be split into two events: the message input and the message output; e.g., the second message in Figure 6(a) can be split into out(m2) (output) and in(m2) (input). In a message parameters may be assigned (see 5.8).

The correspondence between message outputs and message inputs has to be defined uniquely. In the textual representation normally the mapping between inputs and outputs follows from message name identification and address specification. In the graphical representation a message is represented by an arrow.

The loss of a message, i.e., the case where a message is sent but not consumed, is indicated by the keyword **lost** in the textual representation and by a black hole in the graphical representation.

Symmetrically, a spontaneously found message, i.e., a message which appears from nowhere, is defined by the keyword **found** in the textual representation and by a white hole in the graphical representation.

In the textual representation an ordering of message events on different instances may be defined by means of the keyword **before** and **after**. In the graphical representation synchronisation constructs in form of connection lines define these generalized ordering concepts.



The time interval on a timed message gives the delay between the message output and the message input. Also, the message input of an incomplete message input as well as the message output of an incomplete message output can be time constrained. In addition, the message output and the message input event can be time constrained with respect to other events.

### Concrete textual grammar

```

<message event> ::=
    <message output> | <message input>

<message output> ::=
    out <msg identification> to <input address>

<message input> ::=
    in <msg identification> from <output address>

<incomplete message event> ::=
    <incomplete message output> | <incomplete message input>

<incomplete message output> ::=
    out <msg identification> to lost [ <input address> ]

<incomplete message input> ::=
    in <msg identification> from found [ <output address> ]

<msg identification> ::=
    <message name> [ , <message instance name> ]
    [ <left open> <parameter list> <right open> ]

```

The <message instance name> is needed only in the textual notation.

```

<output address> ::=
    <instance name> | { env | <reference identification> } [ via <gate name> ]

<reference identification> ::=
    reference <msc reference identification>
    | inline <inline expr identification>

```

The <gate name> refers to a <def in gate>. If the keyword **env** is used alone it means that the <output address> denotes a <def in gate> which has an implicit name given by the corresponding <msg identification> and direction **in**.

```

<input address> ::=
    <instance name> | { env | <reference identification> } [ via <gate name> ]

```

The <gate name> refers to a <def out gate>. If the keyword **env** is used alone it means that the <input address> denotes a <def out gate> which has an implicit name given by the corresponding <msg identification> and direction **out**.

### Static requirements

For messages exchanged between instances the following rules must hold: To each <message output> one corresponding <message input> has to be specified and vice versa. In case, where the <message name> and the <output address> or <input address> are not sufficient for a unique mapping the <message instance name> has to be employed.

It is not allowed that the <message output> is causally depending on its <message input> via other messages or general ordering constructs. Such causal dependence is the case if the connectivity graph (see Figure 6) contains loops. If a <parameter list> is specified for a <message input> then it has to be specified also for the corresponding <message output> and vice versa. The <parameter list> for <message output> consists only of <expression>s and the <parameter list> for <message input> consists only of <pattern>s.

## Concrete graphical grammar

<message event area> ::=  
    { <message out area> | <message in area> }  
    { *is followed by* <general order area> }\*  
    { *is attached to* <general order area> }\*

Message events may be generally ordered in a number of different general order relations. Message events appear on either side of the order relation.

<message out area> ::=  
    <message out symbol>  
    *is attached to* <instance axis symbol>  
    *is attached to* <message symbol>  
    [ *is attached to*  
      {<int symbol> | <abs time symbol> }\* ]

<message out symbol> ::=  
    <void symbol>

<void symbol> ::= .

The <void symbol> is a geometric point without spatial extension.

The <message out symbol> is actually only a point which is on the instance axis. The end of the message symbol which has no arrow head is also on this point on the instance axis.

<message in area> ::=  
    <message in symbol>  
    *is attached to* <instance axis symbol>  
    *is attached to* <message symbol>  
    [ *is attached to*  
      {<int symbol> | <abs time symbol> }\* ]


<message in symbol> ::=  
    <void symbol>

The <message in symbol> is actually only a point which is on the instance axis. The end of the message symbol which is the arrow head is also pointing on this point on the instance axis.

<message area> ::=  
    <message symbol>  
    *is associated with* <msg identification> [ **time** <time interval> ]  
    *is attached to* { <message start area> | <message end area> }

<message start area> ::=  
    <message out area> | <actual out gate area>  
    | <def in gate area> | <inline gate area>

<message end area> ::=  
    <message in area> | <actual in gate area>  
    | <def out gate area> | <inline gate area>

<message symbol> ::=  
    

The mirror image of the <message symbol> is allowed. The point of the arrow head must be on the instance axis.

In the graphical representation the message instance name is not necessary for a unique syntax description.

<incomplete message area> ::=  
    { <lost message area> | <found message area> }  
    { *is followed by* <general order area> }\*  
    { *is attached to* <general order area> }\*

<lost message area> ::=  
     <lost message symbol> *is associated with* <msg identification>  
     [ *is associated with* { <instance name> | <gate name> } ]  
     *is attached to* <message start area>

<lost message symbol> ::=



The <lost message symbol> describes the event of the output side, i.e., the solid line starts on the <message start area> where the event occurs. The optional intended target of the message can be given by an identifier associated with the symbol. The target identification should be written close to the black circle, while the message identification should be written close to the arrow line.

The mirror image of the symbol may also be used.

<found message area> ::=  
     <found message symbol> *is associated with* <msg identification>  
     [ *is associated with* { <instance name> | <gate name> } ]  
     *is attached to* <message end area>

<found message symbol> ::=



The <found message symbol> describes the event of the input side (the arrowhead) which must be on a <message end area>. The instance or gate which supposedly was the origin of the message is indicated by the optional identification given by the text associated with the circle of the symbol. The message identification should be written close to the arrow line.

The mirror image of the symbol may also be used.

## Static requirements

A <parameter list> of a <message area> where both ends are attached to <instance event area>s consists of <binding>. A <parameter list> of a <message area> where one end is attached to an output event area and the other to a gate or is lost, consists of <expression>s. A <parameter list> of a <message area> where one end is attached to an input event area and the other to a gate or is found, consists of <pattern>s.

## Semantics

For an MSC the message output denotes the message sending, the message-input the message consumption. No special construct is provided for message reception (input into the buffer).

An incomplete message is a message which is either an output (where the input is lost/unknown) or an input (where the output is found/unknown).

For the case where message events coincide with other events, see the drawing rules in 2.4.

## 4.4 Control Flow

MSC may describe control flows of not only via asynchronous messages, but also by means of calls and replies.

A method is a named unit of behaviour inside an instance. A method may be invoked remotely and the results of the calculations of the method may be returned through a reply to the caller. The reply will have the same name as the corresponding call.

A method call may be either asynchronous or synchronizing. An asynchronous call implies that the caller may continue without waiting for the reply of the call. On the other hand, a synchronizing call implies that the caller will enter a suspension region where no events occur until the reply of the call returns. Since method calls may occur inside decomposed instances the methods and suspension regions may be omitted. If the <method symbol> and <suspension symbol> is used the

<method symbol> indicates that an instance is active. The <suspension symbol> indicates that an instance is suspended, typically waiting for some kind of blocking condition to be resolved (e.g., waiting for the reply of a synchronous call) or to get access to some shared resource (e.g., CPU) in order to continue with an ongoing task. The normal instance axis symbol (<instance axis symbol1> or <instance axis symbol2>) in this context means that the instance is inactive and waiting for an in event (<message input> or <call in>) to be activated and start one of the tasks it is capable to perform. The symbol which describes the activation level of an instance does not imply any dynamic effects on the formal semantics, which only considers event ordering, but only poses requirements on where messages may have their in and out events.

Method calls and method replies may also be incomplete such that either the call or the reply gets lost.

Analogous to time constraints for messages, method calls and method replies can be time constrained.

### Concrete Textual Grammar

```

<method call event> ::=
    <call out> | <call in> | <reply out> | <reply in>

<call out> ::=
    call <msg identification> to <input address>

<call in> ::=
    receive <msg identification> from <output address>

<reply out> ::=
    replyout <msg identification> to <input address>

<reply in> ::=
    replyin <msg identification> from <output address>

<incomplete method call event> ::=
    <incomplete call out> | <incomplete call in> |
    <incomplete reply out> | <incomplete reply in>

<incomplete call out> ::=
    call <msg identification> to lost [<input address>]

<incomplete call in> ::=
    receive <msg identification> from found [<output address>]

<incomplete reply out> ::=
    replyout <msg identification> to lost [<input address>]

<incomplete reply in> ::=
    replyin <msg identification> from found [<output address>]

<start method> ::=
    method <end>

<end method> ::=
    endmethod <end>

<start suspension> ::=
    suspension <end>

<end suspension> ::=
    endsuspension <end>

```

### Concrete graphical grammar

```

<method call area> ::=
    <message symbol>
    is associated with <method identification>[time <time interval>]
    is attached to { <method call start area> <method call end area> }
    is attached to { <instance axis symbol> }
    [is attached to {<method area>}]

```

<method identification> ::=  
     **call** <msg identification>

<method call start area> ::=  
     <call out area> | <actual out gate area> |  
     <def in gate area> | <inline gate area>  
     **is attached to** <instance axis symbol>  
     **is attached to** <message symbol>  
     [**is attached to** <suspension symbol>]  
     [**is attached to**  
     {<int symbol> | <abs time symbol> }\*]

<method call end area> ::=  
     <call in area> | <actual in gate area> |  
     <def out gate area> | <inline gate area>  
     **is attached to** <instance axis symbol>  
     **is attached to** <message symbol>  
     [**is attached to** <method symbol>]  
     [**is attached to**  
     {<int symbol> | <abs time symbol> }\*]

<reply area> ::=  
     <reply symbol>  
     **is associated with** <msg identification>[**time** <time interval>]  
     **is attached to** { <reply start area> <reply end area> }

<reply start area> ::=  
     <reply out area> | <actual out gate area> |  
     <def in gate area> | <inline gate area>  
     **is attached to** <instance axis symbol>  
     **is attached to** <reply symbol>  
     [**is attached to** <method symbol>]  
     [**is attached to**  
     {<int symbol> | <abs time symbol> }\*]

<reply end area> ::=  
     <reply in area> | <actual in gate area> |  
     <def out gate area> | <inline gate area>  
     **is attached to** <instance axis symbol>  
     **is attached to** <reply symbol>  
     [**is attached to** <suspension symbol>]  
     [**is attached to**  
     {<int symbol> | <abs time symbol> }\*]

<reply symbol> ::=  
     -----▶

<incomplete method call area> ::=  
     { <lost method call area> | <found method call area> }  
     { **is followed by** <general order area> }\*  
     { **is attached to** <general order area> }\*

<lost method call area> ::=  
     <lost message symbol> **is associated with** <method identification>  
     [ **is associated with** { <instance name> | <gate name> } ]  
     **is attached to** <method call start area>

<found method call area> ::=  
     <found message symbol> **is associated with** <method identification>  
     [ **is associated with** { <instance name> | <gate name> } ]  
     **is attached to** <method call end area>

<incomplete reply area> ::=  
     { <lost reply area> | <found reply area> }  
     { **is followed by** <general order area> }\*  
     { **is attached to** <general order area> }\*

<lost reply area> ::=  
 <lost reply symbol> *is associated with* <msg identification>  
 [ *is associated with* { <instance name> | <gate name> } ]  
*is attached to* <reply start area>

<lost reply symbol> ::=  
 --▶●

<found reply area> ::=  
 <found reply symbol> *is associated with* <msg identification>  
 [ *is associated with* { <instance name> | <gate name> } ]  
*is attached to* <reply end area>

<found reply symbol> ::=  
 ◀--○

<method call event area> ::=  
 {<call out area> | <call in area>}  
 {*is followed by* <general order area> }\*  
 {*is attached to* <general order area> }\*

<call out area> ::=  
 <call out symbol>  
*is attached to* <instance axis symbol>  
*is attached to* <message symbol>

<call out symbol> ::=  
 <void symbol>

<call in area> ::=  
 <call in symbol>  
*is attached to* <instance axis symbol>  
*is attached to* <message symbol>

<call in symbol> ::=  
 <void symbol>

<reply event area> ::=  
 {<reply out area> | <reply in area>}  
 {*is followed by* <general order area> }\*  
 {*is attached to* <general order area> }\*

<reply out area> ::=  
 <reply out symbol>  
*is attached to* <instance axis symbol>  
*is attached to* <message symbol>

<reply out symbol> ::=  
 <void symbol>


<reply in area> ::=  
 <reply in symbol>  
*is attached to* <instance axis symbol>  
*is attached to* <message symbol>

<reply in symbol> ::=  
 <void symbol>

<method area> ::=  
 <method symbol>  
*is attached to* <instance axis symbol>  
 [*contains* <method event layer>]

<method event layer> ::=  
 <method event area> | <method event area> *above* <method event layer>

<method event area> ::=  
     <event area>

<method symbol> ::=  



<suspension area> ::=  
     <suspension symbol>  
     *is attached to* <instance axis symbol>  
     [*contains* <suspension event layer>]

<suspension event layer> ::=  
     <method invocation area>  
     | <method invocation area> *above* <suspension event layer>

<method invocation area> ::=  
     <method start area>  
     *is followed by* <method area>  
     *is followed by* <method end area>

<method start area> ::=  
     <call in area> | <found method call area>

<method end area> ::=  
     <reply out area> | <lost reply area>

<suspension symbol> ::=  


## Static requirements

The following grammatical rules refer to events on a specific instance:

<start method> <instance event list> <end method>

<start suspension> <instance event list> <end suspension>

<call out> <instance event list> <reply in>, where the <reply in> corresponds to the <call out>.

<call in><instance event list><reply out>, where <reply out> corresponds to <call in>.

A <start method> may only follow directly after a <call in> event, <incomplete call in> event, <message input> event, <incomplete message input> event, <end suspension> or a <create> event where the created instance is subject to the <start method>. A method symbol may end at any time, but an <end method> must always follow directly after a <reply out> or an <incomplete reply out> event.

Dynamically call and reply (if any) must come in pairs. A specific instance that emits a <call out> event may not emit another call out event before a corresponding <reply in> event is received or an <call in> event is received that is causally dependent upon its <call out> event (i.e., calls and replies may be nested). If a suspension region starts directly after an outgoing call it ends with the corresponding reply.

All <instance event>s are allowed on method symbols. Methods may be superimposed on methods and suspension regions. By "superimposed" we mean that the regions may be drawn with a small horizontal offset relative to the method or suspension region in which it is contained.

Suspension regions are not allowed to contain any events. A suspension region may however be superimposed by a method region when the suspension region starts with a <call out> and consequently is called recursively with a <call in> (directly or indirectly).

A suspension region may end at any time if it does not follow directly after a <call out> event, in which case it ends with the corresponding <reply in> event.<sup>1</sup>

#### 4.5 Environment and gates

The gates represent the interface between the MSC and its environment. Any message or order relation attached to the MSC frame constitutes a gate. See Figure 7 for examples of gates.

A message gate always has a name. The name can be defined explicitly by a name associated with the gate on the frame. Otherwise the name is given implicitly by the direction of the message through the gate and the message name, e.g., "in\_X" for a gate receiving a message X from its environment.

The message gates are used when references to the MSC are put in a wider context in another MSC. The actual gates on the MSC reference are then connected to other message gates or instances. Similar to gate definitions, actual gates may have explicit or implicit names.

Order gates represent incomplete order relations where an event inside the MSC will be ordered relative to an event in the environment. Order gates are always explicitly named. Order gates are considered to have direction – from the event ordered first to the event coming after it.

Also order gates are used on references to MSCs in other MSCs. The actual order gates of the MSC reference are connected to other order gates or to events.

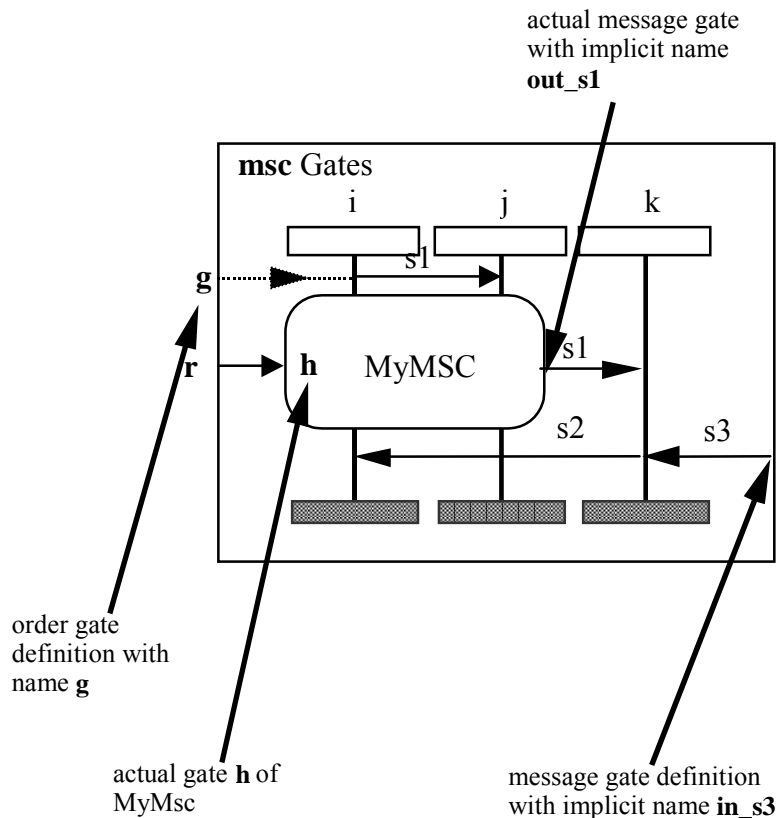
Create gates represent the possibility to divide the creation event from the created instance.

Gates on inline expressions are similar to gates on MSC frames and MSC references, but the difference lies in the fact that the inline expression frame is both the frame of gate definition (on the inside) and the symbol of actual gate use (on the outside). In the case where the inline actual gate is directly drawn to another frame (inline expression or MSC diagram frame), the arrowheads on the intersection points can be omitted to prevent cluttering the diagram with very tight arrowheads.

---

<sup>1</sup> Additional static requirements may be *added* to the static requirements to fit a specific execution model or come out as a consequence when that model is traced. For example if the control flow of a purely procedural language is described (i.e., no parallelism), then a <call out> event must always be followed by a <start suspension> or if each instance always is able to handle events (i.e., has its own CPU), then a suspension region may only start after a <call out> event.





**Figure 7 – Example of gates**

**Concrete textual grammar**

```

<actual out gate> ::=
    [ <gate name> ] out <msg identification> to <input dest>

<actual in gate> ::=
    [ <gate name> ] in <msg identification> from <output dest>

<input dest> ::=
    lost [ <input address> ] | <input address>

<output dest> ::=
    found [ <output address> ] | <output address>

<def in gate> ::=
    [ <gate name> ] out <msg identification> to <input dest>

<def out gate> ::=
    [ <gate name> ] in <msg identification> from <output dest>

```

If the <gate name> is omitted, an implicit name given by direction and corresponding <msg identification> is assumed.

```

<actual order out gate> ::=
    <gate name> before <order dest>

<order dest> ::=
    <event name> | { env | <reference identification> } via <gate name>

```

The <event name> refers to an orderable event. The <gate name> refers to either a <def order out gate>, <actual order in gate>, <inline order out gate> or <inline order in gate>.

```

<actual order in gate> ::=
    <gate name>
    [ after <order dest list> ]

```

<def order in gate> ::=  
    <gate name> **before** <order dest>

The first <gate name> defines the name of this order gate.

<def order out gate> ::=  
    <gate name>  
    [ **after** <order dest list> ]

<actual create out gate> ::=  
    **create out** <create gate identification> **create** <create target>

<actual create in gate> ::=  
    **create in** <create gate identification>

<create target> ::=  
    <instance name> | { **env** | <reference identification> } [ **via** <gate name> ]

<def create in gate> ::=  
    **create out** [ <create gate identification> ] **create** <create target>

<def create out gate> ::=  
    **create in** <create gate identification>

If the <gate name> is omitted, an implicit name given by direction and corresponding <msg identification> is assumed. The <gate name> defines the name of this order gate.

<inline out gate> ::=  
    <def out gate>  
    [ **external out** <msg identification> **to** <input dest>]

<inline in gate> ::=  
    <def in gate>  
    [ **external in** <msg identification> **from** <output dest>]

<inline out call gate> ::=  
    <def out call gate>  
    [ **external call** <msg identification> **to** <input dest>]

<inline in call gate> ::=  
    <def in call gate>  
    [ **external receive** <msg identification> **from** <output dest>]

<inline out reply gate> ::=  
    <def out reply gate>  
    [ **external replyout** <msg identification> **to** <input dest>]

<inline in reply gate> ::=  
    <def in reply gate>  
    [ **external replyin** <msg identification> **from** <output dest>]

<inline create out gate> ::=  
    <def create out gate>  
    [ **external** <create>]

<inline create in gate> ::=  
    <def create in gate>  
    [ **external create from** <create source>]

<create source> ::=  
    <instance name> |  
    { **env** | <reference identification> } [ **via** <create gate identification>]

<inline order out gate> ::=  
    <gate name>  
    [ [ **after** <order dest list> ] **external before** <order dest> ]

<inline order in gate> ::=  
    <gate name> **before** <order dest>  
    [ **external** [ **after** <order dest list> ] ]

```

<actual out call gate> ::=
    [ <gate name> ] call <msg identification> to <input dest>

<actual in call gate> ::=
    [ <gate name> ] receive <msg identification> from <output dest>

<def in call gate> ::=
    [ <gate name> ] call <msg identification> to <input dest>

<def out call gate> ::=
    [ <gate name> ] receive <msg identification> from <output dest>

```

If the <gate name> is omitted, an implicit name given by direction and corresponding <msg identification> is assumed.

```

<actual out reply gate> ::=
    [ <gate name> ] replyout <msg identification> to <input dest>

<actual in reply gate> ::=
    [ <gate name> ] replyin <msg identification> from <output dest>

<def in reply gate> ::=
    [ <gate name> ] replyout <msg identification> to <input dest>

<def out reply gate> ::=
    [ <gate name> ] replyin <msg identification> from <output dest>

```

If the <gate name> is omitted, an implicit name given by direction and corresponding <msg identification> is assumed.

## Concrete graphical grammar

```

<inline gate area> ::=
    { <inline out gate area> | <inline in gate area> |
      <inline create out gate area> | <inline create in gate area> |
      <inline out call gate area> | <inline in call gate area> |
      <inline out reply gate area> | <inline in reply gate area> }
    [ is associated with <gate identification> ]

<inline out gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to { <message symbol> | <found message symbol> } ]
    [ is attached to { <message symbol> | <lost message symbol> } ]

<inline in gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to { <message symbol> | <lost message symbol> } ]
    [ is attached to { <message symbol> | <found message symbol> } ]

<inline out call gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to { <message symbol> | <found message symbol> } ]
    [ is attached to { <message symbol> | <lost message symbol> } ]

<inline in call gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to { <message symbol> | <lost message symbol> } ]
    [ is attached to { <message symbol> | <found message symbol> } ]

<inline out reply gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to { <reply symbol> | <found reply symbol> } ]
    [ is attached to { <reply symbol> | <lost reply symbol> } ]

```

```

<inline in reply gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to { <reply symbol> | <lost reply symbol> } ]
    [ is attached to { <reply symbol> | <found reply symbol> } ]

<inline create out gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to <createline symbol> ]
    [ is attached to <createline symbol> ]

<inline create in gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    [ is attached to <createline symbol> ]
    [ is attached to <createline symbol> ]

```

An inline expression gate is normally attached to one message symbol inside the inline expression frame, and a message symbol outside the inline expression frame. When there is no symbol attached on the inside of the gate, this means that there is an incomplete message which is associated with the gate.

```

<inline order gate area> ::=
    <inline order out gate area> | <inline order in gate area>

<inline order out gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    is attached to <general order symbol>
    [ is attached to <general order symbol> ]

```

The first <general order symbol> is a general ordering relation inside the inline expression such that the event inside the expression is before the gate. The optional <general order symbol> refers to a general order relation attached to the gate outside the inline expression.

```

<inline order in gate area> ::=
    <void symbol>
    is attached to <inline expression symbol>
    is attached to <general order symbol>
    [ is attached to <general order symbol> ]

```

The first <general order symbol> is a general ordering relation inside the inline expression such that the gate is before the event inside the expression. The optional <general order symbol> refers to a general order relation attached to the gate outside the inline expression.

```

<def gate area> ::=
    { <def out gate area> | <def in gate area> |
      <def order out gate area> | <def order in gate area>
      <def out call gate area> | <def in call gate area>
      <def out reply gate area> | <def in reply gate area>
      <def create out gate area> | <def create in gate area> }
    is attached to <msc symbol>

```

```

<def out gate area> ::=
    <void symbol> [ is associated with <gate identification> ]
    is attached to { <message symbol> | <found message symbol> }

```

The <message symbol> or <found message symbol> must have its arrow head end attached to the <def out gate area>.

```

<def in gate area> ::=
    <void symbol> [ is associated with <gate identification> ]
    is attached to { <message symbol> | <lost message symbol> }

```

The <message symbol> or <lost message symbol> must have its open end attached to the <def in gate area>.

```
<def out call gate area> ::=
    <void symbol> [ is associated with <gate identification> ]
    is attached to { <message symbol> | <found message symbol> }
```

The <message symbol> or <found message symbol> must have its arrow head end attached to the <def out call gate area>.

```
<def in call gate area> ::=
    <void symbol> [is associated with <gate identification>]
    is attached to { <message symbol> | <lost message symbol> }
```

The <message symbol> or <lost message symbol> must have its open end attached to the <def in call gate area>.

```
<def out reply gate area> ::=
    <void symbol> [ is associated with <gate identification> ]
    is attached to { <reply symbol> | <found reply symbol> }
```

The <reply symbol> or <found reply symbol> must have its arrow head end attached to the <def out reply gate area>.

```
<def in reply gate area> ::=
    <void symbol> [is associated with <gate identification>]
    is attached to { <reply symbol> | <lost reply symbol> }
```

The <reply symbol> or <lost reply symbol> must have its open end attached to the <def in reply gate area>.

```
<def order out gate area> ::=
    <void symbol> [is associated with <gate identification> ]
    is attached to <general order area>
```

```
<def order in gate area> ::=
    <void symbol> [is associated with <gate identification> ]
    is followed by <general order area>
```

```
<def create out gate area> ::=
    <void symbol> [ is associated with <create gate identification> ]
    is attached to <createline symbol>
```

The <createline symbol> must have its arrow head end attached to the <def create out gate area>.

```
<def create in gate area> ::=
    <void symbol> [is associated with <create gate identification>]
    is attached to <createline symbol>
```

The <createline symbol> must have its open end attached to the <def create in gate area>.

```
<gate identification> ::=
    <gate name>
```

```
<create gate identification> ::=
    [<gate identification> : ] <instance kind>
```

The <instance kind> refers to the instance kind of the instance to be created. The instance kind may also correspond to a singular <instance name>.

```
<actual gate area> ::=
    <actual out gate area> | <actual in gate area> |
    <actual out call gate area> | <actual in call gate area> |
    <actual out reply gate area> | <actual in reply gate area> |
    <actual create out gate area> | <actual create in gate area> |
    <actual order out gate area> | <actual order in gate area>
```

<actual out gate area> ::=  
     <void symbol> [*is associated with* <gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* { <message symbol> | <lost message symbol> } ]

The <actual out gate area> is attached to the open end of the <message symbol> or <lost message symbol>.

<actual in gate area> ::=  
     <void symbol> [ *is associated with* <gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* { <message symbol> | <found message symbol> } ]

The <actual in gate area> is attached to the arrow head end of the <message symbol> or <found message symbol>.

<actual out call gate area> ::=  
     <void symbol> [*is associated with* <gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* { <message symbol> | <lost message symbol> } ]

The <actual out call gate area> is attached to the open end of the <message symbol> or <lost message symbol>.

<actual in call gate area> ::=  
     <void symbol> [ *is associated with* <gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* { <message symbol> | <found message symbol> } ]

The <actual in call gate area> is attached to the arrow head end of the <message symbol> or <found message symbol>.

<actual out reply gate area> ::=  
     <void symbol> [*is associated with* <gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* { <reply symbol> | <lost reply symbol> } ]

The <actual out reply gate area> is attached to the open end of the <reply symbol> or <lost reply symbol>.

<actual in reply gate area> ::=  
     <void symbol> [ *is associated with* <gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* { <reply symbol> | <found reply symbol> } ]

The <actual in reply gate area> is attached to the arrow head end of the <reply symbol> or <found reply symbol>.

<actual create out gate area> ::=  
     <void symbol> [*is associated with* <create gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* <createline symbol> ]

The <actual create out gate area> is attached to the open end of the <createline symbol>.

<actual create in gate area> ::=  
     <void symbol> [ *is associated with* <create gate identification>]  
     *is attached to* <msc reference symbol>  
     [ *is attached to* <createline symbol> ]

The <actual create in gate area> is attached to the arrow head end of the <createline symbol>.

<actual order out gate area> ::=  
     <void symbol> [ *is associated with* <gate identification> ]  
     *is attached to* <msc reference symbol>  
     *is followed by* <general order area>

<actual order in gate area> ::=  
     <void symbol>[ *is associated* with <gate identification> ]  
     *is attached to* <msc reference symbol>  
     *is attached to* <general order area>

### Static requirements

Defined <gate name>s of an MSC are allowed to be ambiguous, but references to ambiguous <gate name>s are illegal.

For gate connections the associated gate definition must correspond with the actual gate. Correspondence means for messages that the message attached to the gate definition is of the same type as the message attached to the actual gate. The directions should also correspond.

The kind of the gate definition must be the same as that of the connected actual gate. The kinds of gates are message gates, ordering gates, and create-gates.

## 4.6 General ordering

General ordering is used to impose additional orderings upon events that are not defined by the normal ordering given by the MSC semantics. For example, to specify that an event on one instance must happen before an otherwise unrelated event on another instance, or to specify orderings on events contained within a coregion.

### Concrete textual grammar

The textual grammar is given in 4.1.

### Concrete graphical grammar

<general order area> ::=  
     <general order symbol>  
     *is attached to* <ordered event area>

<general order symbol> ::=  
     <general order symbol1> | <general order symbol2>

<general order symbol1> ::=

⋮

The <general order symbol1> is allowed to have a staircase like shape. There cannot be both upwards and downwards steps on the same ordering. This means that it may consist of consecutive vertical and horizontal segments. Segments of <general order symbol1> may overlap, but this is only allowed if it is unambiguous which <general order symbol1>s are involved. That means that no new orders may be implied.

The <general order symbol1> may only occur inside an <instance axis symbol2> (column form). The connection lines from the <general order symbol1> to the <ordered event area>s that it orders must be horizontal.

<general order symbol2> ::=

⋮

▼

⋮

The <general order symbol2> may have any orientation and also be curved.

```

<ordered event area> ::=
    | <actual order in gate area>
    | <actual order out gate area>
    | <def order in gate area>
    | <def order out gate area>
    | <inline order gate area>
    | <message event area>
    | <incomplete message area>
    | <method call event area>
    | <incomplete method call area>
    | <reply event area>
    | <incomplete reply area>
    | <timer area>
    | <create area>
    | <action symbol>

```

### Static requirements

The partial order on events defined by the general ordering constructs and the messages must be irreflexive.

### Semantics

The general order symbols describe orders between events which otherwise would not be ordered. This is particularly useful when events in a coregion should be more ordered than all possible permutations.

In the graphical notation when using the <general order symbol1>, the event that is graphically higher must occur before the lower event. When using the <general order symbol2> the arrow points from one event to another event that comes after.

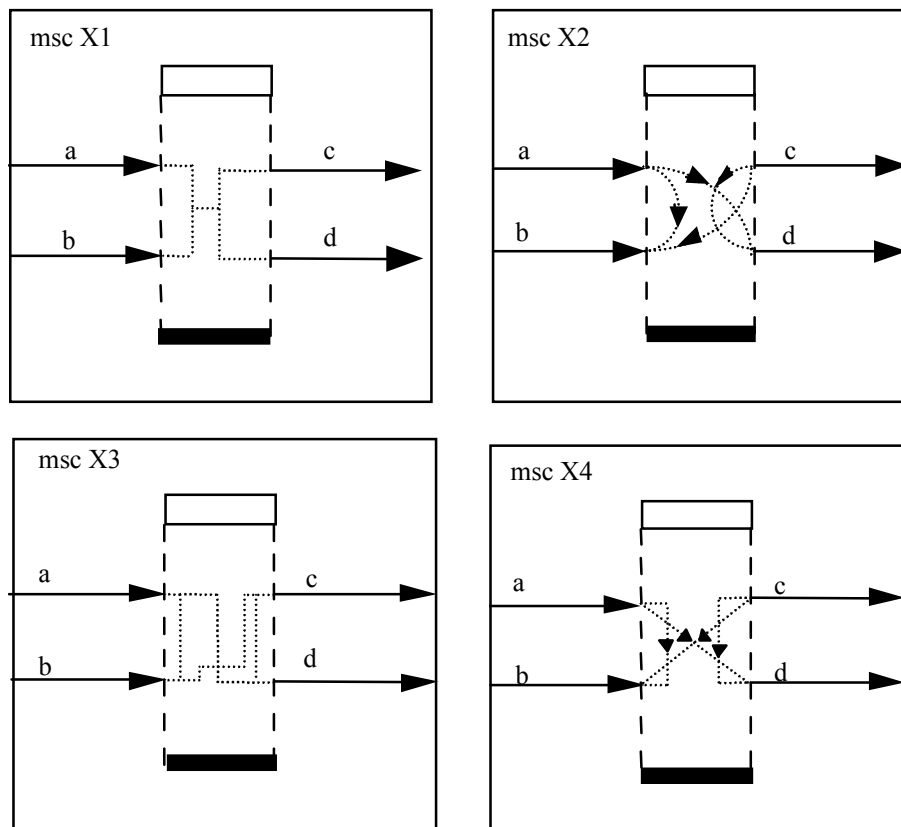
The textual grammar defines the partial order relations by the keywords **before** and **after**. They indicate directly in what order the involved events must come in the legal traces. Timing relations are defined by the keyword **time**. They indicate in what time quantified order the involved events must come in the legal traces.

### Example of general ordering

Figure 8 shows four examples which describe the same general ordering. The following orders are implied:

a **before** b, a **before** d, c **before** b, c **before** d. a and c are unordered and b and d are unordered.





**Figure 8 – Example for general ordering**

#### 4.7 Condition

Conditions can be used to restrict the traces that an MSC can take, or the way in which they are composed into High-Level MSCs. There are two types of condition: *setting* and *guarding* conditions. Setting conditions set or describe the current global system state (global condition), or some non-global state (nonglobal condition). In the latter case the condition may be local, i.e., attached to just one instance.

Guarding conditions restrict the behaviour of an MSC by only allowing the execution of events in a certain part of the MSC depending on their values. This part, which is the scope of the condition, is either the complete MSC or the innermost operand of an inline expression or a branch of an HMSC. The guard must be placed at the beginning of this scope. The condition is either a (global or non-global) state, which the system should be in (as defined by prior setting conditions), or a Boolean expression in the data language.

In the textual representation the condition has to be defined for each instance to which it is attached using the keyword **condition** together with the condition name. If the condition refers to more than one instance then the keyword **shared** together with the list of instances that the condition is referred on must be added. A global condition, that is, one referring to all instances, may be defined by means of the keywords **shared all**. Guarding conditions are shown with the keywords **condition** and **when**, setting conditions only with the keyword **condition**. In the graphical syntax, the keyword **when** is included in the condition symbol for guarding conditions; for setting conditions there is no extra keyword or symbol.

Guarding conditions can be used for example in inline expressions (see 7.2) to determine which operand of an **alt** expression is applicable depending on choices that have been made in previous **alt** expressions, or in HMSCs (see 7.5).

## Concrete textual grammar

```
<shared condition> ::=
    [<shared>] <condition identification> <shared> <end>

<condition identification> ::=
    condition <condition text>

<condition text> ::=
    <condition name list>
    | when { <condition name list> | <left open> <expression> <right open> }
    | otherwise

<condition name list> ::=
    <condition name> { , <condition name> } *

<shared> ::=
    shared { [ <shared instance list> ] | all }

<shared instance list> ::=
    <instance name> [ , <shared instance list> ]

<condition> ::=
    [<shared>] <condition identification> <end>
```

The optional <shared> before <condition identification> in the concrete textual grammar is analogous to the optional <shared> in the concrete graphical grammar (see below).

## Static requirements

For each <instance name> contained in a <shared instance list> of a <condition>, an instance with a corresponding shared <condition> must be specified. If instance *b* is contained in the <shared instance list> of a shared <condition> attached to instance *a* then instance *a* must be contained in the <shared instance list> of the corresponding shared <condition> attached to instance *b*. If instance *a* and instance *b* share the same <condition> then for each message exchanged between these instances, the <message input> and <message output> must be placed either both before or both after the <condition>.

If two conditions are ordered directly (because they have an instance in common) or ordered indirectly via conditions on other instances, this order must be respected on all instances that share these two conditions. According to the possibilities of an *instance oriented* and an *event oriented* syntax description (see 4.1) there are two syntax forms for conditions: within the *instance oriented* description the <shared condition> form is used whereas in the *event oriented* description a multi instance form is used employing a <multi instance event list> followed by a colon and the non-terminal <condition>. (see 4.1). In the <shared condition> form the keyword **shared** is used consistently also for local conditions, therefore within the non-terminal <shared> the <shared instance list> is optional.

**Otherwise** can only occur as the guard of exactly one operand of an alternative expression.

## Concrete graphical grammar

```
<condition area> ::=
    <condition symbol> contains <condition text> [<shared>]
    is attached to { <instance axis symbol>* } set

<condition symbol> ::=
```



The <condition area> may refer to just one instance, or is attached to several instances. If a shared <condition> crosses an <instance axis symbol> which is not involved in this condition the <instance axis symbol> is drawn through:



The instances covered by a condition consist of those attached to it in the diagram plus those listed explicitly in the <shared> clause.

### Static requirements

A guarding condition must be placed at the beginning of its scope, where a scope is either a whole MSC, an operand of an inline expression, or a branch of an HMSC.

A guarding condition should always cover all ready instances of its scope, where an instance is ready if it contains an event that may be executed before any other event in the scope.

If a guard contains a data expression, then this expression must be of type Boolean. If this expression furthermore contains dynamic variables, it may only cover a single instance, which thus must be the only ready instance of the scope.

An instance that is graphically covered by a condition symbol cannot be included also via its <shared> list. If there are any ambiguities regarding event ordering on <shared> instances due to the implicit condition symbol, they must be represented in the diagram explicitly with an <instance area> rather than via the <shared> list.

### Semantics

Setting conditions define the actual system state of the instance(s) that share the condition. Guarding conditions can be used to restrict the possible ways in which an MSC can be continued.

The events in the scope guarded by a guarding condition can only be executed if the guarding condition is true at the time the first such event is executed. If the condition is an expression in the data language, this means that it evaluates to 'true'. If the guard is a set of condition names, the *last* setting condition on that set of instances must have a non-empty intersection with the guard. Only setting conditions on exactly the same instances are checked; conditions that set the state of a subset or superset of these instances do not.

Specific interpretations of guards in inline expressions are given in 7.2. The **otherwise** guard is *true* if guards of all other operands of the alternative expression are *false*.

## 4.8 Timer

In MSCs either the setting of a timer and a subsequent time-out due to timer expiration or the setting of a timer and a subsequent timer stop may be specified. In addition, the individual timer constructs – timer setting, stop/time-out – may be used separately, e.g., in case where timer expiration or time supervision is split between different MSCs. In the graphical representation the start symbol has the form of an hour glass connected with the instance axis by a line symbol. Time-out is described by a message arrow pointing at the instance which is attached to the hour glass symbol. The stop symbol has the form of a cross symbol which is connected with the instance by a line.

The specification of timer instance name and timer duration is optional both in the textual and graphical representation.

A timer can be started with or without a duration. For the duration, there is the possibility to express a lower and upper bound for the timeout to occur. The upper bound for a timeout can be defined to be infinity which is represented by the keyword **inf**. The start, stop and timeout events of a timer can be time constrained in addition. This timing is then represented in the **time** relation of orderable events.

The timeout period of a timer is defined as follows:

[0, <b>inf</b> ),	if no duration is given
[< duration min >, <b>inf</b> ),	if only the minimal duration is given
[0, < duration max >],	if only the maximal duration is given
[< duration min >, < duration max >],	if minimal and maximal duration are given

### Concrete textual grammar

```

<timer statement> ::=
    <starttimer> | <stoptimer> | <timeout>

<starttimer> ::=
    starttimer <timer name> [ , <timer instance name> ]
    [ <duration> ] [ <left open> <parameter list> <right open> ]

<duration> ::=
    <left square bracket>
    [ <min durationlimit> ] [ , <max durationlimit> ] <right square bracket>

<durationlimit> ::=
    <expression string> | inf

<stoptimer> ::=
    stoptimer <timer name> [ , <timer instance name> ]

<timeout> ::=
    timeout <timer name> [ , <timer instance name> ]
    [ <left open> <parameter list> <right open> ]

```

Where the <timer name> is not sufficient for a unique mapping the <timer instance name> must be employed.

### Static requirements

The <parameter list> for <starttimer> consists only of <expression>s and the <parameter list> for <timeout> consists only of <pattern>s. The <expression string>s used in specifying a <duration> must have the assumed type Time.

### Concrete graphical grammar

```

<timer area> ::=
    { <timer start area> | <timer stop area> | <timeout area> }
    { is followed by <general order area> } *
    { is attached to <general order area> } *

<timer start area> ::=
    <timer start area1> | <timer start area2>

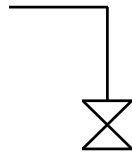
<timer start area1> ::=
    <timer start symbol> is associated with <timer name>
    [ <duration> ] [ <left open> <parameter list> <right open> ]
    is attached to <instance axis symbol>
    [ is attached to
      { <int symbol> | <abs time symbol> } * ]
    [is attached to { <restart symbol> | <timer stop symbol2>
      | <timeout symbol3> } ]

```

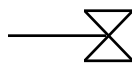
<timer start area2> ::=  
 <restart symbol> *is associated with* <timer name>  
 [ <duration> ] [ <left open> <parameter list> <right open> ]  
*is attached to* <instance axis symbol>  
 [ *is attached to*  
 {<int symbol> | <abs time symbol> }\*]  
*is attached to* <timer start symbol>  
 [ *is attached to* { <timer stop symbol2> | <timeout symbol3> } ]

<timer start symbol> ::=  
 <start symbol1> | <start symbol2>

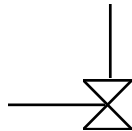
<start symbol1> ::=



<start symbol2> ::=



<restart symbol> ::=



<timer stop area> ::=

<timer stop area1> | <timer stop area2>

<timer stop area1> ::=

<timer stop symbol1> *is associated with* <timer name>  
*is attached to* <instance axis symbol>  
 [ *is attached to*  
 {<int symbol> | <abs time symbol> }\*]

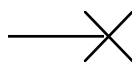
<timer stop area2> ::=

<timer stop symbol2> *is associated with* <timer name>  
*is attached to* <instance axis symbol>  
 [ *is attached to*  
 {<int symbol> | <abs time symbol> }\*]  
*is attached to* { <timer start symbol> | <restart symbol> }

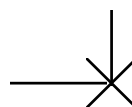
<timer stop symbol> ::=

<timer stop symbol1> | <timer stop symbol2>

<timer stop symbol1> ::=



<timer stop symbol2> ::=



<timeout area> ::=

<timeout area1> | <timeout area2>

<timeout area1> ::=

<timeout symbol> *is associated with* <timer name>  
 [ <left open> <parameter list> <right open> ]  
*is attached to* <instance axis symbol>  
 [ *is attached to*  
 {<int symbol> | <abs time symbol> }\*]



## Concrete graphical grammar

<action area> ::=

<action symbol>  
*is attached to* <instance axis symbol>  
[ *top or bottom is attached to top or bottom*  
{ {<int symbol> | <abs time symbol> }\* } *set*]  
{ *is followed by* <general order area> }\*  
{ *is attached to* <general order area> }\*  
*contains* <action statement>

<action symbol> ::=



In case where the instance axis has the column form, the width of the <action symbol> must coincide with the width of the column.

## Semantics

An action describes an internal atomic activity of an instance. When an action contains data statements, the event modifies the state through evaluating each statement concurrently. This concurrency reflects the atomicity of an action.

### 4.10 Instance creation

Analogously to SDL, creation and termination of instances may be specified within MSCs. An instance may be created by another instance. No message events before the creation can be attached to the created instance.

## Concrete textual grammar

<create> ::=

**create** <instance name> [ <left open> <parameter list> <right open> ]

For each <create> there must be a corresponding instance with the specified name. An instance can be created only once, i.e., within one *simple* MSC, two or more <create>s with the same name must not appear.

## Concrete graphical grammar

<create area> ::=

<createline symbol> [ *is associated with* <parameter list> ]  
*is attached to*  
{ {<instance axis symbol> | <def create in gate area> |  
<actual create out gate area> }  
{ <instance head area> | <def create out gate area> |  
<actual create in gate area> } } *set*  
{ *is followed by* <general order area> }\*  
{ *is attached to* <general order area> }\*  
[*is attached to*  
{ {<int symbol> | <abs time symbol> }\* } *set*]

The creation event is depicted by the end of the <createline symbol> that has no arrowhead. The creation event is attached to an instance axis. If the <create area> is generally ordered, this ordering applies to the creation event. If a time constraint is assigned, it applies to the creation event. The arrowhead points to the <instance head symbol> of the created instance.

<createline symbol> ::=

— — — ➔

The mirror image of the <createline symbol> is allowed.

## Semantics

Create defines the dynamic creation of an instance by another. Dynamically there can be only one creation in the life of an instance and no events on the instance may take place before its creation.

For the case where the instance creation coincides with other events, see the drawing rules in 2.4

### 4.11 Instance stop

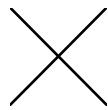
The instance stop is the counterpart to the instance creation, except that an instance can only stop itself whereas an instance is created by another instance.

#### Concrete textual grammar

`<stop> ::=`  
`stop <end>`

#### Concrete graphical grammar

`<stop symbol> ::=`



## Semantics

The stop at the end of an instance represents the termination of this instance.

Dynamically there can be only one stop event in the life of an instance and no events may take place after the instance stop.

## 5 Data concepts

### 5.1 Introduction

Data is incorporated into MSC into a number of places, such as message passing, action boxes, and MSC references. Data is used in two distinguishable ways: statically, such as in the parameterisation of an MSC diagram, or dynamically, such as in the acquisition of a value through a message receipt. To enable MSC to be used with data languages of the user's choice, no MSC specific language is defined. Instead MSC has a number of points in the definition where a string representing some aspect of data usage is required, such as expressions or type declarations. In order to define the semantics of MSC with data a number of functions are required that extract the required information from such strings to interface to the MSC data concepts.

In the absence of an explicit binding to a data language, MSC takes as default the binding to SDL data provided by ITU-T Z.121.

### 5.2 Syntax interface to external data languages

Where a terminal string to the data language is used, the production name is prefixed with underlined words suggestive of its actual interface meaning. For example, `<variable string>` is a terminal string that is to be parsed in the data language as a variable. The list of terminal data strings is:

- `<variable string>`,
- `<type ref string>`,
- `<data definition string>`,
- `<expression string>`.



In order that data be distinguishable in MSC, parenthesis delimiters and escape characters are defined. The actual syntax for data strings are given in 2.1. The idea is that data strings in the data language will be recognizable without excessive use of escape sequences and special delimiters.

The syntactic interface gives instructions to how an MSC lexical analyzer should extract data strings from the MSC notation as a whole. The lexical analyzer is given the start of a data string and applies the parenthesis declarations to determine when to terminate the string and pass it to the data language analyzer that is external to the MSC interpretation as such.

There are 3 types of parentheses and one global escape mechanism. See 2.1 for closer syntactic explanation of the four concepts. Each of the three parenthesis mechanisms permit the declaration of native escape characters from the data language, to minimise the need to use the global escape mechanism. That is, it is possible for data strings to contain native escape sequences that are taken into account in delimiting a data <string> as an MSC lexical unit.

A nestable parenthesis is where the string contained between the left and right parentheses may also contain parentheses that should be recognized. The parenthesizing must be balanced to be well formed, as defined in 2.1.

The equal parenthesis uses the same delimiter on left and right side of the demarked substring. Thus such parentheses cannot be nested since the left and right delimiters cannot be distinguished. String and character quotes are often of this form.

Finally there is the non-nestable parenthesis where the insides should simply be ignored. The lexical analyzer will search for the matching right parenthesis characters. Comment delimiters are often of this form.

The escape mechanism is interpreted such that the string following the escape character should be considered to constitute the insides of a parenthesis and not a delimiter or escape character.

### Concrete textual grammar

```

<parenthesis declaration> ::=
    parenthesis <par decl list> <end>

<par decl list> ::=
    { <nestable par pair> | <non nestable par pair> | <equal par decl> | <escape decl> }
    [ <par decl list> ]

<nestable par pair> ::=
    nestable <pair par list> <end>

<non nestable par pair> ::=
    nonnestable <pair par list> <end>

<equal par decl> ::=
    equalpar <equal par list> <end>

<escape decl> ::=
    escape <escapechar>

<pair par list> ::=
    <pair par> [ <escape decl> ] [ , <pair par list> ]

<pair par> ::=
    <delim> <open par> <delim> <close par> <delim>

<equal par list> ::=
    <equal par item> [ <escape decl> ] [ , <equal par list> ]

<equal par item> ::=
    <delim> <equal par> <delim>

```

```

<delim> ::=
    | <apostrophe>
    | <alphanumeric>
    | <other character>
    | <special>
    | <full stop>
    | <underline>

<par> ::=
    <character string>

<escapechar> ::=
    <delim> <character string> <delim>

```

## Static requirements

The delimiter <delim> character must be the same character used on the left, centre, and right in a <pair par> production, and likewise on the left and right of a <equal par item> or an <escapechar> production, but they may be different between different occurrences of these productions. The delimiter character must not be contained in the character strings they enclose. For example, the following is legal:

```

nestable '()' , /[/] / ;

```

In parsing the parenthesis character declarations, an MSC analyser will read the first delimiter character and then read all characters up to but not including the next occurrence of this delimiter and take that as the parenthesis string. In the case of a paired parenthesis the analyser will then take all characters following the middle delimiter up to but not including the third occurrence of the delimiter as the matching closing parenthesis string. No escape mechanism is provided for the character strings between the delimiters since a delimiter character not contained in the parenthesis strings must be selected by the user to prevent any conflict.

## Semantics

The parenthesis and escape declarations are used to describe the lexical analysis of data <string> in 2.1. The optional <escape decl>s that may accompany the declaration of parenthesis strings denote native escape characters. These are typically used only within string or character delimiters of languages, which fall under the <equal par> type of parenthesis. The meaning attributed to a native escape declaration is as follows.

If the native escape character is identical to (one of) the <par> parenthesis string(s) to which it is attached, then it is deemed to be a *double character* type of escape. If the escape character is different from the parenthesis to which it is attached, then it is of the *escape following character* type of escape. The first type is typified by SDL strings, in which *Wayne's World* is written 'Wayne''s world' and the second is typified by C in which '\ is the escape character, giving 'Wayne\'s world'.

The SDL form can be declared by:

```

equalpar \'\ escape \'\ ;

```

And the C form can be declared by:

```

equalpar '\"' escape '\ ' ;

```

The rule for interpreting a data string after encountering the opening parenthesis <par> of the *double character* type is: if <par><par> is encountered within the parenthesis, then treat this as part of the string and not as indicating the closing parentheses, otherwise if <par> is encountered it is treated as the matching closing parenthesis.

In the case of the *escape following character* form, once an opening parenthesis having a declared escape string <par> is encountered in a data string, if <par> is encountered then treat the next character as an ordinary character and continue.

If a parenthesis declaration has a native escape declared with it then the global escape mechanism defined by <escape decl> does not apply within the scope of such parenthesis. Thus, if we have declared '?' as the global escape character, and have either the SDL or C style string delimiter/escape declarations given above, then the question mark in the string 'Why?' has no special significance.

### 5.3 Semantic interface to external data languages

In order to define the semantics of an MSC with data a number of functions relating to the data strings have to be supplied, ranging from functions to ensure that the data strings are syntactically well formed, to semantic functions that can evaluate data expressions. That is, the semantics of an MSC is parameterised by these data functions, and an MSC analysis tool would have only to be provided with instances of these functions to compute the MSC semantics. The functions can be separated into those required for static requirements and the remainder for dynamic semantics.

#### Static requirements interface functions

To check that each of the four kinds of data <string> conform to their grammar, four *well-formedness* predicates are required that are to be satisfied only when their argument strings correctly parse. Thus we require four predicates Wf1, Wf2, Wf3, and Wf4 having the following signatures:

Wf1: <variable string> → Bool

Wf2: <data definition string> → Bool

Wf3: <type ref string> → Bool

Wf4 <expression string> → Bool

To ensure that the data strings not only parse correctly, but also conform to the static requirements of the language, we require three *type checking* functions to be defined. The first of these functions, Tc1 for <data definition string>s requires no additional arguments.

Tc1: <data definition string> → Bool

The second function Tc2 is to check <type ref string>s, but as these strings may use information defined in the <data definition string>, such as the definition of constants used to supply array bounds, or references to type definitions, the function has to take these as parameters.

Tc2: <data definition string> → <type ref string> → Bool

The type-checking function for <expression string>s, Tc3, has to be parameterised by the <data definition string>, together with information defined by the variable and wildcard declarations. The latter information can be coded as a set of pairs, the first element being a <variable string> and the second its corresponding <type ref string>, regardless of whether the <variable string> refers to a variable declaration or a wildcard declaration. Thus the signature for Tc3 is:

Tc3: <data definition string> × (<variable string> × <type ref string>)-set →  
<expression string> → Bool

In several places the static requirements of MSC requires that variable names be unique, and so a function EqVar to compare variable names is also required in the interface:

EqVar: (<variable string> × <variable string>) → Bool

The last function is used in checking type conformance between different data strings, for example to check that the type of a variable is the same as that of an expression to which it is being assigned. It is sufficient to have a function that can check an expression has a required type; again contextual information from the data definition strings, and the variable/wildcard declarations is required.

Tc4:  $\langle \text{data definition string} \rangle \times (\langle \text{variable string} \rangle \times \langle \text{type ref string} \rangle)\text{-set} \rightarrow$   
 $\langle \text{type ref string} \rangle \times \langle \text{expression string} \rangle \rightarrow \text{Bool}$

### Dynamic semantic interface functions

Four functions are required to define the dynamic semantics of MSC with data. The first three of these are functions over the syntax of the data language, and only the fourth, Eval, is a semantic evaluation function. The syntactic functions are required to deal with data expressions that contain wildcards as these have different semantics from plain expressions found in many data languages. The dynamic semantics makes transformations from expressions that contain wildcards to ones that do not via the first three functions.

The first function, Vars, is used to extract the  $\langle \text{variable string} \rangle$ s from an  $\langle \text{expression string} \rangle$ , and is required both to perform dynamic semantic checks, such as variables being written before being read, and as a means to identify wildcards. Because the semantics treats multiple wildcards appearing in one expression as independent quantities, Vars has to identify how many times a  $\langle \text{variable string} \rangle$  appears in an expression. Just as the function Tc4 is parameterised by  $\langle \text{data definition string} \rangle$ s and variable/wildcard declaration information, so is Vars. The result of applying Vars to a data expression is a set of pairs, the first element of each pair being a  $\langle \text{variable string} \rangle$  and the second the number of occurrences of the variable in the expression. Only variables appearing in the expression are to appear in the functions result. The signature of Vars is given by:

Vars:  $\langle \text{data definition string} \rangle \times (\langle \text{variable string} \rangle \times \langle \text{type ref string} \rangle)\text{-set} \rightarrow$   
 $\langle \text{expression string} \rangle \rightarrow (\langle \text{variable string} \rangle \times \text{Nat})\text{-set}$

For example, the application of Vars to the expression "f(x, c, x + y)", where x and y are declared as variables/wildcards and c is a constant would produce the set:

{ (x, 2), (y, 1) },

since x appears twice in the expression and y appears once.

In order that an expression containing multiple identical wildcards be correctly evaluated separate occurrences must be replaced by unique variable names. This requires the next two functions, Replace that can substitute an occurrence of a variable or wildcard string in an expression with another variable string, and NewVar that can generate a new variable string. The substitute function takes as arguments a  $\langle \text{data definition string} \rangle$ , the variable string that is to be substituted, a number defining which occurrence is to be substituted, the replacement variable string, and finally the expression to be transformed. The  $\langle \text{data definition string} \rangle$  has to be supplied as an argument because – depending upon the data language – identifiers can be overloaded as constants/variables, etc., and distinguishing between a variable identifier string and other identifier strings requires contextual information. The signature is:

Replace:  $\langle \text{data definition string} \rangle \rightarrow$   
 $\langle \text{variable string} \rangle \times \text{Nat} \times \langle \text{variable string} \rangle \rightarrow \langle \text{expression string} \rangle \rightarrow$   
 $\langle \text{expression string} \rangle$

The result of Replace(data\_defs)(x, 2, z)(f(x, c, x + y)) could be the expression f(x, c, z + y), or f(z, c, x + y), depending on how indexing is done. In supplying a Replace function for a data language, we do not specify how occurrences of variables are to be indexed, as the function is used in the dynamic semantics to replace all wildcards, their order of replacement being unimportant.

The substitution of wildcards requires that new variable strings be created, and this is achieved by supplying a NewVar function to the interface. Given a set of <variable string>s the function is required to generate a new <variable string> that is different from any in the supplied set. As for Replace, the names of other defined identifiers will have to be taken into account from the data definition strings. The NewVar signature is:

NewVar: <data definition string> → <variable string>-set → <variable string>

The Eval function is used to evaluate a data expression, and provides the semantic interface to MSC. The Eval function returns the value of a data expression in some data domain. For example evaluating an integer expression will result in an integer value. In general expressions will return more complex values representing structured types, such as arrays, etc. Implicitly, the data domains are part of the semantic interface, and will be represented by U, a universe of data values that includes structured values. Eval takes a <data definition string> as its first argument, to provide contextual information about the second argument, the expression string to be evaluated. The final argument represents the state information and consists of bindings between variable strings and their domain values. The result of the function is a domain value.

Eval: <data definition string> → <expression string> → (<variable string> × U)-set → U

The Eval function is partial and is only defined if all variables appearing in the expression have defined values in the state argument.

## 5.4 Declaring data

The declaration of data mostly takes place in an MSC document, the only exception being static variables, which are declared in an MSC head. MSC document declarations include:

- messages and timers that have data parameters,
- dynamic variables,
- wildcard symbols,
- data definitions.

In addition, the data string parentheses and escape-character are also declared in an MSC document. Messages that have parameters are declared so that the type and number of parameters are defined. Messages that do not have parameters need not be declared. Dynamic variables are declared inside an MSC document's instance list, as dynamic variables are owned by instances. A declaration gives the names of the variables and defines their type. Variables that are to represent wildcards are declared together with their type in the MSC document, as are the data definitions. The data definitions consist of text in the data language that, for example, defines structured types, constants, and functions signatures. It must provide all information required to type check and evaluate data expressions used in MSCs within the scope of the enclosing MSC document.

### Concrete textual grammar

```

<message decl list> ::=
    <message decl> [ <end> <message decl list> ]

<message decl> ::=
    <message name list> [ : <left open> <type ref list> <right open> ]

<message name list> ::=
    <message name> [ , <message name list> ]

<timer decl list> ::=
    <timer decl> [ <end> <timer decl list> ]

<timer decl> ::=
    <timer name list> [<duration>] [ : <left open> <type ref list> <right open> ]

<timer name list> ::=
    <timer name> [ , <timer name list> ]

```

```

<type ref list> ::=
    <type ref string> [ , <type ref list> ]

<dynamic decl list> ::=
    variables <variable decl list> <end>

<variable decl list> ::=
    <variable decl item> [ <end> <variable decl list> ]

<variable decl item> ::=
    <variable list> : <type ref string>

<variable list> ::=
    <variable string> [ , <variable list> ]

<data definition> ::=
    [ language <data language name> <end> ]
    [ <wildcard decl> ]
    [ data <data definition string> <end> ]

<wildcard decl> ::=
    wildcards <variable decl list> <end>

```

## Concrete graphical grammar

The data concepts are contained within textual parts and no graphical grammar is needed.

### Static requirements

All messages and timers that possess parameters must be declared in the <document head>. The number and type of parameters of a message or timer that carries data must also be given at its declaration. A type is given in the parameter data language as defined by <type ref string>. All type references used in variable, wildcard, timer and message declaration must be legally defined in the context of the <data definition string>s according to the rules of the data language, that is the type references must be syntactically correct according to the Wf3 function and type correct according to the Tc2 function. All variable and wildcard declarations must be legal according to the Wf1 function, and all their names must be unique as judged by the EqVar function. The <data definition string> must contain all the information to be used in performing the static and dynamic semantic checks required by this standard. Although this string may not form legal fragments of the intended data language, they must represent a formalised abstraction of the language and have legal syntax and (static) semantics, that is they must be syntactically correct according to Wf2 function and type correct according the Tc1 function. For example, it would be reasonable to give only procedure headers that define a function's signature and not its complete header/body, but such abstraction would have to be formalised and capable of being checked via the interface functions.

### Semantics

A wildcard is used in data expressions as a "don't care" value. In defining the meaning of an MSC with data, a wildcard will generate a set of concrete traces corresponding to each uninterpreted trace, where each concrete trace is derived from the uninterpreted trace by substituting a different concrete value for the wildcard. If an expression contains multiple occurrences of a wildcard then each represents a different reference, so that different concrete values will, in general, be substituted for each occurrence.

## 5.5 Static data

Optionally an MSC can define a formal parameter list. A corresponding MSC reference must define a list of actual parameters. An MSC parameter list declares a list of formal parameter variables, whose scope is the MSC body. When a parameter appears in the body, it must only be used in expressions that reference its value, hence they cannot be modified dynamically.

## Concrete textual grammar

```
<data parameter decl> ::=
    [ variables ] <variable decl list>

<actual data parameters> ::=
    [ variables ] <actual data parameter list>

<actual data parameter list> ::=
    <expression string> [ , <actual data parameter list> ]
```

## Static requirements

The <data parameter decl> declares the formal parameters of an MSC. If the variables declarations follow after timer, or message, etc. declaration block then the keyword **variables** is used to indicate that the previous block is complete. If the variable declarations occur first amongst the parameter declarations then **variables** keyword is optional. Each formal parameter in the declaration must be unique within the declaration and also different from all dynamic variables and wildcards declared in the owning MSC document. Each variable must be well formed according to Wf1, and its type reference well formed and type correct in the context of the enclosing MSC document data string according to Wf3 and Tc2, respectively. The number and type of parameters must be adhered to in an MSC reference via its <actual data parameter list>; type conformance is performed by the Tc4 function, given the context of the MSC data definition string and the formal parameter declaration. Each actual parameter is an expression that may contain variables and wildcards. All variables appearing in the <actual data parameter list> must be static variables declared in the enclosing MSC; dynamic variables are forbidden. The use of the keyword **variables** in the <actual data parameter list> follow the rules as for <data parameter decl>.

## Semantics

The meaning of an MSC reference with actual parameters is "call by value", in which the formal parameters are substituted by the actual parameters wherever they appear in its body. However, if there are multiple wildcards given in the <actual data parameter list> then each of these represents a separate and unique reference that will be semantically distinguished in the body of the referenced MSC by substituting them with distinct new variable names, each allowed to take any value in their domains. Evaluating the actual parameters is done by the Eval function once the wildcards have been identified and replaced by new unique variables. Replacement is done by first extracting all the expression variables/wildcards using the Vars function, and then using the MSC document wildcard declarations to identify which are the expression wildcards. Finally the wildcards are replaced using the Subst function with new variables generated by the NewVar function. These new variables can take any value in their data domains.

### 5.6 Dynamic data

Dynamic data refers to MSC variables that can be assigned and reassigned values through action boxes, message and timer parameters, and instance creation. The value that a dynamic variable may possess at any point in a trace will, in general, depend upon the previous events in the trace. The home of a dynamic variable is an instance, and its declaration is given in the MSC document. Only events on the owning instance of a dynamic variable are allowed to modify its value, although other instances may reference it under certain restrictions.

The mechanism for assigning or modifying the value of a dynamic variable is that of a binding, which consists of a pattern and an expression. Bindings occur as a result of message passing, timer set, timer timeout or instance creation via their parameter lists, or in action boxes. The dynamic semantics defines a set of current bindings for each event in an execution trace, and is called the event's state. The use of wildcards in expressions results in underspecification in MSC. Each wildcard is allowed to range over all values of its domain type.

In a defining MSC there must be no trace through an MSC in which a variable is referenced without being defined. That is, each variable appearing in an expression must be bound in the state used to compute the value of the expression. In a utility MSC, references to undefined variables are permitted.

A state associated with a current event is computed from previous states together with the data content of that event. The previous states used to compute the new state depend upon the type of event, all are derived from at least the last non-creating event executed on the same instance as the current event. In addition, for message receiving events and for the first event on a created instance, the state of the corresponding send or creating events is also used in the computation. Effectively, this means that a state is maintained by each instance, and a new state is derived from the instance's previous state together with state information passed to the instance through messaging, or from the parent instance in the case of instance creation. Because information is allowed to flow between instances via message passing and instance creation, the state associated with each event may contain bindings to variables not owned by the instance upon which the event occurs. The rules governing the access to the value of variables owned by foreign instances are defined as follows.

If a sending event or a create event has a binding to a variable *x* in its associated state, and *x* appears in one of its parameter expressions, and *x* is not owned by the receiving or created instance, then the binding is to be added to the resulting state of the receiving event, or replace a binding if one exists in the old state. That is, if the value of *x* is recorded in the state of the sending/creating instance, then this binding is implicitly inherited by the state of the receiving/created instance, so long as *x* is not owned by the receiving/created instance.

If *x* is owned by the receiving instance then the binding defined in the sending event can only be inherited if *x* is not bound in the state of the receiving instance and *x* is not bound in the parameter list. That is, it can only be inherited if *x* is undefined by the receiving instance and also by the message parameters.

If *x* is owned by the receiving/created instance and *x* is bound in the message parameters, then the new state takes this binding. That is, *x* becomes bound to the value defined by the expression part of the parameter binding. This is the normal mode of parameter binding.

If *x* is owned by the receiving instance and is bound in the old state of this instance, but is not bound in the parameter list, then this old binding is retained in the new state. That is, the binding of the sending event is not inherited. This reflects the expectation that a variable, once defined on its instance can only have its value modified explicitly. All references to it must take the last explicitly defined value.

In summary, if *x* is not bound either in the old state or in the parameter list, then the binding from the sending event can be inherited. Intuitively, the binding of a variable can be inherited from another instance only if the variable is sent to the instance by appearing in a parameter's expression. However, a binding cannot be inherited if the variable is owned and in scope by the receiving instance, as the local binding takes precedence. Thus, the value of a variable can be transmitted by a chain of messages to other instances, so long as each message explicitly references the variable in its parameter list.

## **5.7 Bindings**

Bindings are more general than simple assignments found in programming languages because of the use of wildcard symbols, which permit underspecification. A binding consists of an expression part and a pattern part that are connected by a bind symbol. The bind symbol has a left and a right form both of which are equivalent, but which permits a more natural reading of a binding associated with a message or timer. Wildcard symbols can be used in a binding wherever a variable could be used, but have a different interpretation from variables that amounts to them taking all permissible values rather than one assigned value.



The pattern part of a binding consists of either a wildcard or a dynamic variable; the expression part is a data expression, which may contain wildcards, dynamic variables, and static variables. In MSC the wildcard symbols are user defined, but we shall use "\_" in the following examples to be the wildcard. The example below shows equivalent left and right binding that are free from wildcards:

$$x := y + 3, \quad y + 3 =: x$$

The foregoing bindings can be read as an assignment, or binding, of the value of the expression 'y + 3' to the variable x. If a wildcard is used in place of the x, as in the following example, then no assignment is made and so the grammar for message parameters allows an equivalent shorthand of just using the expression, also shown. This shorthand is useful where the sending value of a parameter is specified but the receiving instance does not bind this value to any of its dynamic variables.

$$\_ := y + 3, \quad y + 3$$

Wildcards are used in expressions to represent "don't care" values. If the dynamic semantics prescribes a single uninterpreted trace through an MSC that contains an expression wildcard, then any value may be substituted for the wildcard to give a legal concrete trace. Furthermore, if there are multiple occurrences of wildcards (even those using the same wildcard symbol), then each of these is independent and so can be substituted with different values. Suppose in the expression  $\_ + 3$  that the wildcard '\_' must be of type Natural (i.e., a non-negative integer), then semantically it can be substituted with any of the values 0, 1, 2, ... . Thus the values that the expression can take are 3, 4, 5, ... . In the expression  $\_ + \_$ , the two occurrences of the wildcard symbols are independent, and so both the first and second occurrence are free to range independently over the natural numbers. For example, the first wildcard can take the value 1 and the second the value 3 to give the result 4. If the same values are required, then a variable can be bound to a wildcard in an action box and then this variable be used in the expression in place of the wildcards. Given the initial binding of x to a wildcard, then  $x + x$  can only be evaluated to even natural numbers.

### Concrete textual grammar

```

<binding> ::=
                <left binding> | <right binding>

<left binding> ::=
                <pattern> <left bind symbol> <expression>

<left bind symbol> ::=
                :=

<right binding> ::=
                <expression> <right bind symbol> <pattern>

<right bind symbol> ::=
                =:

<expression> ::=
                <expression string>

<pattern> ::=
                <variable string> | <wildcard>

<wildcard> ::=
                <wildcard string>

```

### Static requirements

The expression part of a binding must comply with the static requirements rules of the data language given the context of the <data definition string> defined in the <document head> according to the functions Wf3 and Tc3. All variables appearing in an expression must be declared as static variables, dynamic variables, or wildcards; all other identifiers must either be defined via the <data definition string> or be predefined in the data language. All pattern variables must be

declared as dynamic variables owned by the correct instance – the correct instance being determined by the event in which the binding appears.

The type of the pattern must match the type of the expression in a binding. Type equivalence can be checked by applying the function Tc4 to both pattern and expression.

## Semantics

A binding, or list of bindings, is evaluated in the context of a state and gives rise to a new state. A state consists of a set of bindings between variables and their values. To evaluate a binding, its expression is first evaluated using the current state, where the values of variables in the expression are taken from its binding in the state. If there is no binding of an expression variable in the current state, then there is an illegal reference to an undefined variable in the expression. The resulting value computed from the expression is bound to the pattern variable and the resulting new binding is added to the current state to form the new state. If the pattern variable is already bound in the current state then it is superseded in the new state by the new binding.

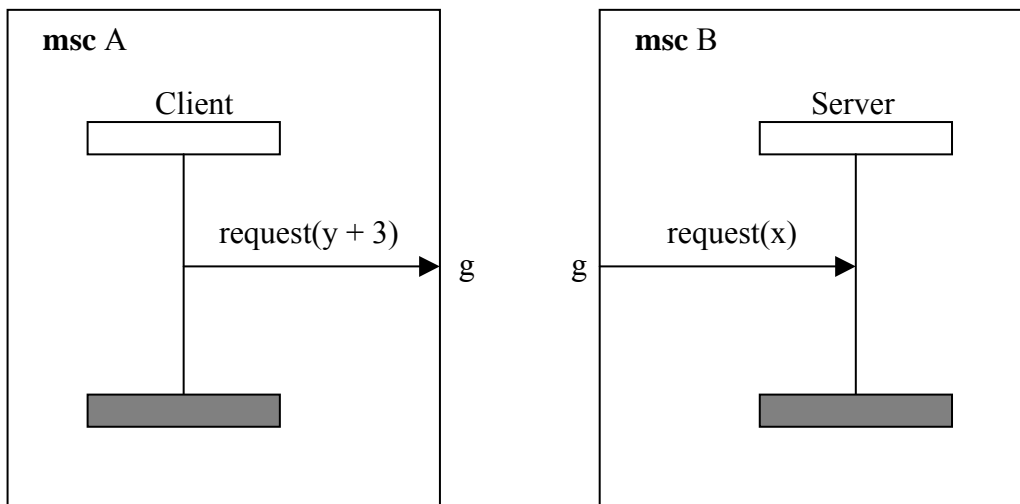
If the pattern part of a binding is a wildcard, then the binding does not modify the state. If the expression part of a binding contains wildcards, then each is replaced by unique new variables that are allowed to take any value in their domain. Evaluating an expression is done by the Eval function once the wildcards have been identified and replaced by new unique variables. Replacement is done by first extracting all the expression variables/wildcards using the Vars function, and then using the MSC document wildcard declarations to identify which are the expression wildcards. Finally the wildcards are replaced using the Replace function with new variables generated by the NewVar function. These variables can take any value in the trace semantics of an MSC.

### 5.8 Data in message and timer parameters

Dynamic data partakes in messages and timers via their parameter lists. A parameter list consists of a list of parameters, which may be bindings, expressions, or patterns. The static requirements determine which of these three options can or must be used. For completed messages in an MSC, a binding or an expression can be used, but not a pattern. In this context an expression is a shorthand for binding to a wildcard. For incomplete messages more complex rules apply.

Incomplete messages occur when a message originates or terminates at a gate, or is a lost or found message. When a message terminates at or originates from a gate, bindings cannot be explicitly defined in its parameter list since they only make sense when both the source and destination instances are known. Bindings will be dynamically inferred in the semantics by pairing up patterns and expressions given by the two messages matched across the gate. For a message that is sent from an instance and terminates at a gate only the expression can be given. For a message that is picked up from a gate and sent to an instance, only the pattern can be given. Similar restrictions are placed upon lost and found messages. In the example in Figure 9, the dynamic semantics will match the message *request* across the gate *g* and infer the binding  $y + 3 =: x$ . If a message both originates and terminates at a gate then no parameter information is permitted.

The set of bindings defined or formed by a message's parameters results in a change of state at the receiving event. The bindings are evaluated concurrently, and are used to update the old state. As the bindings are concurrent, the set of variables appearing as pattern variables in the bindings must be distinct to prevent two bindings attempting to bind two values to the one variable. Furthermore, the pattern variables must all be owned by the receiving instance, as this instance is allowed only to modify its own variables and no others.



**Figure 9 – Messages, expressions, patterns and gates**

### Concrete textual grammar

```

<parameter list> ::=
    <parameter defn> [ , <parameter list> ]

<parameter defn> ::=
    <binding> | <expression> | <pattern>

```

### Static requirements

The number and type of message or timer parameters must comply with the types defined in the message or timer declaration. The type of a binding is determined from either its pattern or expression as they also must match. Type conformance is checked using the Tc4 function. In a parameter list, the set of pattern variables must be unique and must all be owned by the receiving instance of the message. As the bindings of a message parameter list are evaluated concurrently, this prevents ambiguity.

Only a binding or an expression can be given as message parameters in completed messages; patterns cannot. Only an expression can be given for incomplete sending events; bindings or patterns cannot. Only patterns can be given for incomplete receiving events; bindings or expressions cannot.

### Semantics

Message parameters are responsible for updating state in message receive events. Output events do not modify state, and their state is simply inherited from the last non-create event on their instance.

For completed messages all bindings in the message parameter list of the receiving event are evaluated using the old state, and the resulting bindings are used to update the old state to form the new state. That is, a new binding is added to the old state, or if an already bound variable in the old state is rebound in the parameter list, then the newer binding replaces the old one in the new state. Variables that appear in the expression parts of a parameter list also contribute to the updating of the old state. The bindings of these referenced variables are taken from the state of the sending event and also used to update the old state, except for variables that are owned by the receiving instance. In the latter case the binding is only added if the variable is not bound in the old state and is not bound by the parameter list.

For incomplete message receiving events, firstly bindings are dynamically created by pairing the matching parameter expressions from the corresponding send event with the patterns of the receiving event. The resulting bindings are then evaluated as for completed messages, and the state modified in the same way.

## 5.9 Data in instance creation parameters

Dynamic data in instance creation events is treated similarly to message parameters. However as there is no created event corresponding to the creating event, the state is modified by the create event according to any bindings in its parameter list. But this modified state is only used in evaluating the next event on the created instance, and not the next event on the creating instance. The state that existed prior to the create event is used to evaluate the subsequent event on the creating instance. All variables used as pattern variables in a create parameter list must be owned by the created instance.

## 5.10 Data in action boxes

Data can appear in action boxes as a comma separated list of <data statement>s. A statement is either a <define statement>, an <undefine statement>, or a <binding>. A <define statement> is used to indicate that a variable has been assigned some unspecified value; it is the equivalent of a binding of a variable to a wildcard. That is, "**def** x" is the equivalent of "x := \_", where "\_" is a wildcard. An <undefine statement> is used to indicate that a variable is no longer bound, i.e., that the variable cannot be legally referenced, or has moved out of scope. Inside a single action box the statements are evaluated concurrently, not sequentially, as a consequence of which the static requirements rules forbid ambiguity between different statements attempting to bind different values to the same variable. Sequencing can be achieved by sequencing action boxes.

### Concrete textual grammar

```
<data statement list> ::=
    <data statement> [ , <data statement list> ]

<data statement> ::=
    <define statement> | <undefine statement> | <binding>

<define statement> ::=
    def <variable string>

<undefine statement> ::=
    undef <variable string>
```

### Static requirements

All variables occurring in a define statement, undefine statement, or the pattern part of a binding must be dynamic variables owned by the instance on which their enclosing action box appears, and furthermore they must be distinct. The latter is required because the statements of an action box are concurrently evaluated.

### Semantics

Each statement in an action box is evaluated using the state of the previous non-creating event on the same instance. The resultant state is derived from the old by updating the bindings made or destroyed by each of the statements. Because of the static requirements, there can be no ambiguity in forming the resulting state. For <binding>s, the pattern variables become bound in the new state to the values of their expressions. An <undefine statement> removes the variable's binding from the old state, if there is one, in forming the new state. A <define statement> adds or replaces a binding from the old state with one in which the statement's variable is bound to a wildcard.

## 5.11 Required data types

There are three places in this standard where the MSC language requires the existence of data types. These are:

- Boolean valued expressions used in guarding conditions (Section 4.7),
- natural number expressions used to define loop boundaries (Section 7.2),
- time expressions used in specifying timing constraints (Section 6).

In keeping with approach to data used within this standard these types have to be defined as part of the user's chosen data language and not part of the MSC language. However, they are required to have the specific interpretations as the MSC semantics are defined in terms of these interpretations. The default data language binding for MSC is SDL, as defined by Z.121, and in which the SDL types corresponding to the required types are given.

The type of expressions used in guarding conditions must have the standard Boolean domain consisting of a *true* and *false* element only. For loop boundaries the domain of the type must be the natural numbers, equipped with the usual subtraction operation (required to compute the difference between the lower and upper bound of a loop). This condition can be relaxed to permit the domain to be a subset of the natural numbers, since most (programming) languages only have finite representations. This does not affect the semantics of loops, just that boundary values will be limited to the permitted subset. In particular infinite loops still can be expressed by using the keyword **inf**.

The requirements for the time type are more abstract and are defined as:

- the domain must be a total order with a least element, or origin, of time zero,
- the domain must be closed under an addition operation, used to compute time offsets.

These are the minimal requirements that enable the semantics of timed MSCs to be defined. In practice a user would have a richer type that supported extra time operations declared in the data definition part of the MSC document. As with the natural numbers, the actual domain can be permitted to be a subset of the required ideal domain to allow for limitations in actual data languages. The total ordering requirement reflects the linear time model used in MSC, as opposed to, say, branching time. The closure of addition means that time can never be exhausted.

## 6 Time concepts

Time concepts are introduced into MSC to support the notion of quantified time for the description of real-time systems with a precise meaning of the sequence of events in time. MSC events are instantaneous. Time constraints can be specified in order to constrain the time at which events may occur.

Each MSC contains instances with associated events. Classical MSC disregarding time can be interpreted as a set of traces of events. In the untimed interpretation, all time related features are considered to be comments/annotations only. In the timed interpretation, the progress of time is represented explicitly in a quantified manner, i.e., the traces of events are enhanced with a special event which represents the passage of time. The untimed interpretation of an MSC can be derived from the timed interpretation by removing the extra time events from its traces; in doing so groups of different timed traces will reduce to the same untimed trace.

Timing in MSC enhances the traces of an MSC with quantitative time values, which represent the time distance between pairs of events. The time progress (i.e., clocking) is equal for all instances in an MSC. Also, all the clock values are equal, i.e., a global clock is assumed. All events are instantaneous, i.e., atomic and do not consume time.

## 6.1 Timed semantics

The timed semantics of an MSC can be represented by traces with special time events such as

$$\{e1, e2, t3, e4, t5, e6, e7, e8, \dots\}$$

The triple (e4, t5, e6) means for example that after the occurrence of event e4 time t4 passes until event e6 occurs. Events with no time event in between (such as e1 and e2) occur simultaneously, i.e., without any delay. It is assumed that time is progressing and not stagnating. Progressing means that after each event in a trace there is eventually a time event. Non-stagnation means that there is an upper bound on the number of 'normal events' in between a timed event and the succeeding timed event.

The above trace is equal to the trace below (by making the zero delay explicit):

$$\{e1, 0, e2, t3, e4, t5, e6, 0, e7, 0, e8, \dots\}$$

The untimed semantics of an MSC containing traces

$$\{e1, e2, e3, \dots\}$$

correspond to a set of traces with arbitrary delay in between the events, i.e.

$$\{e1, \text{any time}, e2, \text{any time}, e3, \text{any time}, \dots\}$$

On the other hand, the untimed reduct of a timed trace

$$\{e1, t1, e2, t2, e3, t3, e4, t4, e5, t5, \dots\}$$

is

$$\{e1, e2, e3, \dots\}$$

It is intentional that in the timed semantics a trace of an MSC begins with a 'normal' event, i.e., for every event except for the first one there is a preceding event. Timing can be defined with respect to the preceding event.

In general, there is not a unique start event neither for an instance of an MSC nor for an MSC as a whole. An MSC defines a set of traces with potentially different start events.

## 6.2 Relative timing

Relative timing uses pairs of events – preceding and subsequent events, where the preceding event enables (directly or indirectly, i.e., via some intermediate events) the subsequent event. For the use of relative timing please refer to Section 6.10 on Time Intervals.

Relative timing can be specified by the use of arbitrary expressions of type Time, i.e., referencing parameters, wildcards and dynamic variables. The concrete value of a relative time expression is evaluated once the new state of the event relating to this relative timing has been evaluated. Refer to Section 5.6. for the notion of new state.

## 6.3 Absolute timing

Absolute timing is used to define occurrence of events at points in time that relate to the value of the global clock.

Absolute timing can be specified by the use of arbitrary expressions of type Time, i.e., referencing parameters, wildcards and dynamic variables. The concrete values of a time constraint are evaluated at the start of a time interval once the new state of the event relating to the start of the time interval has been evaluated. Refer to Section 5.6. for the notion of new state.

## 6.4 Time domain

The time domain can be dense or discrete. It must be a total order with a least element, or origin, of time zero. It must be closed under an addition operation, used to compute time offsets.

The units of, and representation of elements from, the time domain can be specified via the MSC data interface Section 5.4. However the default data binding for MSC is taken as SDL as defined in Z.121. The default binding is to the Time type in SDL.

### Static requirements

The time domain must be a total order with a least element, or origin, of time zero. It must be closed under an addition operation, used to compute time offsets.

## 6.5 Static and dynamic time variables

An MSC can use static or dynamic time variables. These variables are like any other variable except that they are of the type Time of the MSC.

### Static requirements

Static and dynamic time variables adhere to the requirements of other static and dynamic variables as described in the section on Data Concepts (5).

## 6.6 Time offset

An MSC can be assigned a time offset, which is used as an offset to all absolute time values within that MSC. The time offset is defined by an expression of the time domain of the MSC. In MSC without an explicit time offset, a time offset of 0 is assumed by default.

### Concrete textual grammar

```
<time offset> ::=  
                offset <time expression>
```

### Concrete graphical grammar

Not needed.

### Static requirements

In the offset declaration, only an expression of type Time must be given. The expression must refer to declared MSC parameters only.

## 6.7 Time points, measurements, and intervals

Time constraints can be defined as time points, i.e., concrete time values, or as time intervals, i.e., ranges of time values within given bounds. Time observations are described by measurements.

## 6.8 Time points

### Concrete textual grammar

```
<time point> ::=  
                [ <abs time mark> ] <time expression>
```

Time points are defined by expressions of type Time. The optional absolute time mark indicates an absolute timing.

### Concrete graphical grammar

Not needed.

## Static requirements

The evaluation of a time point yields a concrete quantified time. An event without time constraints can occur at any time.

### 6.9 Measurements

Measurements are used to observe the delay between the enabling and occurrence of an event (for relative timing) and to measure the absolute time of the occurrence of an event (for absolute timing). In order to distinguish a relative from an absolute measurement, different time marks (i.e., '@' for absolute and '&' for relative) are used.

Measurements can be tied to time intervals. For each measurement, a time variable has to be declared for the respective instance.

#### Concrete textual grammar

```
<measurement> ::=
    <rel measurement>
    | <abs measurement>
<rel measurement> ::=
    <rel time mark> <time pattern>
<abs measurement> ::=
    <abs time mark> <time pattern>
```

#### Concrete graphical grammar

Not needed.

## Static requirements

Measurements must use only patterns of type Time, which refer to dynamic variables that are declared in the enclosing MSC document for the respective instance.

## Semantics

The evaluation of the delay (in the relative case) and of the value of the global clock (in the absolute case) and the binding to the time variable is part of the evaluation of the new state of the event to which the measurement relates to.

### 6.10 Time interval

Time intervals are used to define constraints on the timing for the occurrence of events: the delay between a pair of events can be constrained by defining a minimal or maximal bound for the delay between the two events.

A time interval does not imply that the events must occur. The fulfilment of a time constraint is validated only if the event relating to the end of that time intervals occurs in the trace. An MSC trace has to fulfil all its time constraints, i.e., if a trace violates a time constraint the trace is illegal.

Time intervals can be used for relative timing as well as for absolute timing. Time intervals can be specified by the use of arbitrary expressions of type Time, i.e., referencing to parameters, wildcards and dynamic variables. The concrete values of a time constraint imposed by a time interval are evaluated at the start of a time interval once the new state of the event relating to the start of the time interval has been evaluated. Refer to Section 5.6 for the notion of new state.

The order of a pair of events related to a time constraint is determined by the dynamic semantics and not by the time constraint. However unidirectional time constraints only apply to events when they occur in the order indicated by the constraint direction.



By means of interval boundaries, each event can impose constraints on time intervals, in which it is involved. However, these are taken into account only if the event refers to the start of a time interval, which is determined dynamically. In the case that an event refers to the end of a time interval, its time constraints are of no concern. The textual and graphical representation of an MSC indicate the potential pairs of events and whether the constraint is directed or not. Such a pair of event is indicated either by connecting the interval boundaries of two events, by connecting split interval boundaries via an interval label, or via message gates. A directed constraint applies only to events when they dynamically occur in the direction indicated; if dynamically the events occur in the opposite sense, the constraint does not apply.

### Concrete textual grammar

```

<time interval> ::=
    [<interval label>] <singular time>
    | [<interval label>] <bounded time>
    [ <measurement> ]

<interval label> ::=
    [ int_boundary ] <interval name>

<singular time> ::=
    <left closed> <time point> <right closed>
    | <measurement>

<bounded time> ::=
    [<abs time mark>] { <left open> | <left closed> }
    [<time point>], [<time point>]
    { <right open> | <right closed> }

```

### Static requirements

Within a time interval, either only relative time expressions or only absolute time expressions must be used. Either the minimal, the maximal bound or both bounds are given. An interval must define at least one of the two bounds.

An absolute time interval must be of the form [**@1,@3**) or **@**[1,3).

Time intervals can be defined for any two events within an MSC document.

Where an interval label is used, the keyword **int\_boundary** must appear in the programming representation, and must be absent in the graphical representation.

### Concrete graphical grammar

```

<time interval area> ::=
    <interval area>
    | <abs time area>

<interval area> ::=
    <int symbol>
    is associated with <time interval>
    is followed by { <cont interval> | <interval area 2> }

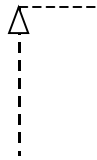
<interval area 2> ::=
    <int symbol>
    [is associated with <time interval>]

<cont interval> ::=
    <cont int symbol>
    is associated with <interval name>

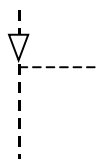
```

<int symbol> ::=  
 {<int symbol 1> | <int symbol 2> | <uni int symbol>}  
*is attached to*  
 { <message out symbol> | <message in symbol> |  
 <reply symbol> | <action symbol> |  
 <timer start symbol> | <timer stop symbol> | <timeout symbol> |  
 <restart symbol> | <createline symbol> |  
 <inline expression symbol> | <separator symbol>  
 <msc reference symbol> | <exc inline expression symbol>  
 <instance head symbol> }

<int symbol 1> ::=



<int symbol 2> ::=



<uni int symbol> ::=



The time interval symbols <int symbol 1>, <int symbol 2>, and <uni int symbol> may be mirrored about a horizontal or vertical axis. The mirror of <int symbol 1> or <uni int symbol> must not be attached to <int symbol 1> or <uni int symbol>. The <uni int symbol> is used to indicate that it is the origin of a directed time constraint. That is, the constraint only applies when the event or region to which it is connected applies only when it is dynamically the first event. The other end of the constraint with which it is connected, possibly via a label, has to be a directed symbol, i.e., <int symbol 1> or <int symbol 2>.

<cont int symbol> ::=



The continuation symbol for time intervals <cont int symbol> may be mirrored about a horizontal axis. The mirror of <cont int symbol> must not be attached to <int symbol 1>, <int symbol 2>, or <uni int symbol>. Similarly, <cont int symbol> must not be attached to the mirror of <int symbol 1>, <int symbol 2>, or <uni int symbol>.

<abs time area> ::=

<abs time symbol>  
*is associated with* <abs time interval>  
*is attached to*  
 { <message out symbol> | <message in symbol> | <action symbol> |  
 <timer start symbol> | <timer stop symbol> | <timeout symbol> |  
 <inline expression symbol> | <separator symbol> |  
 <msc reference symbol> | <exc inline expression symbol>  
 <call in symbol> | <call out symbol> | <reply symbol> }

```

<abs time symbol> ::=
    -----
<abs time interval> ::=
    <abs time expr> | <abs bounded time> | <abs measurement>
<abs bounded time> ::=
    <abs time mark> { <left open> | <left closed> }
    [ <time point> ], [ <time point> ]
    { <right open> | <right closed> }

<abs time expr> ::=
    <abs time mark> <time expression>

```

The graphical representation of a relative timing interval uses dashed lines with one or two arrow heads (such as empty arrow heads) that indicate the events constrained.

Small variations on the actual shapes of the arrow heads such as shading or the shape of the arrow heads are allowed. However, it is recommended that the arrow heads of time intervals are distinguishable from the arrow heads of message or create events.

The graphical representation of an absolute timing interval uses one dashed line to point at the absolutely timed event.

Time intervals can be split into several parts, which are logically connected via a label. In the course of execution, split intervals are joined to constitute the start and end of an interval.

The horizontal lines of the relative and absolute time symbols can be represented by arbitrary polylines and can be stretched and squeezed. But, the vertical lines of the relative time symbols must remain vertical. The final part of an absolute time polyline must be horizontal and must be associated with the absolute time expression or the absolute measurement, respectively.

## Semantics

In every trace of an MSC there has to be a unique correspondence between the start and the end of a split time interval (i.e., joining of split time intervals is done at execution time only). For example, in a trace there must not be two ends for an interval. A unidirectional constraint is specified by having an arrowhead at only one end of the constraint. The end without the arrow head, graphically represented by <uni int symbol>, defines that this is the origin of the constraint. That is, the constraint only applies when the event, or the first event in the region it is attached to, occurs dynamically before the event or region that the arrowhead end of the constraint is attached to. When the events occur dynamically in the reverse order, they remain unconstrained.

## 7 Structural concepts

In Section 7, high level structural concepts are introduced. Coregions make it possible to describe areas where the events may come in any order. Inline expressions help structure notions of alternatives, parallel composition and loops. MSC references are used to refer other MSCs from within an MSC. HMSCs abstract from instances and give overview of more complicated behaviours.

MSC references and inline expressions may have time constraints. Timing is interpreted with respect to the dynamically determined start and end events of the MSC reference and the inline expression, respectively.

## 7.1 Coregion

The total ordering of events along each instance (see 4.2) in general may not be appropriate for entities referring to a higher level than simple-processes.

Therefore, a coregion is introduced for the specification of unordered events on an instance. Such a coregion in particular covers the practically important case of two or more incoming messages where the ordering of consumption may be interchanged. Conversely, when broadcasting messages, the sending of two or more outgoing messages may be interchanged. A generalized ordering can be defined by means of general ordering relations.

### Concrete textual grammar

```
<start coregion> ::=
    concurrent <end>

<end coregion> ::=
    endconcurrent <end>
```

### Concrete graphical grammar

```
<concurrent area> ::=
    <coregion symbol>
    is attached to <instance axis symbol>
    contains <coevent layer>

<coregion symbol> ::=
    <coregion symbol1> | <coregion symbol2>

<coevent layer> ::=
    <coevent area> | <coevent area> above <coevent layer>

<coevent area> ::=
    { <message event area> | <incomplete message area> | <action area> |
    <timer area> | <create area> } *

<coregion symbol1> ::=
    T
    |
    |
    |
    T

<coregion symbol2> ::=
    T      T
    |      |
    |      |
    |      |
    T      T
```

Drawing rule: The statement that the <coregion symbol> is attached to the <instance axis symbol> means that the <coregion symbol> must overlap the <instance axis symbol> as in the following example:

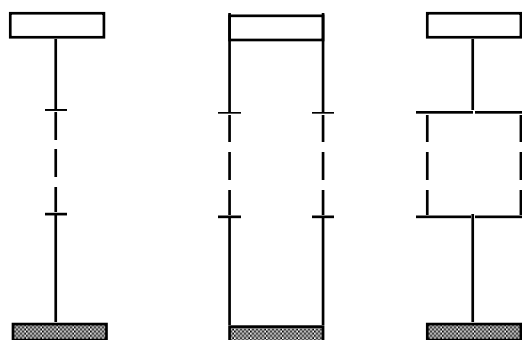


Figure 10 – Different forms of coregion

<coregion symbol1> must not be attached to <instance axis symbol2>.

## Static requirements

On a specific instance, between <start coregion> and <end coregion>, there may only be <orderable event>s.

## Semantics

For MSCs a total ordering of events is assumed within each instance. By means of a coregion an exception to this can be made: events contained in the coregion are unordered if no further synchronization constructs in form of general order relations are prescribed.

If a timer start and the corresponding time-out or stop are contained in a coregion, then an implicit general ordering relation is assumed between the start and the time-out/stop.

## 7.2 Inline expression

By means of inline operator expressions, composition of event structures may be defined inside of an MSC. The operators refer to alternative, parallel and sequential composition, iteration, exception and optional regions.

Timed inline expressions allow to constrain or to measure the execution time of alternative, parallel composition, iteration, exception and optional regions. Both, relative and absolute time intervals (with or without measurements) can be used.

Time intervals may refer to the start or/and the end of an inline expression. The start value is when the first event dynamically takes place, and the end is when the last event takes place. Therefore the time intervals apply to all instances involved in an inline expression.

## Concrete textual grammar

```
<shared inline expr> ::=
    [<extra global>]{ <shared loop expr> | <shared opt expr> |
    <shared alt expr> | <shared seq expr> | <shared par expr> | <shared exc expr> }
    [ time <time interval> <end> ]
    [ top <time dest list> <end> ]
    [ bottom <time dest list> <end> ]

<extra global> ::=
    external

<shared loop expr> ::=
    loop [ <loop boundary> ] begin [ <inline expr identification> ] <shared> <end>
    [ <inline gate interface> ] [<instance event list>]
    loop end <end>

<shared opt expr> ::=
    opt begin [ <inline expr identification> ] <shared> <end>
    [ <inline gate interface> ] [<instance event list>]
    opt end <end>

<shared exc expr> ::=
    exc begin [ <inline expr identification> ] <shared> <end>
    [ <inline gate interface> ] [<instance event list>]
    exc end <end>

<shared alt expr> ::=
    alt begin [ <inline expr identification> ] <shared> <end>
    [ <inline gate interface> ] [<instance event list>]
    { alt <end> [ <inline gate interface> ] [<instance event list>] } *
    alt end <end>
```

<shared seq expr> ::=  
     **seq begin** [ <inline expr identification> ] <shared> <end>  
     [ <inline gate interface> ] [ <instance event list> ]  
     { **seq** <end> [ <inline gate interface> ] [ <instance event list> ] } \*  
     **seq end** <end>

<shared par expr> ::=  
     **par begin** [ <inline expr identification> ] <shared> <end>  
     [ <inline gate interface> ] [ <instance event list> ]  
     { **par** <end> [ <inline gate interface> ] [ <instance event list> ] } \*  
     **par end** [ <time interval> ] <end>

<inline expr> ::=  
     [ <extra global> ] { <loop expr> | <opt expr> | <alt expr> |  
     <seq expr> | <par expr> | <exc expr> }  
     [ **time** <time interval> <end> ]  
     [ **top** <time dest list> <end> ]  
     [ **bottom** <time dest list> <end> ]

<loop expr> ::=  
     **loop** [ <loop boundary> ] **begin** [ <inline expr identification> ] <end>  
     [ <inline gate interface> ] <msc body>  
     **loop end** <end>

<opt expr> ::=  
     **opt begin** [ <inline expr identification> ] <end>  
     [ <inline gate interface> ] <msc body>  
     **opt end** <end>

<exc expr> ::=  
     **exc begin** [ <inline expr identification> ] <end>  
     [ <inline gate interface> ] <msc body>  
     **exc end** <end>

<alt expr> ::=  
     **alt begin** [ <inline expr identification> ] <end>  
     [ <inline gate interface> ] <msc body>  
     { **alt** <end> [ <inline gate interface> ] <msc body> } \*  
     **alt end** <end>

<seq expr> ::=  
     **seq begin** [ <inline expr identification> ] <end>  
     [ <inline gate interface> ] <msc body>  
     { **seq** <end> [ <inline gate interface> ] <msc body> } \*  
     **seq end** <end>

<par expr> ::=  
     **par begin** [ <inline expr identification> ] <end>  
     [ <inline gate interface> ] <msc body>  
     { **par** <end> [ <inline gate interface> ] <msc body> } \*  
     **par end** <end>

<loop boundary> ::=  
     <left angular bracket> <inf natural> [ , <inf natural> ]  
     <right angular bracket>

<inf natural> ::=  
     **inf** | <expression>

<inline expr identification> ::=  
     <inline expr name>

<inline gate interface> ::=  
     { **gate** <inline gate> <end> } +

<inline gate> ::=  
 <inline out gate> | <inline in gate> |  
 <inline create out gate> | <inline create in gate> |  
 <inline out call gate> | <inline in call gate> |  
 <inline out reply gate> | <inline in reply gate> |  
 <inline order out gate> | <inline order in gate>

## Concrete graphical grammar

<inline expression area> ::=  
 { <loop area> | <opt area> | <seq area> | <par area> | <alt area> | <exc area> }  
 [ *top or bottom is attached to top or bottom*  
 { { <int symbol> | <abs time symbol> } \* } *set* ]  
 [ *is attached to* <msc symbol> ]  
 [ *is followed by* <general name area> ]

<loop area> ::=  
 <inline expression symbol> [ *is attached to* <time interval area> ] *contains*  
 { **loop** [ <loop boundary> ] <operand area> }  
*is attached to* { <instance axis symbol> \* } *set*  
*is attached to* { <inline gate area> \* | <inline order gate area> \* } *set*

<opt area> ::=  
 <inline expression symbol> [ *is attached to* <time interval area> ] *contains*  
 { **opt** <operand area> }  
*is attached to* { <instance axis symbol> \* } *set*  
*is attached to* { <inline gate area> \* | <inline order gate area> \* } *set*

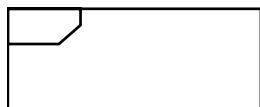
<exc area> ::=  
 <exc inline expression symbol>  
 [ *is attached to* <time interval area> ] *contains*  
 { **exc** <operand area> }  
*is attached to* { <instance axis symbol> \* } *set*  
*is attached to* { <inline gate area> \* | <inline order gate area> \* } *set*

<seq area> ::=  
 <inline expression symbol> [ *is attached to* <time interval area> ] *contains*  
 { **seq** <operand area>  
 { *is followed by* <separator area> *is followed by* <operand area> } \* }  
*is attached to* { <instance axis symbol> \* } *set*  
*is attached to* { <inline gate area> \* | <inline order gate area> \* } *set*

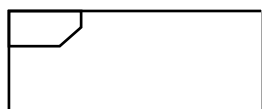
<par area> ::=  
 <inline expression symbol> [ *is attached to* <time interval area> ] *contains*  
 { **par** <operand area>  
 { *is followed by* <separator area> *is followed by* <operand area> } \* }  
*is attached to* { <instance axis symbol> \* } *set*  
*is attached to* { <inline gate area> \* | <inline order gate area> \* } *set*

<alt area> ::=  
 <inline expression symbol> [ *is attached to* <time interval area> ] *contains*  
 { **alt** <operand area>  
 { *is followed by* <separator area> *is followed by* <operand area> } \* }  
*is attached to* { <instance axis symbol> } \* *set*  
*is attached to* { <inline gate area> \* | <inline order gate area> \* } *set*

<inline expression symbol> ::=



<exc inline expression symbol> ::=



```

<operand area> ::=
    { <event layer> | <inline gate area> | <inline order gate area> } * set
    [is followed by <general name area>]

<separator area> ::=
    <separator symbol>

<separator symbol> ::=
    -----

```

## Static requirements

The <inline expression area> may refer to just one instance, or be attached to several instances. If a shared inline expression crosses an instance, which is not involved in this inline expression, it is optional to draw the instance axis through the inline expression.

All exception expressions must be shared by all instances in the MSC.

Extra-global inline expressions are those having the keyword **external** in the textual notation or crossing the MSC frame in the graphical notation. The latter graphical situation is described in the grammar as being "*attached to* <msc symbol>". Extra-global expressions must also cover all instances in the MSC.

Data expressions defining loop boundaries may contain static variables and wildcards, but must not contain dynamic variables.

## Semantics

The operator keywords **seq**, **alt**, **par**, **loop**, **opt** and **exc** which in the graphical representation are placed in the left upper corner denote respectively alternative composition, sequential composition, parallel composition, iteration, optional region and exception. In the graphical form a frame encloses the operands, the dashed lines denote operand separators.

The **seq** operator represents the weak sequencing operation. A guarded **seq** operand with a *false* guard is dynamically illegal.

The **alt** operator defines alternative executions of MSC sections. This means that if several MSC sections are meant to be alternatives only one of them will be executed. In the case where alternative MSC sections have common preamble the choice of which MSC section will be executed is performed after the execution of the common preamble. Alternative operands with a guard that evaluates to *false* cannot be chosen. If all the operands have false guards, no legal trace can go through this **alt**-expression, i.e., it is dynamically illegal.

One operand of an **alt**-expression may be guarded by **otherwise**. **Otherwise** is interpreted as the conjunction of the negations of the guards of all the other operands. Thus the **otherwise** guard is *true* only if the guards of all other operands of the alternative expression are *false*. An operand without guard is considered to have a *true* guard and the **otherwise** branch will be impossible to reach.

The **par** operator defines the parallel execution of MSC sections. This means that all events within the parallel MSC sections will be executed, but the only restriction is that the event order within each section will be preserved. Parallel operands with a guard that evaluates to *false* are excluded from the parallel composition. If all operands have *false* guards, the **par**-expression evaluates to **empty**.

The **loop** construct can have several forms. The most basic form is "**loop** < n, m >" where n and m are expressions of type natural numbers. This means that the operand may be executed at least n times and at most m times. The expressions may be replaced by the keyword **inf**, like "**loop** < n, **inf** >". This means that the loop will be executed at least n times. If the second operand is omitted like in "**loop** < n >" it is interpreted as "**loop** < n, n >". Thus "**loop** < **inf** >" means an



infinite loop. If the loop bounds are omitted like in "**loop**", it will be interpreted as "**loop** < 1, **inf**>". If the first operand is greater than the second one, the loop will be executed 0 times. The passes of a loop are connected by means of the weak sequential composition.

When the loop operand is guarded, the loop is terminated when the guard is *false* and continued when the guard is *true* as long as the upper bound has not been reached. Thus a loop will equal an empty MSC if the guard is false the first time the loop is entered. If the lower boundary of the loop is not reached due to the guard, the whole loop is interpreted as dynamically illegal. The upper boundary represents an upper limit to the number of iterations of the loop.

The **opt** operator is the same as an alternative where the second operand is the empty MSC. A guarded **opt** expression will always go through the option operand if the guard is *true*.

The **exc** operator is a compact way to describe exceptional cases in an MSC. The meaning of the operator is that either the events inside the <exc inline expression symbol> are executed and then the MSC is finished or the events following the <exc inline expression symbol> are executed. The **exc** operator can thus be viewed as an alternative where the second operand is the entire rest of the MSC. All exception expressions must be shared by all instances in the MSC. The exception expression is a shorthand for an alternative expression where the rest of the enclosing frame is the second operand.

In the textual representation, the optional <inline gate interface> defines the messages entering or leaving the inline expression via gates. By means of message name identification and optional gate name identification, the <inline gate interface> also defines the direct message connection between two inline expressions.

Extra-global inline expressions are associated with corresponding inline expressions on the enclosing instance. This means that when interpreted as decomposition, the inline expression will combine its operands one by one with the operands of other inline expressions. See 7.4 for more details. In the context of the innermost enclosing MSC document an extra-global inline expression is interpreted just as any other inline expression.

The time constraints referring to the start and end event of an inline expression are graphically attached to the <inline expression symbol> and given in the textual syntax after the keywords **loop end**, **opt end**, **exc end**, **alt end**, **seq end**, and **par end**. Time constraints referring either to the start or the end of an inline expression are **attached to top or bottom** of the <inline expression symbol>. In the textual syntax, the keywords **top** and **bottom** are used to locate a time constraint. A time constraint attached to the top of an in-line expression refers dynamically to the first event that occurs within the expression, and conversely a time constraint attached to the bottom of an in-line expression refers dynamically to the last event that occurs within the in-line expression. See Section 6 for more explanation of the time constructs.

### 7.3 MSC reference

MSC references are used to refer to other MSCs of the MSC document. The MSC references are objects of the type given by the referenced MSC.

MSC references may not only refer to a single MSC, but also to MSC reference expressions. MSC reference expressions are textual MSC expressions constructed from the operators **alt**, **par**, **seq**, **loop**, **opt**, and **exc**, and MSC references with possible actual parameters.

The **alt**, **par**, **seq**, **loop**, **opt** and **exc** operators are described in 7.2. The **seq** operator denotes the weak sequencing operation where only events on the same instance are ordered.

Plain MSC references may have actual parameters that must match the corresponding parameter declarations of the MSC definition.

The actual gates of the MSC reference may connect to corresponding constructs in the enclosing MSC. By corresponding constructs we mean that an actual message gate may connect to another actual message gate or to an instance or to a message gate definition of the enclosing MSC. Furthermore an actual order gate may connect to another actual order gate, or an orderable event or an order gate definition.

The unrestricted use of gates in an MSC reference can lead easily to the construction of MSCs that contain unexpected deadlocks. Therefore, we define some rules that restrict the graphical use of gates. The basic rule is that the graphical vertical order of gated messages, etc., appearing as gate definitions in a referenced MSC must be reproduced by the actual gated messages, etc., appearing in the referring MSC. This rule prevents the gate equivalent of drawing messages arrows with an upward slope. The complete set of rules is given in the section on graphic Static Requirements.

We also note that the appearance of a gated message on an MSC reference does not guarantee that this message will appear in all traces (indeed in any trace). For example, the gated message may originate and terminate inside optional in-line expressions within their defining MSCs.

Timed MSC references are used to constrain or to measure the execution time of referenced MSC and MSC reference expressions.

Time intervals may refer to the start or/and the end of an MSC reference. The start value is when the first event dynamically takes place, and the end is when the last event takes place.

### Concrete textual grammar

```

<shared msc reference> ::=
    reference [ <msc reference identification> : ]
    <msc ref expr> <shared> <end>
    [ time <time interval> <end> ]
    [ top <time dest list> <end> ]
    [ bottom <time dest list> <end> ]
    <reference gate interface>

<msc reference> ::=
    reference [ <msc reference identification>: ]
    <msc ref expr> <end>
    [ time <time interval> <end> ]
    [ top <time dest list> <end> ]
    [ bottom <time dest list> <end> ]
    <reference gate interface>

<msc reference identification> ::=
    <msc reference name>

<msc ref expr> ::=
    <msc ref par expr> { alt <msc ref par expr> }*

<msc ref par expr> ::=
    <msc ref seq expr> { par <msc ref seq expr> }*

<msc ref seq expr> ::=
    <msc ref ident expr> { seq <msc ref ident expr> }*

<msc ref ident expr> ::=
    loop [ <loop boundary> ] <msc ref ident expr> |
    exc <msc ref ident expr> |
    opt <msc ref ident expr> |
    empty |
    <parent>* <msc name> [<actual parameters>] |
    <left open> <msc ref expr> <right open>

<actual parameters> ::=
    <left open> <actual parameters list> <right open>

<actual parameters list> ::=
    <actual parameters block> [ <end> <actual parameters list> ]

```

```

<actual parameters block> ::=
    <actual data parameters>
    | <actual instance parameters>
    | <actual message parameters>
    | <actual timer parameters>

<actual instance parameters> ::=
    inst <actual instance parm list>

<actual instance parm list> ::=
    <actual instance parameter> [ , <actual instance parm list>]

<actual instance parameter> ::=
    <instance name>

<actual message parameters> ::=
    msg <actual message list>

<actual message list> ::=
    <message name> [ , <actual message list>]

<actual timer parameters> ::=
    timer <actual timer list>

<actual timer list> ::=
    <timer name> [ , <actual timer list>]

<parent> ::=
    #

<reference gate interface> ::=
    { <end> gate <ref gate> }*

<ref gate> ::=
    <actual out gate> | <actual in gate> |
    <actual order out gate> | <actual order in gate> |
    <actual create out gate> | <actual create in gate> |
    <actual out call gate> | <actual in call gate> |
    <actual out reply gate> | <actual in reply gate>

```

## Static requirements

An MSC reference must attach to every instance present in the enclosing diagram which is contained in the MSC that the MSC reference refers. If two diagrams referenced by two MSC references in an enclosing diagram share the same instances, these instances must also appear in the enclosing diagram.

The interface of the MSC reference must match the interface of the MSCs referenced in the expression, i.e., any gates attached to the reference must have a corresponding gate definition in the referenced MSCs. The correspondence is given by the direction and name of the message associated with the gate and if present by the gate name which is unique within the referenced expression.

In case where <msc ref expr> consists of a textual operator expression instead of a simple <msc name> and when more than one MSC reference refer the same MSC, the optional <msc reference name> in <msc reference identification> has to be employed in order to address an MSC reference in the message definition (see 4.3).

The <reference gate interface> must list all gates of the diagram.

## Concrete graphical grammar

<msc reference area> ::=  
    <msc reference symbol>  
    [ *top or bottom is attached to top or bottom*  
      { {<int symbol> | <abs time symbol> }\* } *set*]  
    *contains* { <msc ref expr> [**time** <time interval>]  
      [ <actual gate area>\* ] } *set*  
    *is attached to* { <instance axis symbol>\* } *set*  
    *is attached to* { <actual gate area>\* } *set*

<msc reference symbol> ::=



The <msc reference area> may be attached to one or more instances. If a shared <msc reference area> crosses an <instance axis symbol> which is not involved in this MSC reference the <instance axis symbol> is drawn through.

### Static requirements

The graphic order of the formal gates of the MSC definition referred by the MSC reference must be preserved by the actual gates of the MSC reference. In the textual notation this means that the order of the <msc gate interface> is the same as the order of the actual gates in the <reference gate interface>.

When an MSC reference contains reference expressions the situation is more complex. In general we have to build up a set of possible vertical orders from the constituent parts of the MSC expression. The actual vertical order used on the MSC reference must then be amongst this set.

When an expression consists of just an MSC name, the set of vertical orderings is taken to consist of just one element, namely the vertical order defined by the referenced MSC.

For an alternative expression each operand must define the same set of vertical orderings, which is then taken as the set representing the expression. This definition also covers optional and exceptional expressions, which are taken as shorthand forms of alternatives.

In the case of a parallel expression the set of vertical orderings is constructed by interleaving the orderings defined by each operand. For example, if we have  $e$  **par**  $f$ , where  $e$  defines the set of vertically ordered gates {< a, b >, < b, a >} and  $f$  defines the set {< u, v >}, then we have to interleave < u, v > with < a, b > to give six derived orderings, and also interleave < u, v > with < b, a >, to give a further 6 orderings. Thus  $e$  **par**  $f$  defines a set of twelve possible vertical orderings. So the actual gate ordering on the MSC reference must be one of these twelve orderings.

The set of orderings defined by  $e$  **seq**  $f$  is given by appending each of the orderings defined by  $f$  onto the end of each of the orderings defined by  $e$ . For example, if  $e$  defines the set of vertically ordered gates {< a, b >, < b, a >} and  $f$  defines the set {< u, v >}, then the resulting set is {< a, b, u, v >, < b, a, u, v >}.

MSC references must not directly or indirectly refer to their enclosing MSC, i.e., recursion is forbidden.

MSC actual parameters must match the corresponding parameter declarations of the MSC definition. The elements separated by <end> in the <actual parameters> must correspond one-to-one with the elements separated by <end> in the <msc parameter decl> of the MSC definition, but the order of blocks may be different. Within a parameter block, the order of individual parameters must be the same between declaration and use.

Actual instance parameters must have the same kind as the instance parameter declaration. The kind of the actual instance parameter may be of a kind inherited from the kind of the instance parameter declaration.

Actual messages must have the same message signature as the message of the message parameter declaration. To have the same signature for a message means that the list of parameters to the message is the same for the actual message and the message parameter declaration.

Actual timers must have the same timer signature as the timer of the timer parameter declaration. To have the same signature for a timer means that the list of parameters of the timer is the same for the actual timer and the timer parameter declaration.

An MSC containing references and actual parameter bindings is illegal if, after repeatedly expanding the references and binding actual parameters, an illegal MSC results.

## Semantics

Each MSC can be seen as a definition of an MSC type. MSC types can be used in other MSC types through MSC references.

An MSC type may be attached to its environment via gates. Gates are used to define the connection points in case where an MSC type is used in another type. Gate identifiers may be associated to the connection points in form of names.

Instances that are attached to the MSC reference are default actual instance parameters to the MSC reference. This means that if the instance kind is unambiguously identifying the formal instance parameter, the attached instances need not appear in the actual parameter list.

An MSC reference may be attached to instances which are not contained in the referenced diagram.

In general, the MSC reference may refer to a textual MSC expression. In the simple case, where the MSC expression consists of an MSC name only, the MSC reference points to a corresponding MSC type definition. The correspondence is given by the MSC name, which is unique within the MSC document. When the MSC has actual parameters, the result is that the formal parameters of the referenced diagram are replaced by the actual parameters of the MSC reference.

In the textual representation, the <reference gate interface> defines the messages entering or leaving the MSC reference via gates. By means of message name identification and optional gate name identification, the <reference gate interface> also defines the direct message connection between two MSC references.

An MSC reference with the keyword **empty** refers to an MSC with no events and no instances.

A <parent> prefix of an MSC reference indicates that the MSC reference refers to the MSC which it redefines through inheritance rather than the MSC with that name within the inheriting instance kind. The <parent> prefix can be repeated in order to access grandparents etc.

The time constraints referring to the start and end event of an MSC expression are graphically attached to the <msc reference symbol> and given in the textual syntax after the keyword **time**. Time constraints referring either to the start or the end of an MSC expression are **attached to top or bottom** of the <msc reference symbol>. In the textual syntax the keywords **top** and **bottom** are used to indicate that the start event of the MSC expression is before or after certain events, or that the end event of the MSC expression is before or after certain events, respectively.

## 7.4 Instance decomposition

An MSC document contains a set of instances. Each instance is of an instance kind. Instances that have no reference to the instance kind implicitly have the instance kind with the same name as the instance. In order to describe interaction on different levels of detail MSC introduces decomposition.

The inner structure and behaviour of an instance kind is defined through an MSC document with the same name as the instance kind. Thus there will be a hierarchy of MSC documents defining the instance hierarchy. To indicate how the behaviours of the different levels are related, the behaviour of an instance inside an MSC diagram can be specified to be refined in an MSC of the MSC document defining the instance being decomposed.

Thus an MSC document may be interpreted relative to its own instances only, disregarding any decomposition, or it may be interpreted relative to lower levels of instances by following the decomposition relations.

MSC requires that there is a structural similarity between the decomposed instance and the corresponding decomposition, but there is no requirement that there should be some behavioural refinement.

### Concrete textual grammar

```

<decomposition> ::=
    decomposed [ <substructure reference> ]

<substructure reference> ::=
    as <message sequence chart name>
  
```

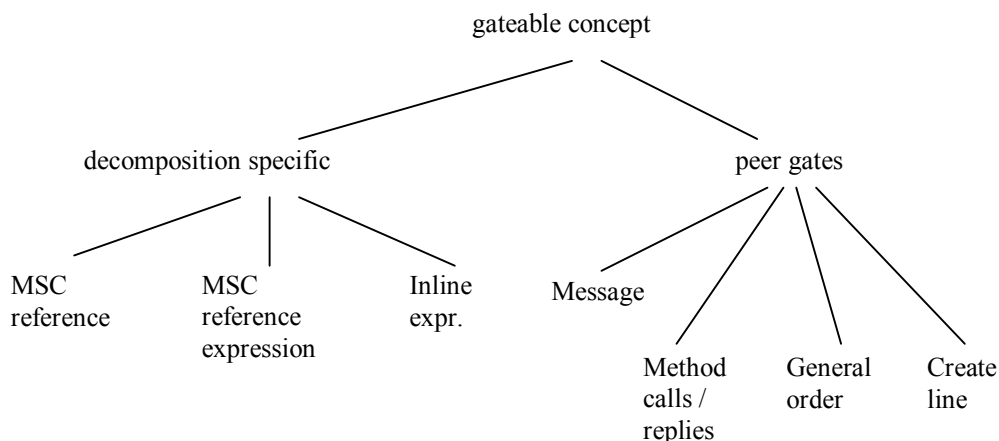
### Concrete graphical grammar

The graphical grammar is given in Section 4.2.

### Static requirements

For each instance having the keyword **decomposed** a corresponding refining MSC document has to be specified with the same name as the decomposed instance. If the <substructure reference> is added after **decomposed** then the MSC document must contain a refining MSC having the name specified as the <substructure reference>; the refining MSC will appear in the same utility/defining part of the refining MSC document as that of the MSC containing the decomposed instance does in its MSC document.

The decomposed instance can be understood as a sequence of language constructs of which some should be interpreted as gates relative to the decomposition diagram. An overview of the gateable concepts are given in Figure 11. There are *peer gates* and *decomposition gates*. Peer gates are known also from the connection of MSC references, while the decomposition gates are only relevant for decomposed instances.



**Figure 11 – Gateable concepts**

The interpretation of the decomposition specific gate concepts is given in the semantics section below. The interpretation of the peer gates is given in the chapter on gates and on method calls.

Statically the graphic order of the gates must be preserved from the decomposed instance to the decomposition diagram, meaning that if the two vertical edges of the decomposition diagram are superimposed the sequence of the gate definitions of the diagram must be the same as that of the decomposed instance. In the textual notation this means that the order of the <msc gate interface> is the same as the order of the corresponding events on the decomposed instance.

Take instance  $i$  in MSC  $M$  in MSC document  $D$  of Figure 12. Along the instance there is in sequence an alternative expression, a simple MSC reference (to  $A$ ), an output of message  $s$  and an MSC reference expression ( $B$  **alt**  $C$ ). The decomposition is found in MSC  $iM$  of Figure 16 and it contains in sequence an extra-global inline expression, a simple MSC reference (to  $iA$ ), an output peer gate of message  $s$  and an MSC reference expression ( $iB$  **alt**  $iC$ ).

With the peer gates this sequencing is simple as there are events on the decomposed instance to match peer gates on the decomposition. With decomposition specific gateable concepts the situation is slightly more involved. The constructs on an instance that are not matched by a gateable concept are disregarded from the static requirements on the decomposition. Such constructs are actions, timers and coregions. Coregions are in this respect considered a plain instance area.

When the decomposed instance is stopped, the decomposition must include stops on all its contained instances.

### Static requirements

The following static requirement must hold for MSC references covering a given instance. Let  $i$  be a decomposed instance inside MSC  $M$  (Figure 12). Let  $i$  in  $M$  be decomposed as  $iM$  (Figure 16). Let there be an MSC reference  $A$  covering  $i$  inside  $M$ . This  $A$  is then a gateable MSC reference. It must be matched by a corresponding global<sup>2</sup> reference in  $iM$  to (say)  $iA$  (Figure 17) defined in the MSC document defining  $i$ .  $i$  inside  $A$  must then be decomposed as  $iA$  (Figure 13).

The following static requirement holds for inline expressions covering a given decomposed instance (see Figure 12). The decomposition must contain a corresponding inline expression of the same operation and operand structure (see Figure 16). The inline expression in the decomposition must be extra-global (see inline expressions) indicating that the operands are connected to other operands of similar inline expressions when interpreted through decomposition. When interpreted only in the context of the closest enclosing MSC document and extra-global inline expression is treated as a plain global inline expression.

Recursively every operand of the inline expression have the same static requirements as the whole decomposed instance.

The following static requirement holds for MSC reference expressions covering a given decomposed instance. The decomposition must contain a corresponding MSC reference expression with the same expression structure. Each of the operands of the MSC reference expression must obey the static requirements for MSC reference expressions or MSC references.

An instance that is decomposed in one MSC must be decomposed in all MSCs that are dependent upon this MSC through referencing.

---

<sup>2</sup> By "global" we mean "covering all instances in the MSC document".

## Semantics

The semantics of decomposition is defined through a model of transformation. The following is an algorithm that will transform an MSC including decomposition of one MSC document to another MSC of another (constructed) MSC where the decomposed instance has been split into its components. An example will accompany the transformations.

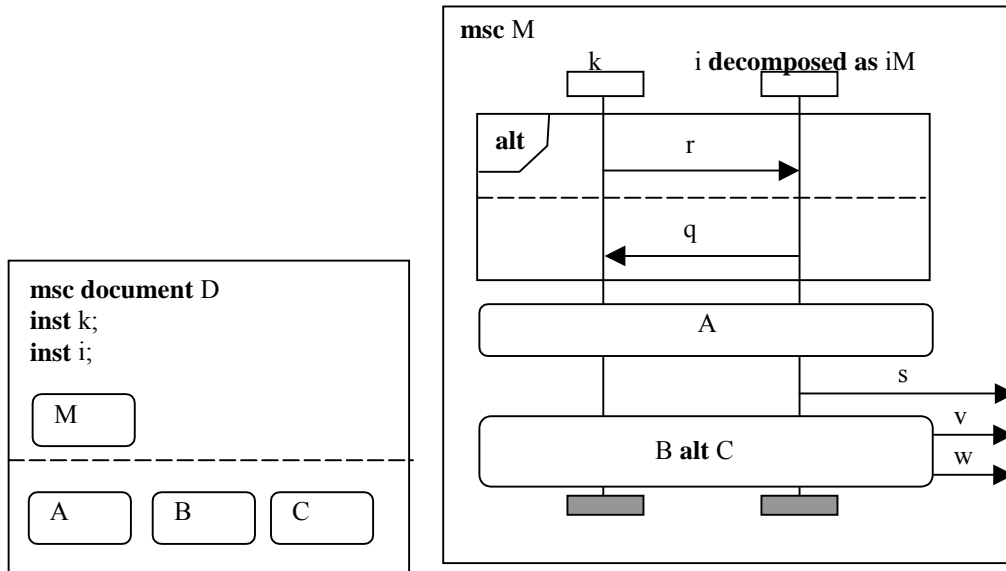


Figure 12 – Top level MSC document and defining MSC

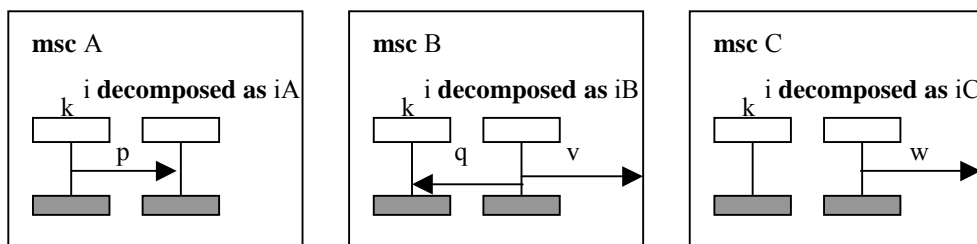
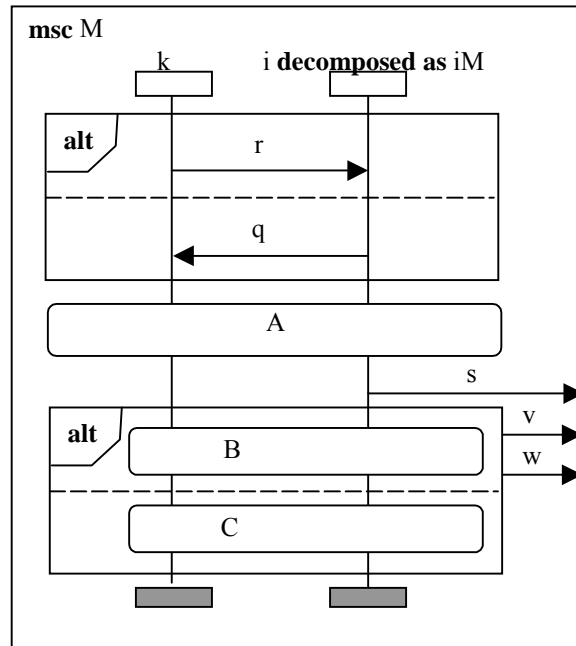


Figure 13 – Utilities on upper level

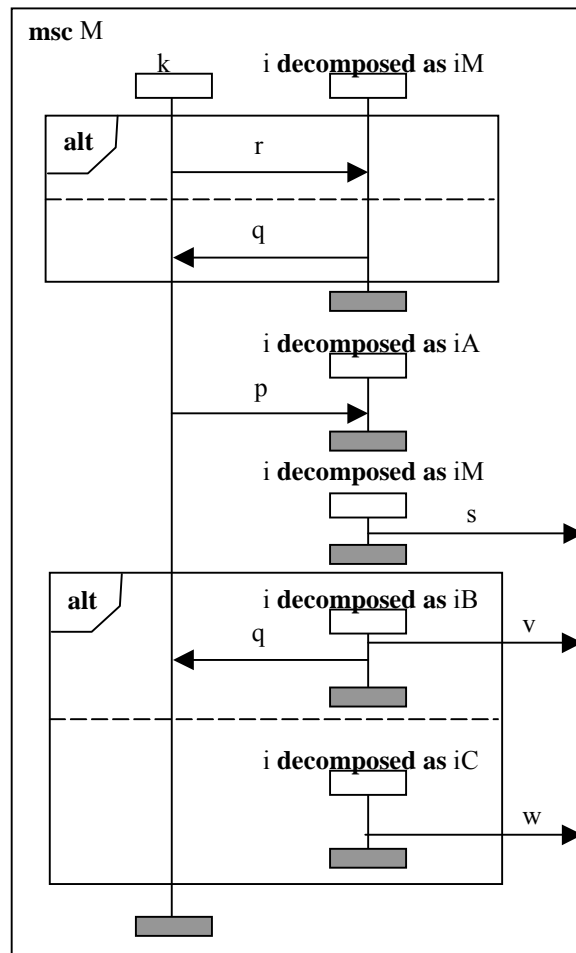
*Rule 1.* Transform MSC reference expressions to inline expressions.





**Figure 14 – Having used Rule 1**

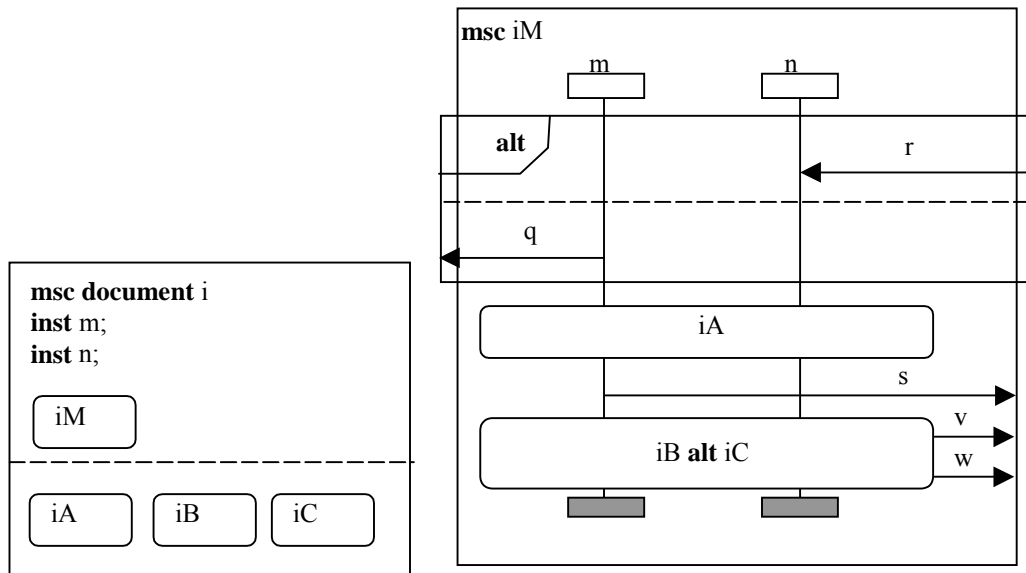
*Rule 2:* Resolve the MSC references by substituting the contents of the MSC diagrams wherever the references appear. Match the gates.



**Figure 15 – Having used Rule 2**

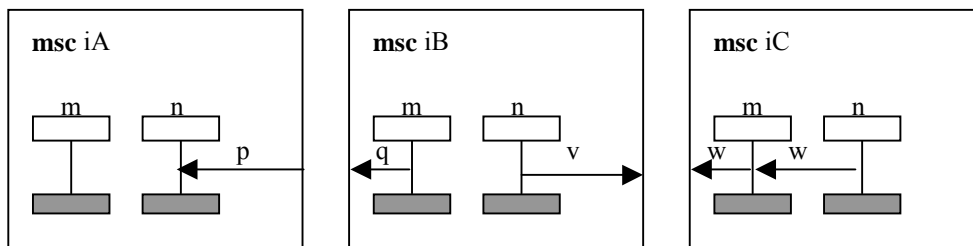
The algorithm has now reached a diagram where the decompositions occur piecewise and according to the static requirements. The instances that are not decomposed are trivially connected through gates.

Now the algorithm turns to the decomposition and there is a need to show the definition of the MSC document *i*.



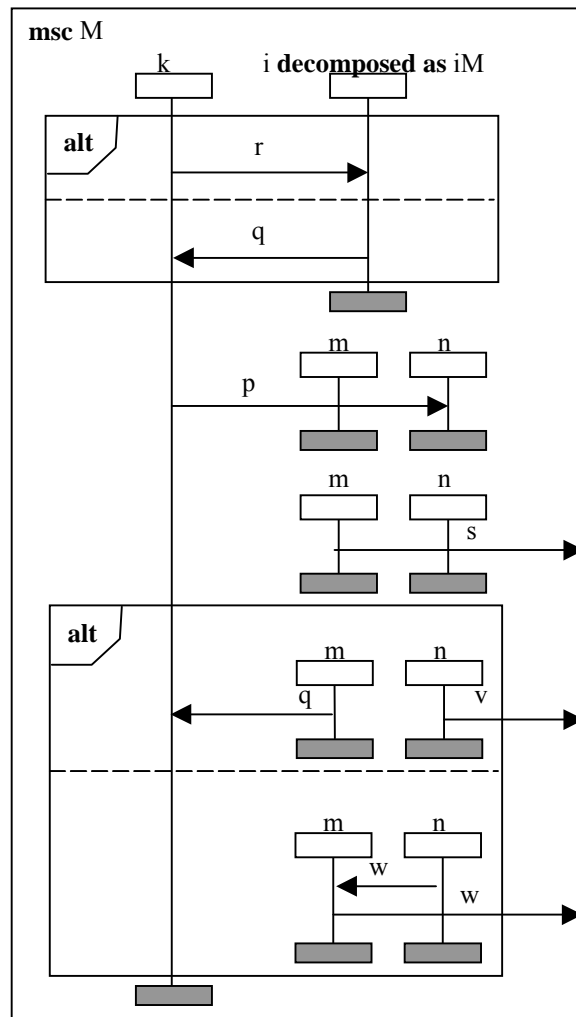
**Figure 16 – Lower level MSC document**

Please notice the extra-global inline expression and that the static requirements are fulfilled in *iM* relative to *i* in *M* of MSC document *D* in Figure 12.



**Figure 17 – Utilities on lowest level**

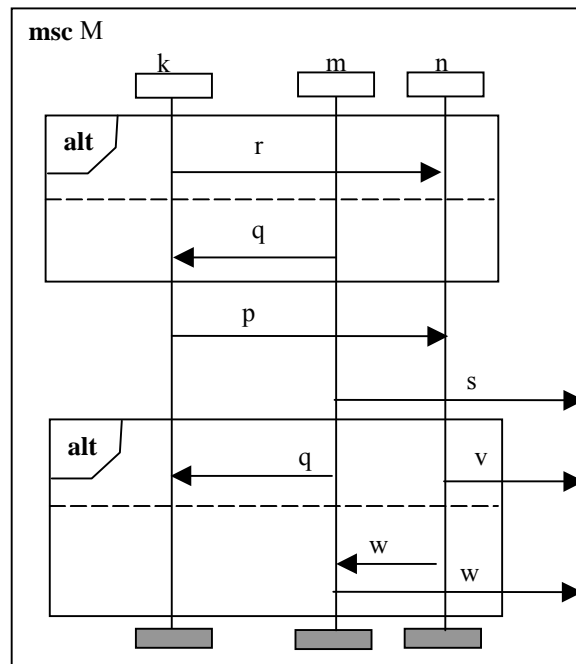
*Rule 3:* Transform decomposed instances with merely peer gates by pure substitution of the diagram contents and matching the peer gates.



**Figure 18 – Having used Rule 3**

Rule 3 gives instructions to substitute simple MSCs and match their gates. From the example one can see the substitutions and the substitutes. From the starting point in Figure 15 consider the decomposition of  $i$  in  $M$  given by  $iM$  found in Figure 16. From the top and down, the following substitutions occur. Firstly leave the inline expression to the next rule below. Secondly in  $iM$  there is the reference to  $iA$  which corresponds well with the fragment indicated. Use  $iA$  in Figure 17 for substitution. Thirdly there is the output of  $s$  decomposed directly in  $iM$ . Use the portion of  $iM$  between the reference to  $iA$  and the MSC reference expression as the substitute. Fourthly arrive at the MSC reference expression ( $iB \text{ alt } iC$ ) and find that this corresponds well with the inline expression. Perform substitution according to the same principles within each operand.

**Rule 4:** Transform inline expressions by substituting with the corresponding extra-global inline expressions such that the operands are connected in a one-one-relation. When there are nested expressions, this rule is applied recursively on every operand.



**Figure 19 – Having applied Rule 4**

In Figure 19 Rule 4 has been applied and the extra-global expression has been substituted in operand by operand. Furthermore the diagram has been simplified by combining the fragments to one instance each.

The resulting MSC M would have been an MSC in an MSC document D' that would contain instances k, m, n.

The transformation scheme could equally well have been expressed by following the decomposition first before resolving the references. More notation would have been needed for the intersection between an MSC reference (expression) and decomposition diagram contents.

### 7.5 High-level MSC (HMSC)

High-level MSCs provide a means to graphically define how a set of MSCs can be combined. An HMSC is a directed graph where each node is either:

- a start symbol (there is only one start symbol in each HMSC),
- an end symbol,
- an MSC reference,
- a condition,
- a connection point, or
- an HMSC in-line expression frame.

The flow lines connect the nodes in the HMSC and they indicate the sequencing that is possible among the nodes in the HMSC. The incoming flow lines are always connected to the top edge of the node symbols whereas the outgoing flow lines are connected to the bottom edge. If there is more than one outgoing flow line from a node this indicates an alternative.

The MSC references can be used either to reference a single MSC or a number of MSCs using a textual MSC expression.

The conditions in HMSCs can be used to indicate global system states or guards and impose restrictions on the MSCs that are referenced in the HMSC.

The HMSC in-line expression frames permit the composition of constituent HMSC diagrams in the same way as MSC in-line expressions permit the composition of basic MSC fragments. The frame contains the operator name in the upper left hand corner, such as **par**, **alt**, **loop**, etc, and can be subdivided into separate operand areas in the same way as in-line expressions.

The connection points are introduced to simplify the layout of HMSCs and have no semantical meaning.

High-level MSCs can be constrained and measured with time intervals as for MSC expressions. In addition, the execution time of an HMSC in-line expression frame can be constrained or measured. The interpretation is similar to the interpretation of Timed MSC in-line expressions.

### Concrete textual grammar

```

<hmsc> ::=
    <hmsc body>

<hmsc body> ::=
    <hmsc statement>*

<hmsc statement> ::=
    <text definition> | <node definition>

<node definition> ::=
    <initial node> | <final node> | <intermediate node>

<initial node> ::=
    initial <connection list> <end>

<final node> ::=
    <label name> : final <end>

<connection list> ::=
    connect <label list>

<label list> ::=
    <label name> , <label list>

<intermediate node> ::=
    <label name> : <intermediate node type> <connection list> <end>

<intermediate node type> ::=
    <timeable node> | <untimeable node>

<untimeable node> ::=
    <hmsc connection node> | <hmsc condition node>

<hmsc connection node> ::=
    empty

<hmsc condition node> ::=
    <condition identification> [ <shared> ]

```

The <shared> part permits explicit shared conditions, which allows the instances covered by a condition to be explicitly declared. This is of relevance, for example, for data guards, which are allowed to cover one instance only. If the <shared> part is omitted then the guard has global instance scope.

```

<timeable node> ::=
    { <hmsc ref expr node> | <hmsc expression> }
    [ time <time interval> <end> ]
    [ top <time dest list> <end> ]
    [ bottom <time dest list> <end> ]

<hmsc ref expr node> ::=
    reference [ <msc reference identification> : ] <msc ref expr>

```

```

<hmsc expression> ::=
    {<hmsc loop expr> | <hmsc opt expr> | <hmsc alt expr> |
     <hmsc seq expr> | <hmsc par expr> | <hmsc exc expr> }

<hmsc loop expr> ::=
    loop [ <loop boundary> ] begin [ <inline expr identification> ] <end>
    <hmsc body>
    loop end

<hmsc opt expr> ::=
    opt begin [ <inline expr identification> ] <end>
    <hmsc body>
    opt end

<hmsc alt expr> ::=
    alt begin [ <inline expr identification> ] <end>
    <hmsc body>
    { alt <end> <hmsc body> }*
    alt end

<hmsc seq expr> ::=
    seq begin [ <inline expr identification> ] <end>
    <hmsc body>
    { seq <end> <hmsc body> }*
    seq end

<hmsc par expr> ::=
    par begin [ <inline expr identification> ] <end>
    <hmsc body>
    { par <end> <hmsc body> }*
    par end

<hmsc exc expr> ::=
    exc begin [ <inline expr identification> ] <end>
    <hmsc body>
    exc end

```

## Static requirements

Labels cannot create connections across frame or operand boundaries, so all referenced labels must be defined in the immediately enclosing HMSC body only. That is, a <label name> appearing in any <connection list> of a node definition statement must occur as a label of a node occurring within the local <hmsc body>. In particular, it must not refer to a label of a node occurring in:

- an operand of an inner <hmsc expression>,
- an outer <hmsc body> if the label reference occurs within an <hmsc expression> operand,
- another operand of an enclosing <hmsc expression>.

Therefore, label names can be reused outside the immediate local scope of <hmsc expression>s.

Every node in the graph defined by an <hmsc expression> must be reachable from its <initial node>s, i.e., each graph must be connected.

## Concrete graphical grammar

```

<hmsc area> ::=
    {<text layer> <node area>} set

<node area> ::=
    {<initial area> | <final area> | <intermediate area>} set

<initial area> ::=
    <hmsc start symbol> is followed by { <alt op area>+ } set

<final area> ::=
    <hmsc end symbol> is attached to { <hmsc line symbol>+ } set

```

```

<intermediate area> ::=
    <intermediate node area>
    is followed by { <alt op area> + } set
    is attached to { <hmsc line symbol>+ } set

<alt op area> ::=
    <hmsc line symbol> is attached to { <intermediate area> | <hmsc end symbol> }

<intermediate node area> ::=
    <timeable area> | <untimeable area>

<untimeable area> ::=
    <hmsc connection area> | <hmsc condition area>

<hmsc connection area> ::=
    <connection point symbol>

<hmsc condition area> ::=
    <condition symbol> contains <condition text> [ <shared> ]

```

The <shared> part permits explicit shared conditions, which allows the instances covered by a condition to be explicitly declared. This is of relevance, for example, for data guards, which are allowed to cover one instance only. If the <shared> part is omitted then the guard has global instance scope.

```

<timeable area> ::=
    <timeable node area>
    top or bottom is attached to top or bottom { <time interval area>* } set

<timeable node area> ::=
    <hmsc ref expr area> | <hmsc expression area>

<hmsc ref expr area> ::=
    <msc reference symbol> contains <msc ref expr> [ time <time interval> ]

<hmsc expression area> ::=
    { <hmsc loop area> | <hmsc opt area> | <hmsc seq area>
      | <hmsc par area> | <hmsc alt area> | <hmsc exc area> }
    [ is followed by <general name area> ]

<hmsc loop area> ::=
    <inline expression symbol> contains
    { loop [ <loop boundary> ] <hmsc operand area> }

<hmsc opt area> ::=
    <inline expression symbol> contains
    { opt <hmsc operand area> }

<hmsc seq area> ::=
    <inline expression symbol> contains
    { seq <hmsc operand area>
      { is followed by <hmsc operand area>
        is followed by <hmsc operand area> }* }

<hmsc par area> ::=
    <inline expression symbol> contains
    { par <hmsc operand area>
      { is followed by <hmsc operand area>
        is followed by <hmsc operand area> }* }

<hmsc alt area> ::=
    <inline expression symbol> contains
    { alt <hmsc operand area>
      { is followed by <hmsc operand area>
        is followed by <hmsc operand area> }* }


<hmsc exc area> ::=
    <exc inline expression symbol> contains
    { exc <hmsc operand area> }


```




<hmsc operand area> ::=  
 { <hmsc area> } \* *set*  
 [ *is followed by* <general name area> ]


The <separator symbol> line used in the <separator area> can be drawn horizontally or vertically, so long as within a <hmsc expression area> all <separator symbol>s are in the same direction.


<hmsc start symbol> ::=  


<hmsc end symbol> ::=  


<hmsc line symbol> ::=  
 <hmsc line symbol1> | <hmsc line symbol2>

<hmsc line symbol1> ::=  


<hmsc line symbol2> ::=  


<connection point symbol> ::=  


### Drawing rules

The <hmsc line symbol> may be bent and have any direction.

That the <hmsc start symbol> is followed by the <alt op area> means that the <hmsc line symbol>s must be attached to the lower corner of the <hmsc start symbol> as illustrated in Figure 20 where two line <hmsc line symbol>s follow an <hmsc start symbol> :

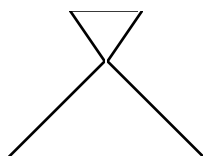


Figure 20 – HMSC drawing rules

That the <hmsc line symbol> is attached to another symbol in the <alt op area> production rule means that the <hmsc line symbol>s must be attached to the upper edge of the symbol in question. For the cases where the symbol is an <msc reference symbol> and an <hmsc end symbol> this is illustrated in the following example:

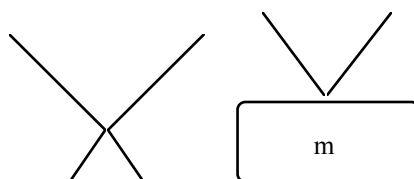
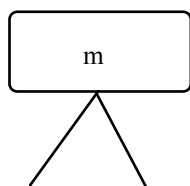


Figure 21 – HMSC drawing rules

That the <intermediate node> is followed by <hmsc line symbol>s in the <intermediate area> production rule means that the <hmsc line symbol>s must be attached to the lower edge of the symbol in the <intermediate node> as illustrated in the following example that shows how an <msc reference symbol> is followed by an <alt op area>:



**Figure 22 – HMSC drawing rules**

## **Semantics**

The graph describing the composition of MSCs within an HMSC is interpreted in an operational way as follows. There must be only a single <hmsc start symbol> within an <hmsc area>, and execution starts there. Next, it continues with a node that follows one of the outgoing edges of this symbol. These nodes are considered to be operands of an **alt** operator (see 7.2). After execution of the selected node, the process of selection and execution is repeated for the outgoing edges of the selected node. Execution of an end node means that execution of the given HMSC ends. Execution of an MSC reference is according to the description in 7.3. Execution of a connection point is an empty operation. Execution of an HMSC in-line expression frame consists of executing the operands of the frame as described in 7.2 for each of the operators. A sequential execution of two nodes that are related by an edge is described by the **seq** operator (see 7.3).

A guarding condition means that the execution may not continue beyond the condition if it evaluates to *false*. If all available branches are blocked by false guards, and no HMSC end has been reached, the whole HMSC has no legal traces. Where guards are used, the rules concerning the instances on which they are defined must be followed as defined in 4.7.

## **Message sequence chart examples**

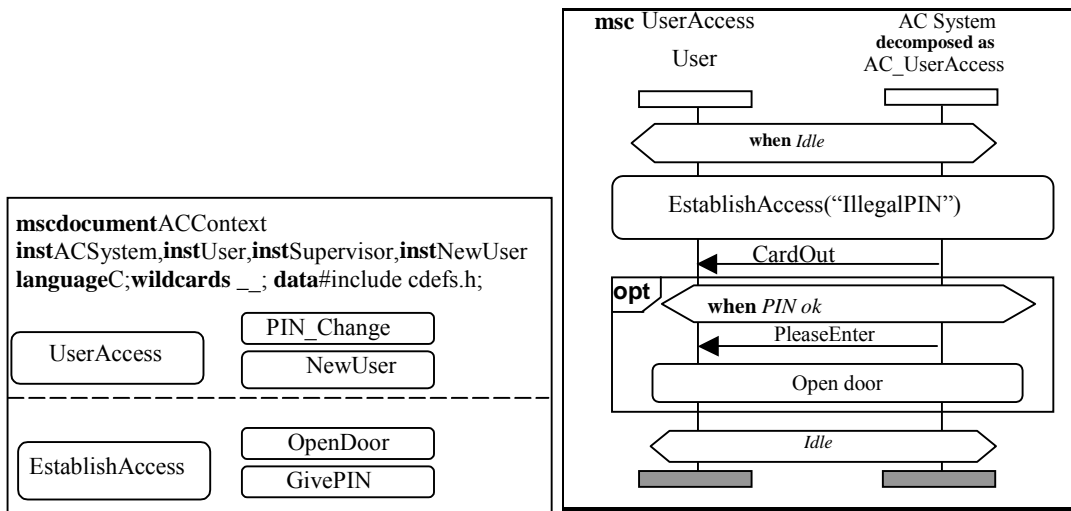
This chapter is informative rather than normative.

## **8 Message Sequence Chart Document**

### **8.1 MSC Documents**

The MSC document diagram in Figure 23 shows that the *ACContext* instance contains the instances *ACSystem*, *User*, *Supervisor* and *NewUser*. The defining (or public) MSC diagrams are *UserAccess*, *PIN\_Change* and *NewUser*. Notice that *NewUser* is the name of both a containing instance and a contained MSC diagram. This is legal since instances and diagrams are of different entity classes.

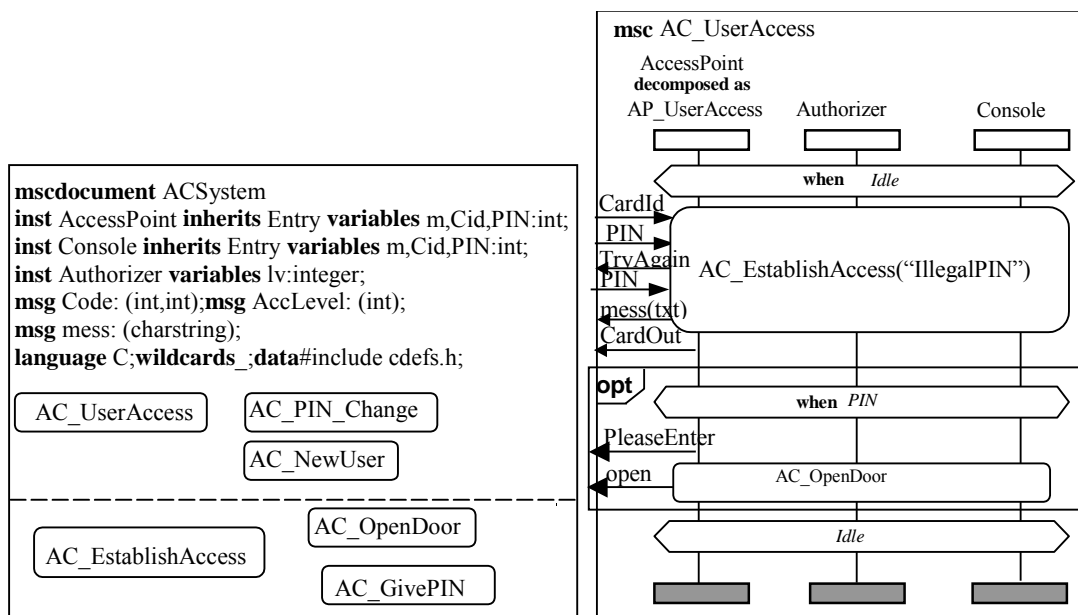
Furthermore the data language interface is given. The data language is "C" and the underscore character is used as variable wildcard. The data definitions are given in the file *cdefs.h*.



**Figure 23 – MSC document ACContext**

To illustrate the static requirements for consistency when decomposing we have also given the MSC diagram *UserAccess* referenced from MSC document *ACContext*.

## 8.2 Instance decomposition



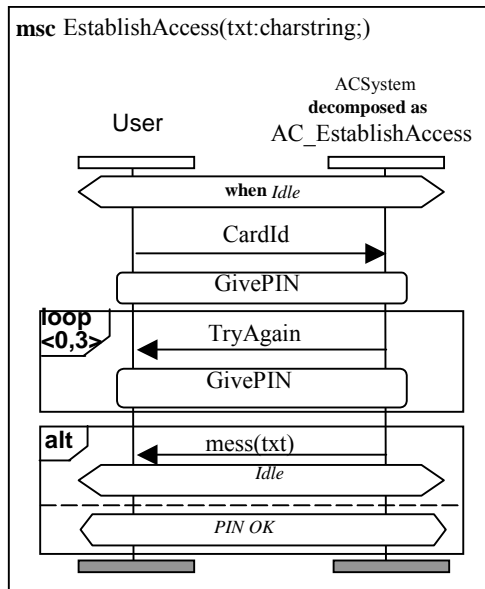
**Figure 24 – MSC document on next level down**

The MSC document *ACSystem* given in Figure 24 is then the description of the instance *ACSystem* given in Figure 23. The description shows also inheritance described for the contained instances. Below in Figure 27 we have shown the definition of the instance kind *Entry*.

We have also given dynamic variable declarations of the instances and the messages that have parameters.

The static requirements for decomposition and MSC references can be checked from the example. In *ACContext* we have the MSC *UserAccess* where the contained instance *ACSystem* is decomposed by *AC\_UserAccess* which in turn refers to *AC\_EstablishAccess*.

On the other hand the *UserAccess* refers to *EstablishAccess* shown in Figure 25.



**Figure 25 – MSC EstablishAccess**

Inside *EstablishAccess* which is a utility of *ACContext* the instance *ACSystem* is decomposed into *AC\_EstablishAccess*. *AC\_EstablishAccess* thus represents the confluence points of the referencing and decomposition from *UserAccess* of *ACContext*. In other words, whether decomposition or referencing is followed first, there should be no difference in the end.

In *EstablishAccess* there is a definition of a static variable *txt*. The actual parameter can be seen in Figure 24.

For completeness *AC\_EstablishAccess* is shown in Figure 26.

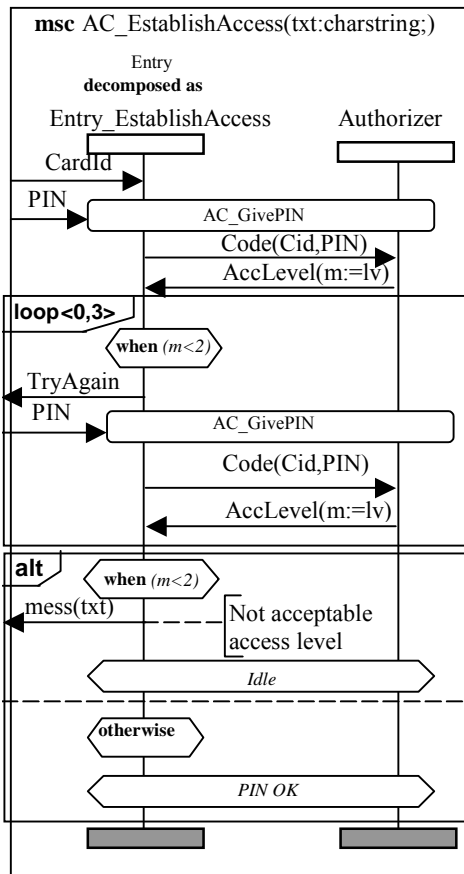


Figure 26 – MSC AC\_EstablishAccess

Within *AC\_EstablishAccess* there are both state-like conditions used as guards and setting. There are also Boolean expressions used as guards. Bindings are shown on some of the messages.

### 8.3 Instance inheritance

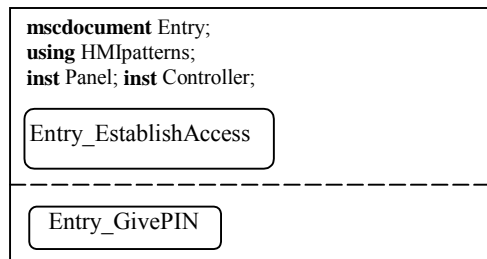


Figure 27 – MSC document Entry

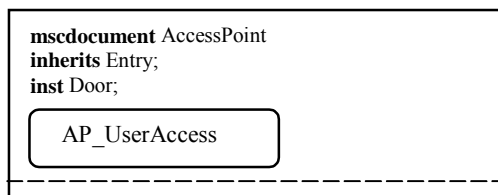


Figure 28 – MSC document AccessPoint inheriting from entry

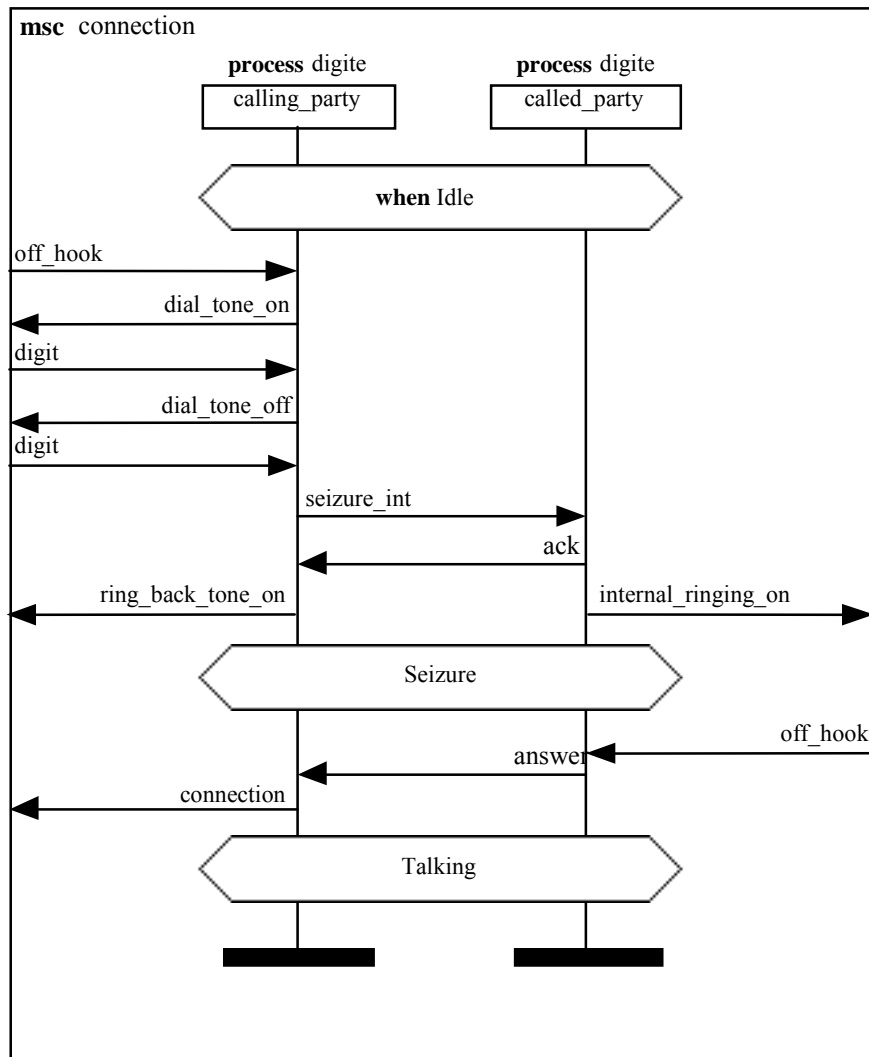
In the MSC document *AccessPoint* we see that there is no need to define again the instances *Panel* and *Controller* as they are already defined in the inherited *Entry*.

## 9 Simple Message Sequence Charts

### 9.1 Basic MSC

This example shows a simplified connection set up within a switching system. The example shows the most basic MSC-constructs: (process) instances, environment, messages, global conditions.

The Idle condition is an initial condition, the Seizure condition is intermediate, and Talking condition is a final condition.



**Figure 29 – MSC connection**

```

msc connection;
inst calling_party: process digite;
inst called_party: process digite;
gate out off_hook to calling_party;
gate in dial_tone_on from calling_party;
gate out digit to calling_party;
gate in dial_tone_off from calling_party;
gate out digit to calling_party;
gate in ring_back_tone_on from calling_party;
gate in internal_ringing_on from called_party;
gate out off_hook to called_party;
gate in connection from calling_party;

```

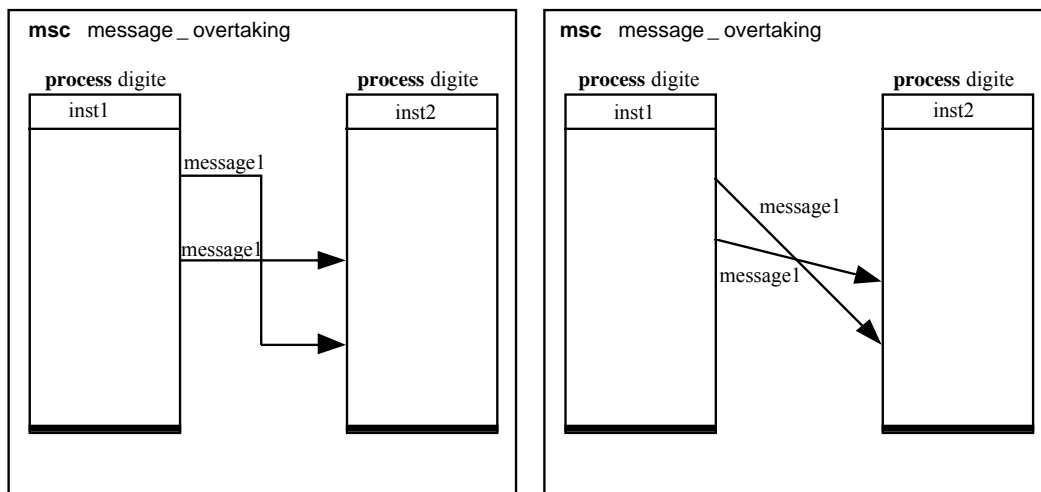
```

calling_party : instance process digite;
condition when Idle shared all;
in off_hook from env;
out dial_tone_on to env;
in digit from env;
out dial_tone off to env;
in digit from env;
out seizure_int to called_party;
in ack from called_party;
out ring_back_tone on to env;
condition Seizure shared all;
in answer from called_party;
out connection to env;
condition Talking shared all;
endinstance;
called_party: instance process digite;
condition when Idle shared all;
in seizure_int from calling_party;
out ack to calling_party;
out internal_ringing_on to env;
condition Seizure shared all;
in off_hook from env;
out answer to calling_party;
condition Talking shared all;
endinstance;
endmsc;

```

## 9.2 Message overtaking

This example shows the overtaking of two messages with the same message name 'message1'. In the textual representation the message instance names (a, b) are employed for a unique correspondence between message input and output. In the graphical representation messages either are represented by horizontal arrows, one with a bend to indicate overtaking or by crossing arrows with a downward slope.



**Figure 30 – MSC message\_overtaking**

```

msc message_overtaking;
inst inst1;
inst inst2;
inst1: instance process digite;
out message1, a to inst2;
out message1, b to inst2;
endinstance;
inst2: instance process digite;
in message1, b from inst1;
in message1, a from inst1;
endinstance;
endmsc;

```

### 9.3 MSC basic concepts

This example contains the basic MSC constructs: instances, environment, messages, conditions, actions and time-out. In the graphical representation both types of instance symbols are used: the single line form and the column form.

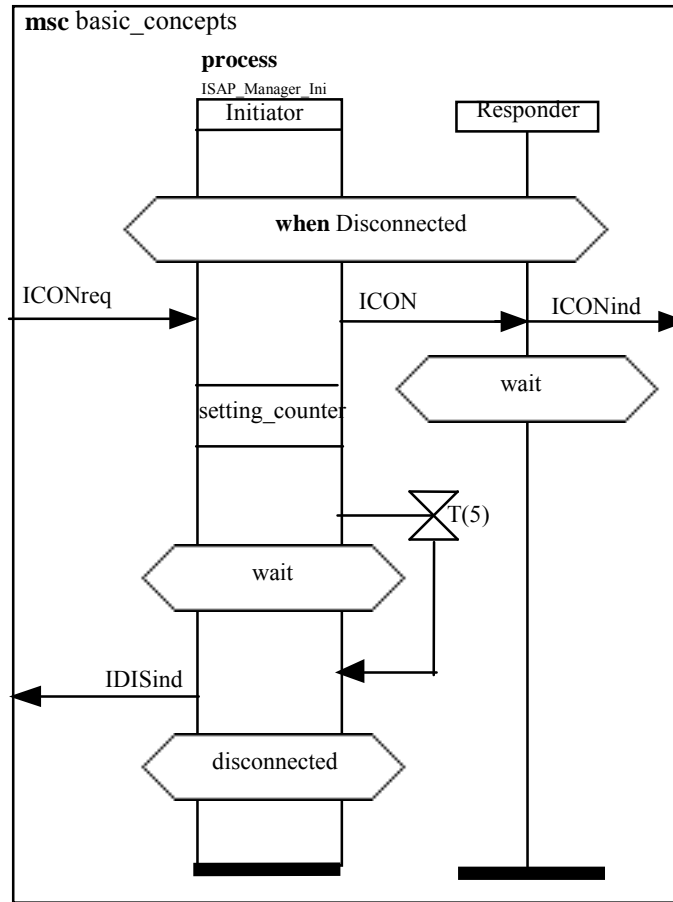


Figure 31 – MSC basic\_concepts

#### Instance oriented textual syntax:

```

msc basic_concepts;
inst Initiator: process ISAP_Manager_Ini;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;
gate in IDISind from Initiator;

Initiator: instance process ISAP_Manager_Ini;
condition when Disconnected shared all;
in ICONreq from env;
out ICON to Responder;
action 'setting_counter';
starttimer T (5);
condition wait shared;
timeout T;
out IDISind to env;
condition disconnected shared;
endinstance;
Responder: instance;
condition when Disconnected shared all;
in ICON from Initiator;
out ICONind to env;
condition wait shared;
endinstance;
endmsc;

```



## Event oriented textual syntax:

```

msc basic_concepts;
inst Initiator: process ISAP_Manager_Ini;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;
gate in IDISind from Initiator;

Initiator: instance process ISAP_Manager_Ini;
Responder: instance;
all: condition when Disconnected;
Initiator: in ICONreq from env;
out ICON to Responder;
Responder: in ICON from Initiator;
out ICONind to env;
condition wait;
endinstance;
Initiator: action 'setting_counter';
starttimer T (5);
condition wait;
timeout T;
out IDISind to env;
condition disconnected;
endinstance;

endmsc;

```

### 9.4 MSC-composition through labelled conditions

In this example the composition of MSCs by means of global conditions is demonstrated. The final global condition 'Wait\_For\_Resp' of MSC 'connection\_request' is identical with the initial global condition of MSC connection confirm. Therefore both MSCs may be composed to the resulting MSC 'connection' (example 9.5).

The composition is defined by means of the HMSC 'con\_setup' given in Figure 34.

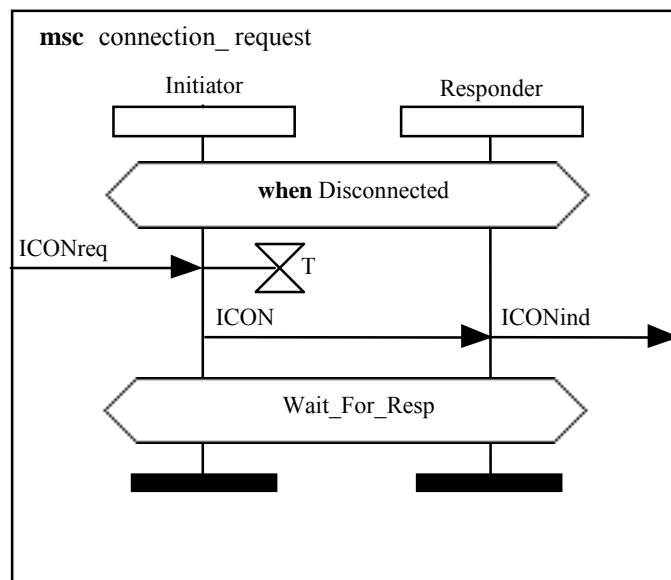


Figure 32 – MSC connection\_request

```

msc connection_request;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;

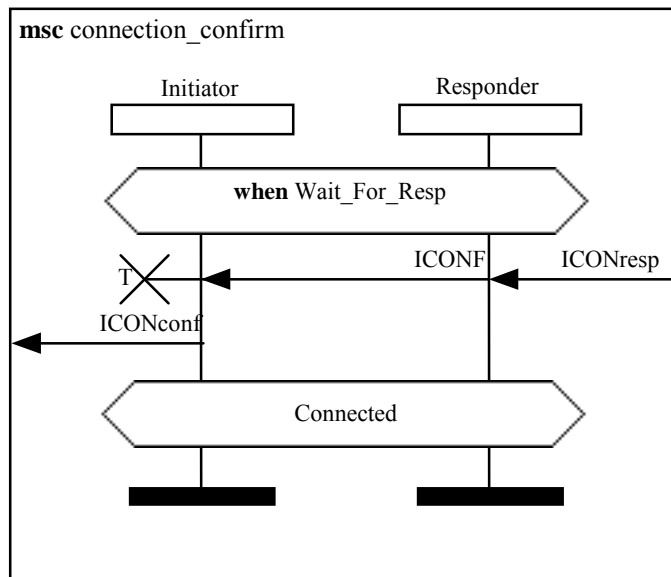
instance Initiator;
condition when Disconnected shared all;

```

```

in ICONreq from env;
starttimer T;
out ICON to Responder;
condition Wait_For_Resp shared all;
endinstance;
instance Responder;
condition when Disconnected shared all;
in ICON from Initiator;
out ICONind to env;
condition Wait_For_Resp shared all;
endinstance;
endmsc;

```



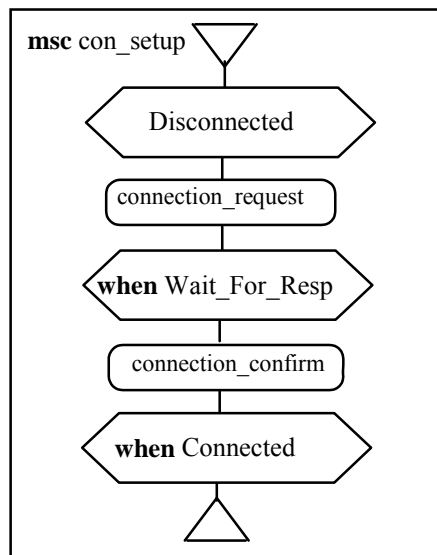
**Figure 33 – MSC connection\_confirm**

```

msc connection_confirm;
inst Initiator;
inst Responder;
gate in ICONconf from Initiator;
gate out ICONresp to Responder;

instance Initiator;
condition when Wait_For_Resp shared all;
in ICONF from Responder;
stoptimer T;
out ICONconf to env;
condition Connected shared all;
endinstance;
instance Responder;
condition when Wait_For_Resp shared all;
in ICONresp from env;
out ICONF to Initiator;
condition Connected shared all;
endinstance;
endmsc;

```



**Figure 34 – MSC con\_setup**

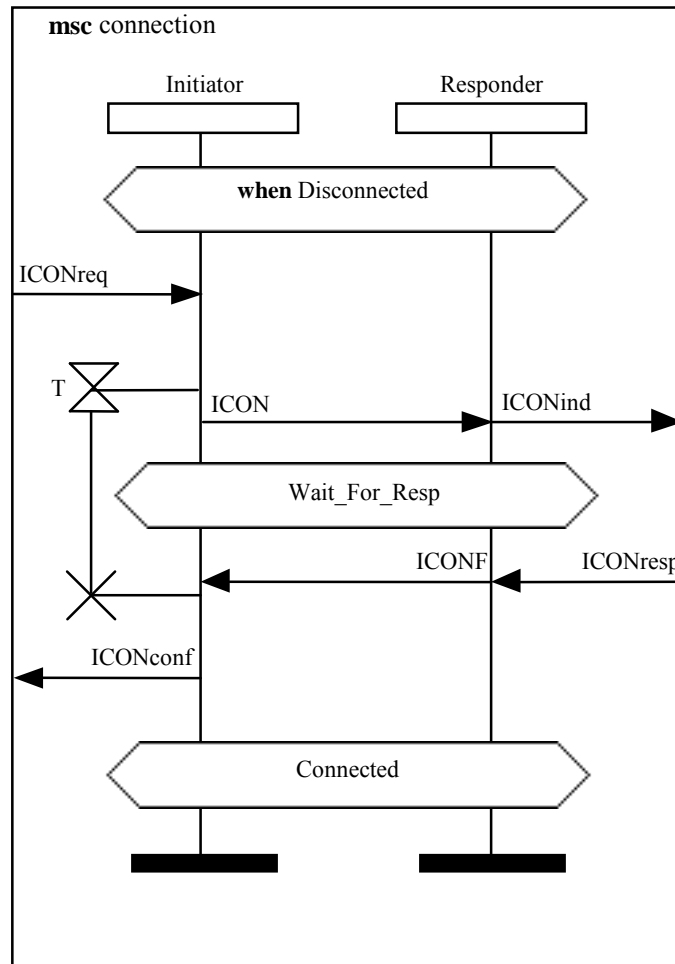
```

mhc con_setup;
  initial connect L1;
  L1: condition Disconnected connect L2;
  L2: reference connection_request connect L3;
  L3: condition when Wait_For_Resp connect L4;
  L4: reference connection_confirm connect L5;
  L5: condition when Connected connect L6;
  L6: final;
endmhc;

```

## 9.5 MSC with time supervision

The MSC 'connection' in this example contains a timer stop.



**Figure 35 – MSC connection**

```

msc connection;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;
gate out ICONresp to Responder;
gate in ICONinf from Initiator;

```

```

Initiator: instance;
condition when Disconnected shared all;
in ICONreq from env;
starttimer T;
out ICON to Responder;
condition Wait_For_Resp shared all;
in ICONf from Responder;
stoptimer T;
out ICONconf to env;
condition Connected shared all;
endinstance;

```

```

Responder: instance;
condition when Disconnected shared all;
in ICON from Initiator;
out ICONind to env;
condition Wait_For_Resp shared all;
in ICONresp from env;
out ICONf to Initiator;
condition Connected shared all;
endinstance;

```

```

endmsc;

```

## 9.6 MSC with message loss

The MSC 'failure' in this example contains a timer expiration due to a lost message.

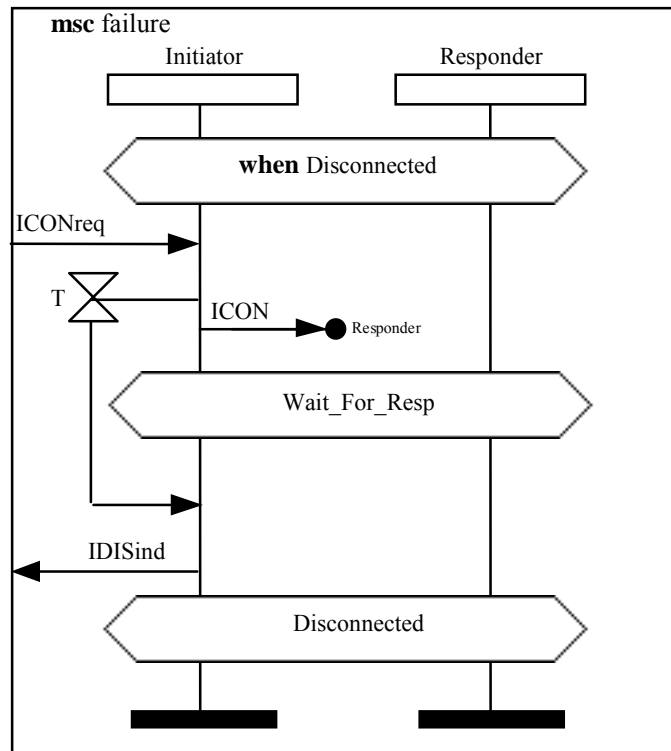


Figure 36 – MSC failure

```
mhc failure;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in IDISinf from Initiator;
```

```
Initiator: instance;
condition when Disconnected shared all;
in ICONreq from env;
starttimer T;
out ICON to lost Responder;
condition Wait_For_Resp shared all;
timeout T;
out IDISinf to env;
condition Disconnected shared all;
```

```
endinstance;
Responder: instance;
condition when Disconnected shared all;
in ICON from Initiator;
condition Wait_For_Resp shared all;
condition Disconnected shared all;
endinstance;
```

```
endmhc;
```

## 9.7 Local conditions

In this example local conditions referring to one instance are employed to indicate local states of this instance.

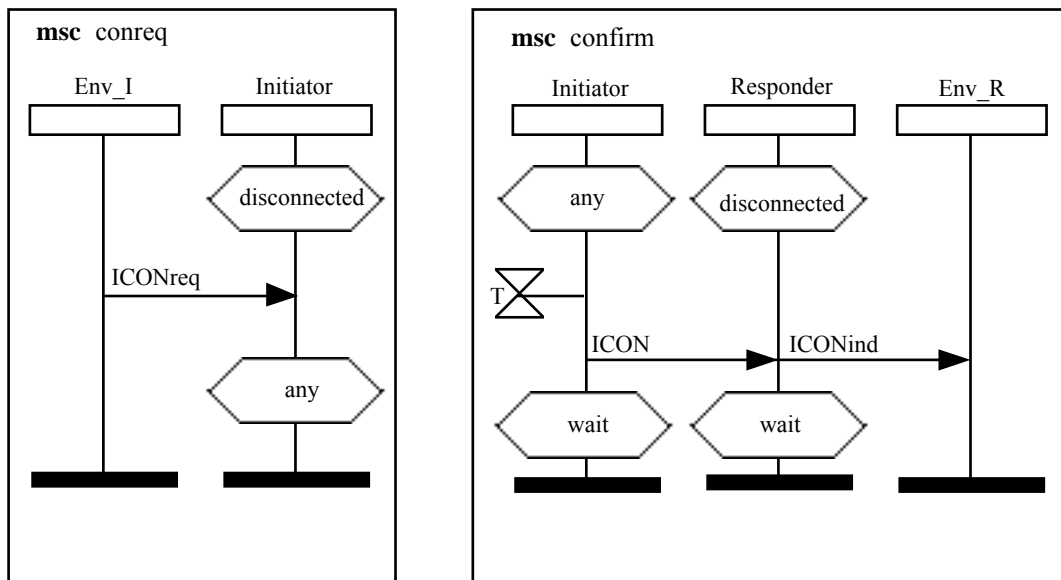


Figure 37 – Local conditions

```

msc conreq;
inst Env_I;
inst Initiator;
  Env_I: instance;
    out ICONreq to Initiator;
  endinstance;
  Initiator: instance;
    condition disconnected shared;
    in ICONreq from Env_I;
    condition any shared;
  endinstance;
endmsc;

msc confirm;
inst Initiator;
inst Responder, Env_R;
  Initiator: instance;
    condition any shared;
    starttimer T;
    out ICON to Responder;
    condition wait shared;
  endinstance;
  Responder: instance;
    condition disconnected shared;
    in ICON from Initiator;
    out ICONind to Env_R;
    condition wait shared;
  endinstance;
  Env_R: instance;
    in ICONind from Responder;
  endinstance;
endmsc;

```

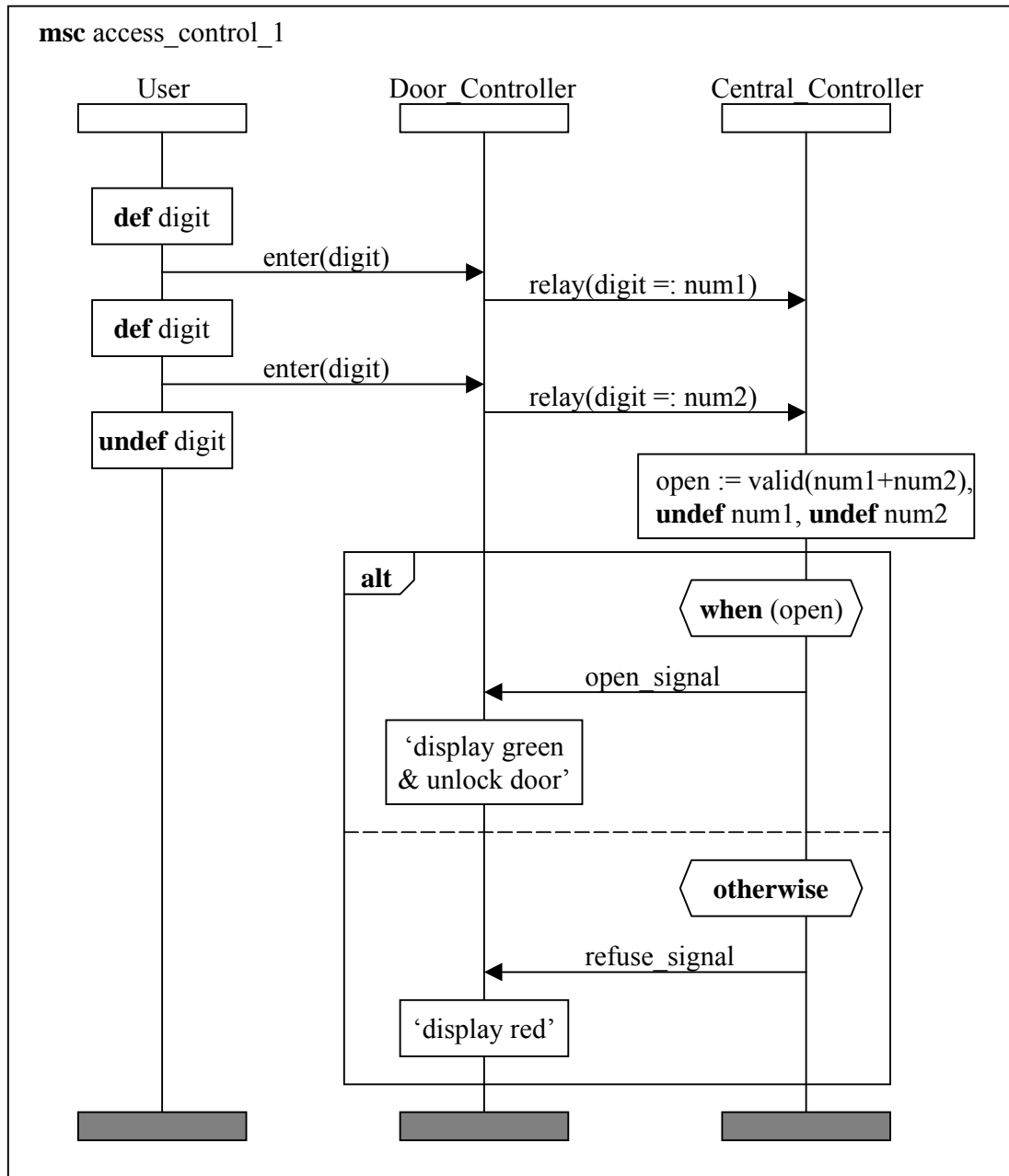
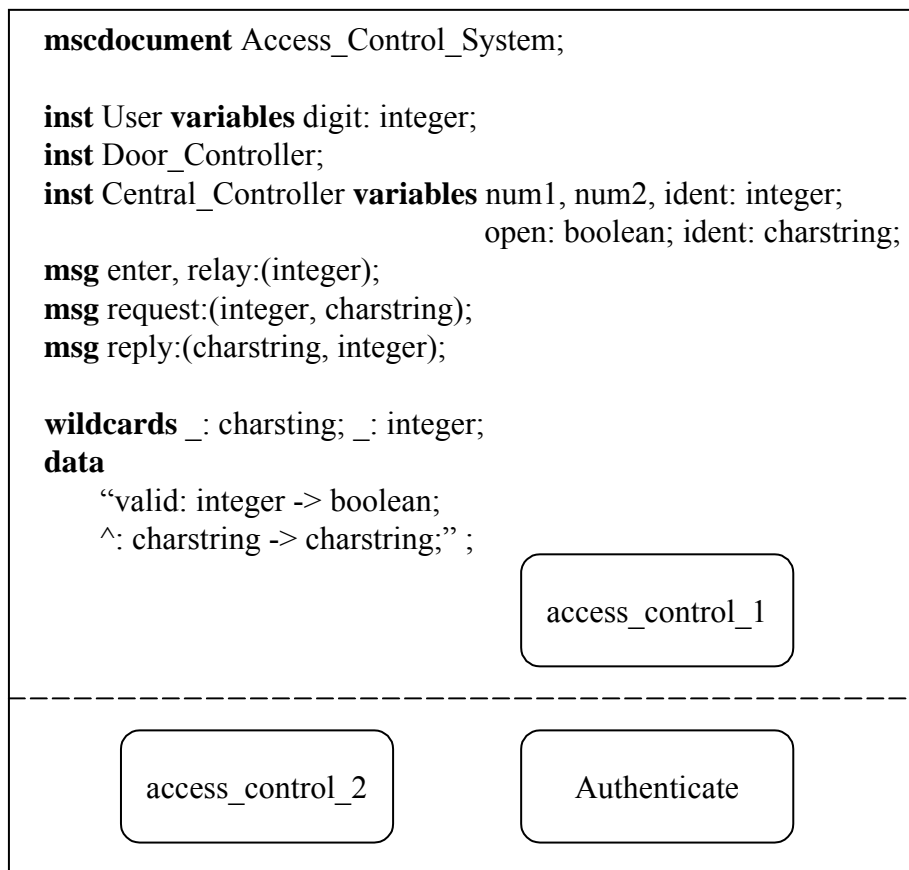


Figure 38 – Data in Action Boxes, Messages, and Conditions

In Figure 38 the instance *User* is assumed to own the variable *digit*, and the instance *Central\_Controller* owns the variables *num1* and *num2*; the declaration of these variables would occur in the enclosing MSC document which is here shown in Figure 39.



**Figure 39 – MSC document for Access\_Control\_System**

The first action box on instance *User* defines the variable *digit*, the effect of which is to assign *digit* an unspecified value; let us call this value *v1*. This value is then passed to instance *Door\_Controller* via the one parameter of the message *enter*. The parameter *digit* occurs as an expression; it is not a binding because there is no bind symbol, and the static requirements forbid it from being a pattern as *enter* is a completed message, i.e., its source and destination are instances. This value *v1* of *digit* is now available to the instance *Door\_Controller* until a new value of *digit* is received, irrespective of what happens to the value of *digit* on its owning instance.

In the first *relay* message the *v1* value of *digit* is passed as a parameter via the expression part of the binding *digit* =: *num1*. This binding causes the value *v1* to become bound to *num1* on instance *Central\_Controller*.

The second action box on *User* redefines the value of *digit* to another unspecified value, say *v2*. The semantics permit the values *v1* and *v2* to be the same. This second value is sent to *Door\_Controller* via the second *enter* message. Note that the two values of *digit* may coexist in a trace through the MSC, although only one value is known to an instance at any particular time.

In the second *relay* message the *v2* value of *digit* becomes bound to the variable *num2* on instance *Central\_Controller*. Thus the value of *num1* is *v1* and the value of *num2* is *v2*.

The final action on *User* undefines *digit*, which means that it becomes an unbound variable of *User*. Thus any subsequent reference to *digit* on this instance is illegal, which could occur if another MSC is sequenced after *access\_control\_1*. Undefining a variable explicitly removes a variable from scope until it is either bound via a binding or a def statement.

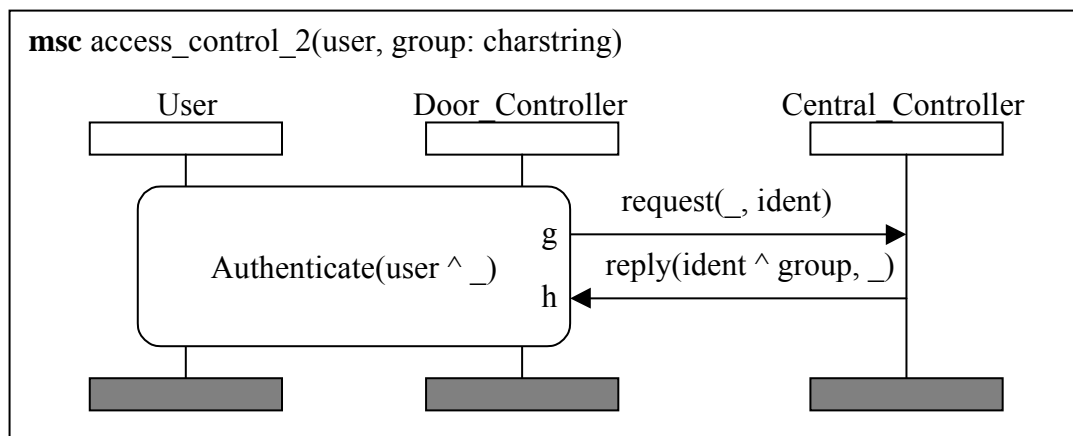
The action box on *Central\_Controller* contains three statements, which are evaluated in parallel. Any expressions are first evaluated using the old state, thus the expression *valid(num1 + num2)* is evaluated using *v1* as the value of *num1*, and *v2* as the value of *num2*. After evaluating any



expressions the state is updated according to the bindings, (def if present) and undef statements. In this case, *open* is bound to the result of the expression *valid(num1 + num2)*, where the Boolean function *valid* is declared/defined in the data section of the enclosing MSC document (Figure 39), and the variables *num1* and *num2* become unbound, i.e., out of scope. Since the statements are evaluated in parallel, they can be given in any order without altering the meaning. So the undef statements could have been given before the statement binding *open*.

The value of *open* is used as a guard in the in-line alternative expression. The guard satisfies the static requirements, in that the guard variables (here just *open*) are owned by the active instance which is *Central\_Controller*. The alternative selected depends upon the value of *open*; if it is true then the first alternative is chosen else the second alternative is chosen.

Notice that the action boxes appearing in the inline expression contain informal text, as indicated by the quote characters enclosing the text.



**Figure 40 – Static Data & Incomplete Message Data**

Figure 40 shows MSC *access\_control\_2* that has two static formal parameters, *user* and *group*, each of type *charstring*. If *access\_control\_2* is referenced in another MSC then two actual parameters must be given for these formal parameters. The MSC reference *Authenticate* has one actual parameter, the expression *user ^ \_*, where "^" represents string concatenation (defined/declared in the data section of the enclosing MSC document, Figure 39), and "\_" represents a *charstring* wildcard (declared in the enclosing MSC document, Figure 39). The MSC *access\_control\_2* also uses a dynamic variable *ident*, that is owned by the instance *Central\_Controller*.

The actual parameter expression given to *Authenticate* cannot contain dynamic variables, but it can contain wildcards and static variables, as it does here. The expression *user ^ \_* denotes any string that starts with the value given to the variable *user*.

The message *request* is an incomplete message that has two parameters. It is incomplete because it originates from the gate *g*. The static requirements for an incomplete message terminating at an instance tell us that the parameters must consist only of patterns. Thus the first parameter "\_" is a wildcard pattern which does not result in any binding, and the second is the pattern *ident* which will result in *ident* being dynamically bound to whatever value was sent from the MSC *Authenticate*. Notice, only dynamic variables owned by the instance *Central\_Controller* or wildcards can appear as patterns in messages received by this instance.

The message *reply* is also an incomplete message that has two parameters, but this time it is incomplete because it terminates at a gate (here *h*). The static requirements tell us that the parameters must consist only of expressions, so that the first parameter is the expression *ident ^ group*, and the second is the wildcard "\_". Notice parameter expressions can contain static variables (here *ident*), dynamic variables (here *group*) and wildcards.

The value given to the expression *ident* ^ *group* will be a string, the initial part consisting of whatever was sent in the second parameter of the message *request*, and the second part consisting of whatever value is supplied for the static variable *group* in a reference to the MSC *access\_control\_2*. The second parameter of *reply* can be any string. The values of these two parameters are sent with the *reply* message and will be dynamically bound to the patterns found on the corresponding receive *reply* parameter list found inside the MSC *Authenticate*.

## 11 Time

The examples in this section show main features of the time concepts in MSCs. The examples are taken from a performance test specification for a TINA access session server. The MSCs include a test component (*TC*) and a system under test (*SUT*). The test covers three steps: getting named access to an access session server, setting the user context for an user and starting a selected service for this user. These steps are defined as utility MSCs.

The defining MSCs cover the behavioural specification including time constraints (MSC *Black\_Box\_Behavior*), the measurement specification (MSC *Black\_Box\_Measurement\_Scenario*), and traces of test executions (MSC *Black\_Box\_Test\_Execution*).

The measurements taken by the test component make use of time variables *rel1*, ..., *abs2*, which are declared as dynamic variables of type Time of instance *TC*.

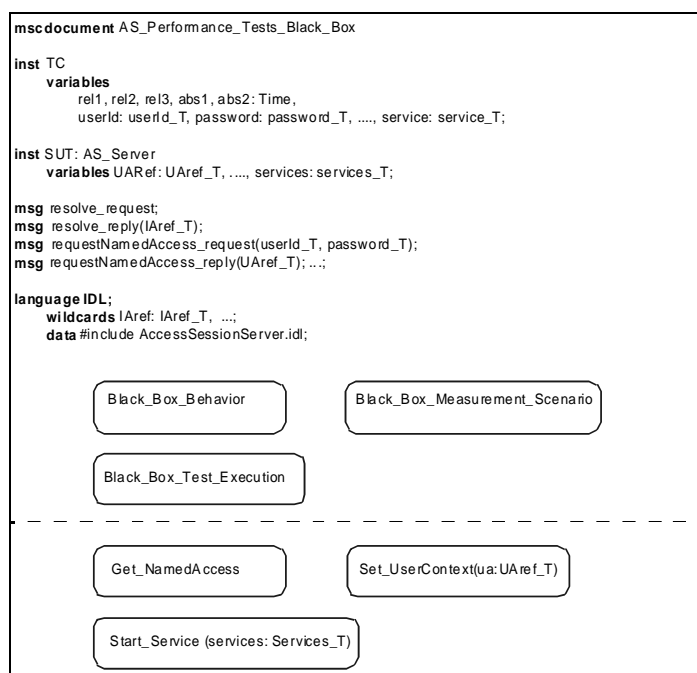
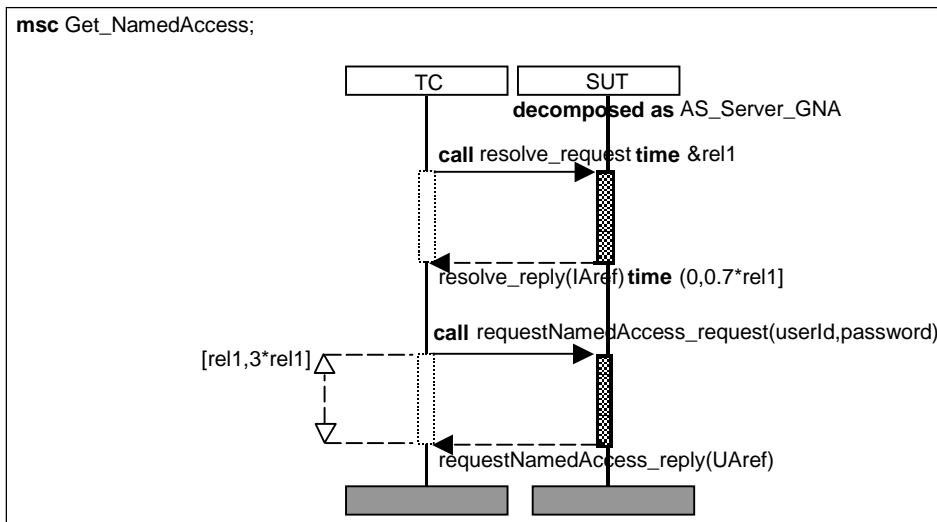


Figure 41 – MSC document and time variable declaration

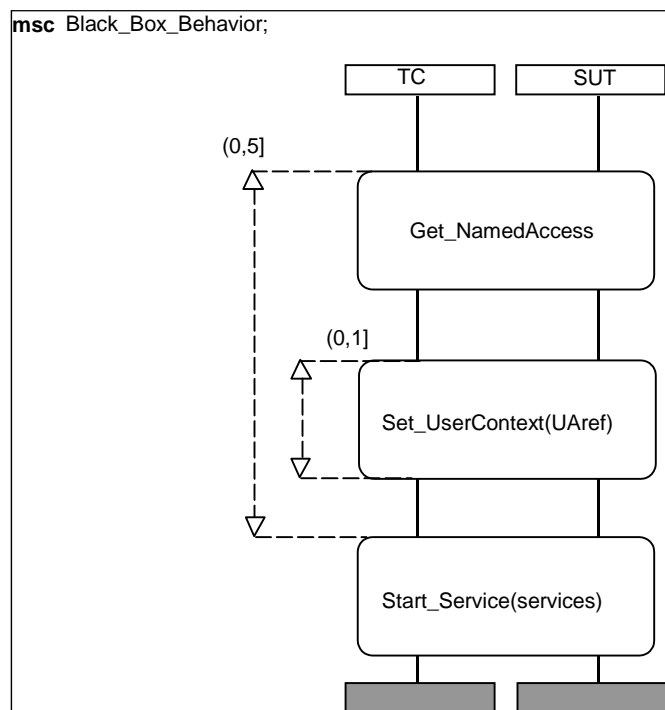


**Figure 42 – Time Constraints and Measurements**

The utility MSC shown in Figure 42 uses a relative time measurement to observe the message duration of the `resolve_request` call. The time variable `rel1` will contain the time it took from the output of the call by `TC` till the input of the call by `SUT`. The variable `rel1` is bound when the input of the call occurs.

The measurement on the duration of the call is subsequently used to restrict the message duration for `resolve_reply`. The relative time constraint  $(0,0.7*rel1]$  allows the message to take at most 70 percent of the time it took to issue the call `resolve_request` from `TC` to `SUT`.

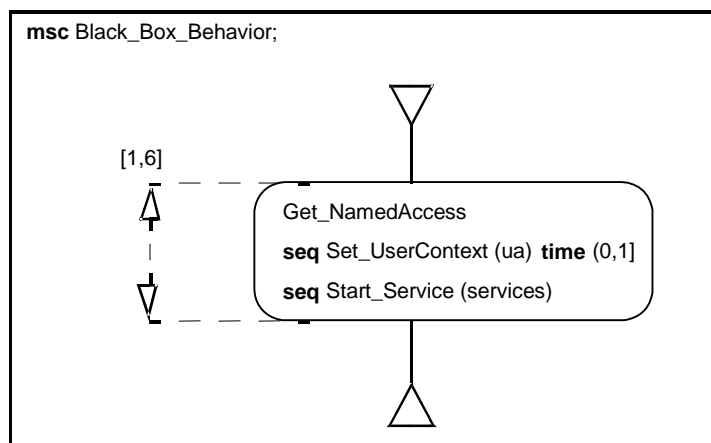
In addition, the measurement on the duration of the call is used to constrain the execution of the instance `TC`: the relative time constraint  $(rel1,3*rel1]$  requires that after the output of the `requestNamedAccess` call it takes at least `rel1` and at most  $3*rel1$  to get the reply.



**Figure 43 – Time Constraints for MSC References**

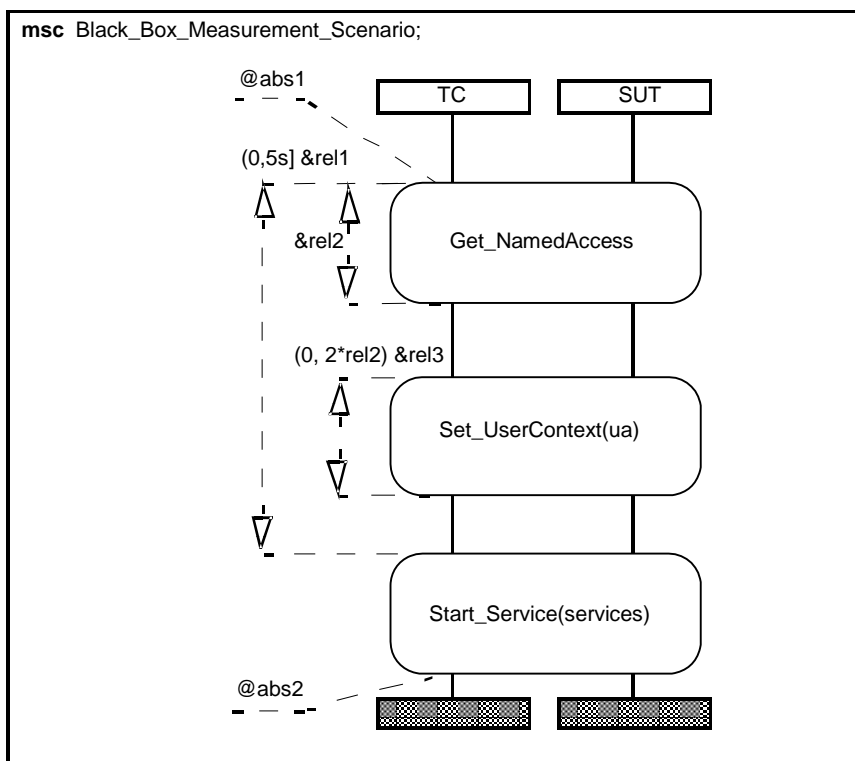
Figure 43 exemplifies the use of time constraints for MSC references: the relative time constraint  $(0,1]$  specifies that it takes at most  $1s$  to have the MSC *Set\_UserContext* executed, i.e., from the start event to the final event of *Set\_UserContext* at most  $1s$  will pass.

The relative time constraint  $(0,5]$  relates the start of *Get\_NamedAccess* to the start of *Start\_Service* and specifies that it takes at most  $5$  time units to be able to start a service from the time point, when the procedure to get named access has been started. This constraint impacts also the execution of *Set\_UserContext*.



**Figure 44 – Time Constraints for HMSC**

Figure 44 shows the use of time constraints for HMSC. It is semantically similar to Figure 43 except that in this figure the relative time constraint  $[1,6]$  is from start of *Get\_NamedAccess* to the end of *Start\_Service*, i.e., starting a service takes at least  $1$  and at most  $6$  time units.



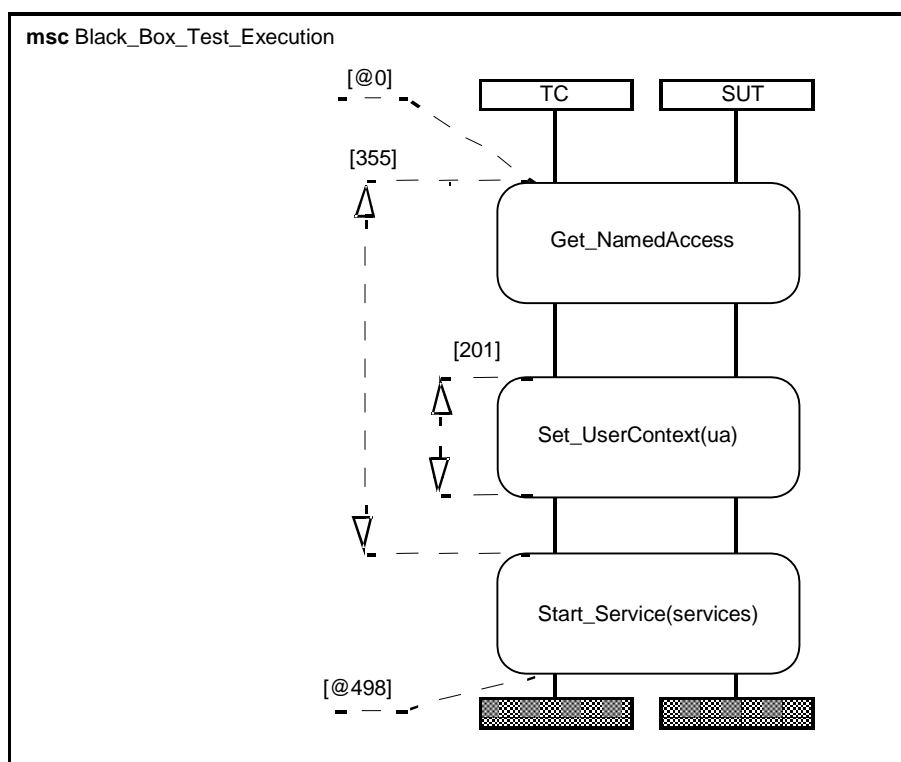
**Figure 45 – Observing Time by Measurements**

The MSC *Black\_Box\_Measurement\_Scenario* in Figure 45 uses absolute measurements (indicated by @) and relative measurements (indicated by &). Absolute measurements observe the value of the global clock.

The absolute measurements *@abs1* and *@abs2* give the absolute start and end of the test described as a sequence of *Get\_NamedAccess*, *Set\_UserContext* and *Start\_Service*. In addition, relative measurements are used to measure the time distance between pairs of events.

For example, *&rel1* measures the distance between the start event of *Get\_NamedAccess* and the start event of *Start\_Service*. This measurement is combined with a constraint  $(0,5]$  meaning that the duration between start of *Get\_NamedAccess* and start of *Start\_Service* is constrained and the time it takes really (within the given bounds of the constraint) is observed by means of the relative measurement.

$(0, 2*rel2)$  *&rel3* is also a relative time constraint combined with a relative measurement. This constraint refers to a measurement on the duration between start of *Get\_NamedAccess* and its end taken before.



**Figure 46 – Singular Time used for System Traces**

Figure 46 shows the representation of system traces (e.g., taken from system executions, simulation or tests) using absolute and relative time point. Absolute time points are indicated by @.  $[0]$  represents that *Get\_NamedAccess* was started at global clock time 0 (i.e., the global clock has been reset). The execution of the sequence *Get\_NamedAccess*, *Set\_UserContext* and *Start\_Service* took 498 time units. The execution of *Set\_UserContext* took 201 time units. The start event of *Start\_Service* occurred 355 time units after the start event of *Get\_NamedAccess*.

## 12 Creating and terminating processes

This example shows the dynamic creation of the instance 'subscriber' due to a connection request and corresponding termination due to a disconnection request.

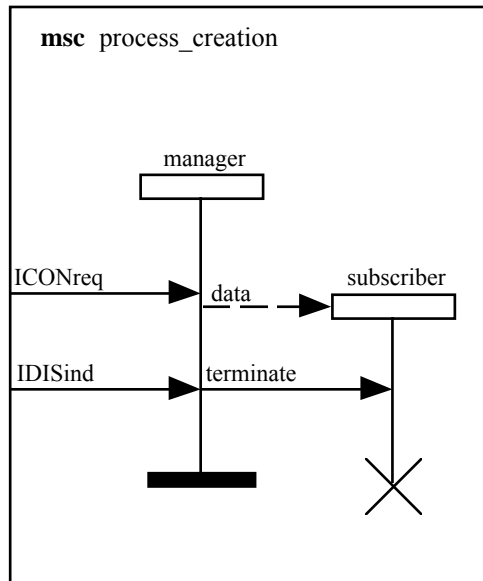


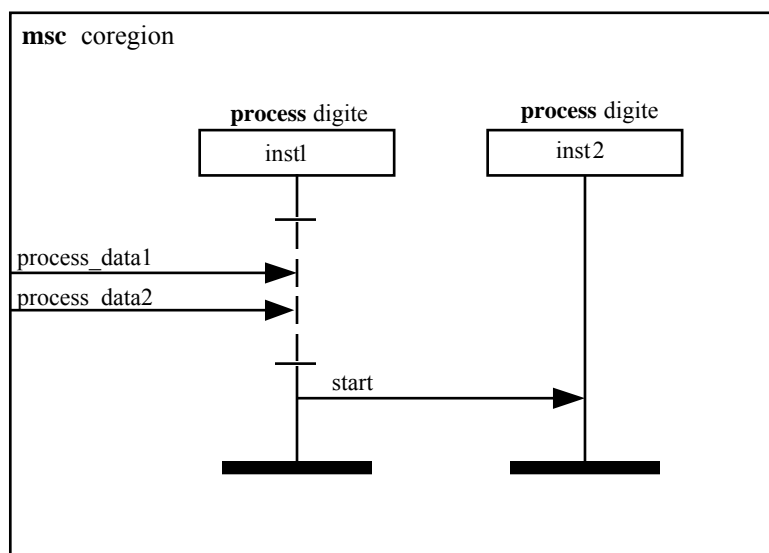
Figure 47 – MSC process\_creation

```
msc process_creation;
inst manager;
inst subscriber;
gate out ICONreq to manager;
gate out IDISind to manager;

    manager: instance;
        in ICONreq from env;
        create subscriber(data);
        in IDISind from env;
        out terminate to subscriber;
    endinstance;
    subscriber: instance;
        in terminate from manager;
        stop;
    endinstance;
endmsc;
```

## 13 Coregion

This example shows a concurrent region which shall indicate that the consumption of 'process\_data1' and the consumption of 'process\_data2' are not ordered in time, i.e., 'process\_data1' may be consumed before 'process\_data2' or the other way round.



**Figure 48 – MSC coregion**

```

msc coregion;
inst inst1;
inst inst2;
gate out process_data1 to inst1;
gate out process_data2 to inst1;

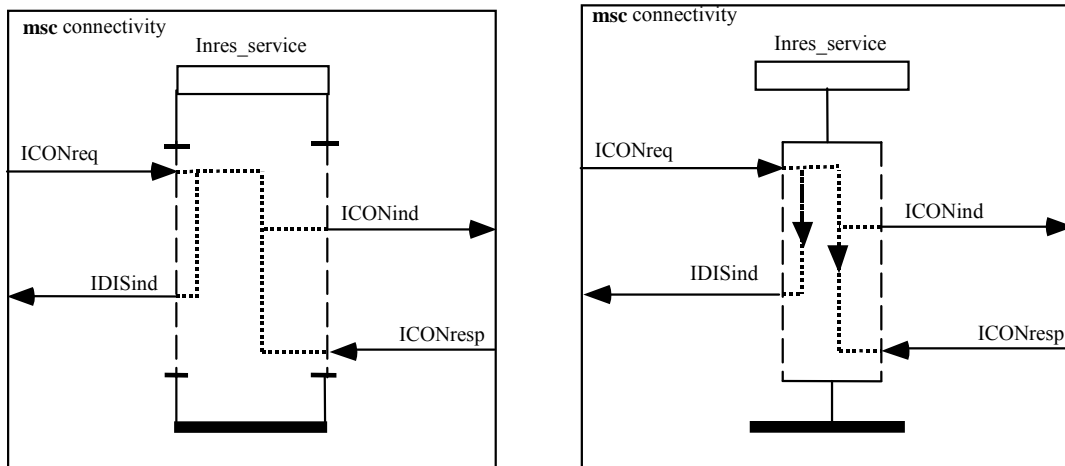
inst1: instance process digite;
concurrent;
in process_data1 from env;
in process_data2 from env;
endconcurrent;
out start to inst2;
endinstance;
inst2: instance process digite;
in start from inst1;
endinstance;
endmsc;
  
```

## 14 General ordering

### 14.1 Generalized ordering within a coregion

This example shows a generalized ordering within a coregion by means of 'connections', i.e., the ordering is described by means of the connections relating the events within the coregion. Within the MSC 'connectivity' the following ordering is defined:  $ICONreq < ICONind < ICONresp$ ,  $ICONreq < IDIsind$ .

It shows the situation where  $IDIsind$  is unordered with respect to  $ICONind$  and  $ICONresp$ .



**Figure 49 – MSC connectivity in two graphical variants**

```

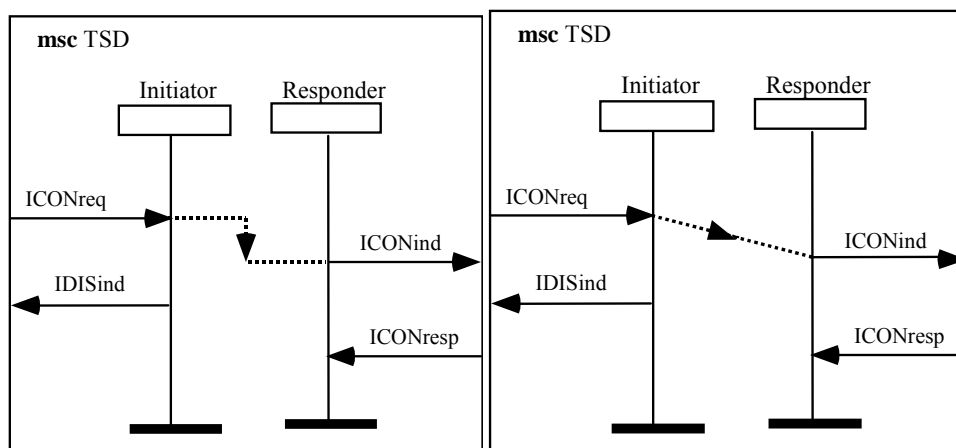
mhc connectivity;
inst Inres_service;
gate out ICONreq to Inres_service;
gate in ICONind from Inres_service;
gate in IDISind from Inres_service;
gate out ICONresp to Inres_service;

Inres_service: instance;
  concurrent;
    in ICONreq from env before Label1, Label2;
    label Label1; out ICONind to env before Label3;
    label Label2; out IDISind to env;
    label Label3; in ICONresp from env;
  endconcurrent;
endinstance;
endmhc;

```

## 14.2 Generalized ordering between different instances

This example shows the use of synchronization constructs taken over from Time sequence diagrams in order to describe a generalized ordering between different instances. The line (bended or with downward slope) between the message input 'ICONreq' and the message output 'ICONind' denotes the ordering  $ICONreq < ICONind$ .



**Figure 50 – MSC TSD in two graphical variants**

```

mhc TSD;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;

```



```

gate in IDISind from Initiator;
gate out ICONresp to Responder;

instance Initiator;
  in ICONreq from env before Label1;
  out IDISind to env;
endinstance;
instance Responder;
  label Label1; out ICONind to env;
  in ICONresp from env;
endinstance;
endmsc;

```

## 15 Inline expressions

### 15.1 Inline expression with alternative composition

In this example the successful connection case is combined with the failure case within one MSC by means of the alternative inline expression (MSC 'alternative'). Within MSC 'exception' the same situation is described by means of an equivalent exception inline expression.

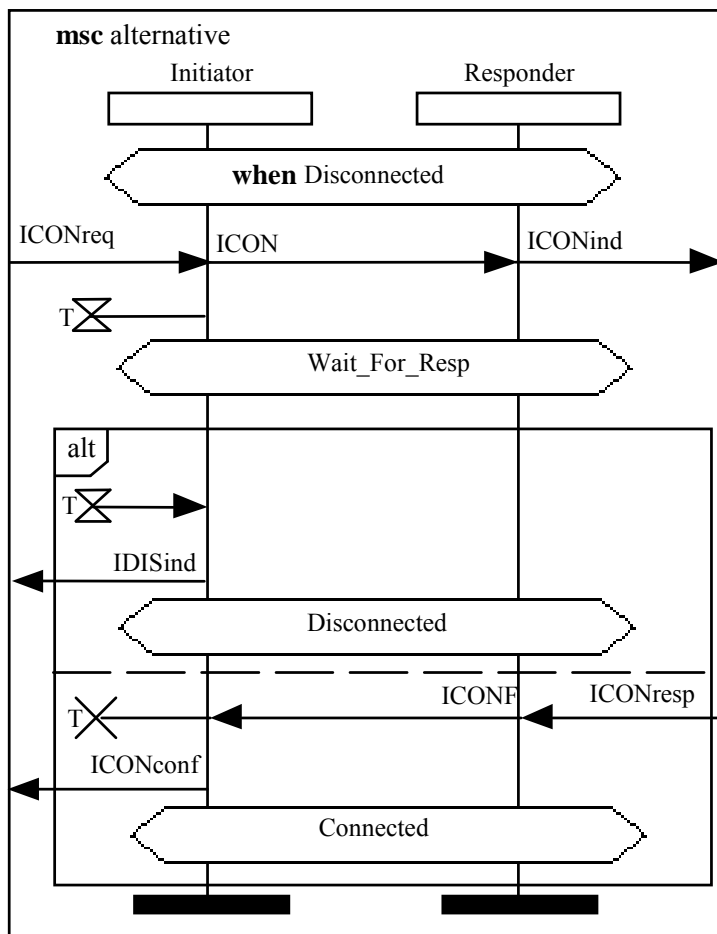


Figure 51 – Inline alternative expression

```

msc alternative;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;
gate in IDISind from Initiator;
gate out ICONresp to Responder;
gate in ICONconf from Initiator;

```

Initiator:           instance;

```

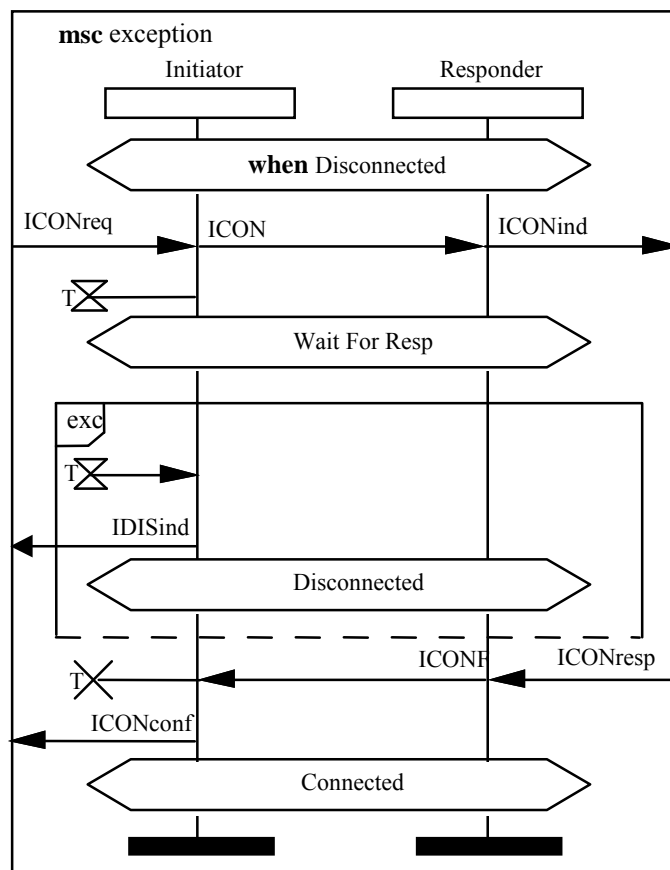
Responder:      instance;
all:          condition when Disconnected;
Initiator:      in ICONreq from env;
                out ICON to Responder;
                starttimer T;

Responder:      in ICON from Initiator;
                out ICONind to env;
all:          condition Wait_for_Resp;
                alt begin;
Initiator:      timeout T;
                out IDISind to env;
all:          condition Disconnected;
                alt;
Responder:      in ICONresp from env;
                out ICONF to Initiator;
Initiator:      in ICONF from Responder;
                stoptimer T;
                out ICONconf to env;
all:          condition Connected;
                alt end;
Initiator:      endinstance;
Responder:      endinstance;
endmsc;

```

The **exc** operator is the same as an alternative where the second operand is the entire rest of the MSC as illustrated by the following example:

MSC 'alternative' means exactly the same as MSC 'exception':



**Figure 52 – Inline exception expression**

```

msc exception;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;
gate in IDISind from Initiator;

```

```

gate out ICONresp to Responder;
gate in ICONconf from Initiator;

Initiator:      instance;
Responder:     instance;
all:         condition when Disconnected;
Initiator:     in ICONreq from env;
               out ICON to Responder;
               starttimer T;

Responder:     in ICON from Initiator;
               out ICONind to env;
all:         condition Wait_for_Resp;
               exc begin;
Initiator:     timeout T;
               out IDISind to env;
all:         condition Disconnected;
               exc end;

Responder:     in ICONresp from env;
               out ICONF to Initiator;
Initiator:     in ICONF from Responder;
               stoptimer T;
               out ICONconf to env;
all:         condition Connected;
Initiator:     endinstance;
Responder:     endinstance;

endmsc;

```

## 15.2 Inline expression with gates

This example describes the scenario of 15.1 in a slightly modified form: The message 'ICONF' from 'Responder' is connected via gates with the alternative inline expression attached to 'Initiator'. The message 'ICONF' from 'Responder' is transferred via gate g1 (on both alternatives) to 'Initiator'. It describes the situation where 'Initiator' is waiting for an answer from 'Responder'. Two cases are combined in MSC 'very\_advanced': a) the 'Responder' is not answering in time: the message input 'ICONF' is discarded after timeout and disconnection of 'Initiator', b) the 'Responder' is answering in time which leads to a successful connection.

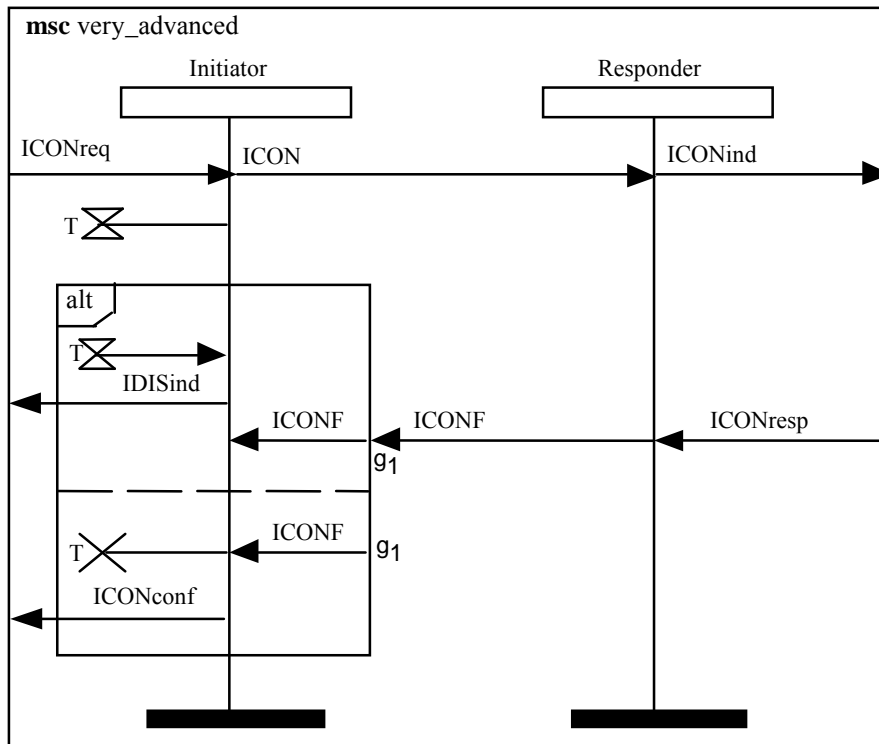


Figure 53 – Inline expression with gates

```

msc very_advanced;
inst Initiator;
inst Responder;

gate out ICONreq to Initiator;
gate in ICONind from Responder;
gate in IDISind from Initiator;
gate out ICONresp to Responder;
gate in ICONconf from Initiator;
Initiator: instance;
Responder: instance;
Initiator: in ICONreq from env;
out ICON to Responder;
starttimer T;

Responder: in ICON from Initiator;
out ICONind to env;
in ICONresp from env;
out ICONF to inline altref via g1;

Initiator: alt begin altref;
gate g1 out ICONF to Initiator
external in ICONF from Responder;
timeout T;
out IDISind to env;
in ICONF from env via g1;
alt;
gate g1 out ICONF to Initiator;
in ICONF from env via g1;
stoptimer T;
out ICONconf to env;
alt end;

Initiator: endinstance;
Responder: endinstance;
endmsc;

```

### 15.3 Inline expression with parallel composition

This example shows how a parallel inline expression describes the interleaving of a connection request initiated by 'Station\_Res', i.e., 'MDAT(CR)' followed by 'ICONind', with a disconnection request initiated by the environment, i.e., 'IDISreq' followed by 'MDAT(DR)'.

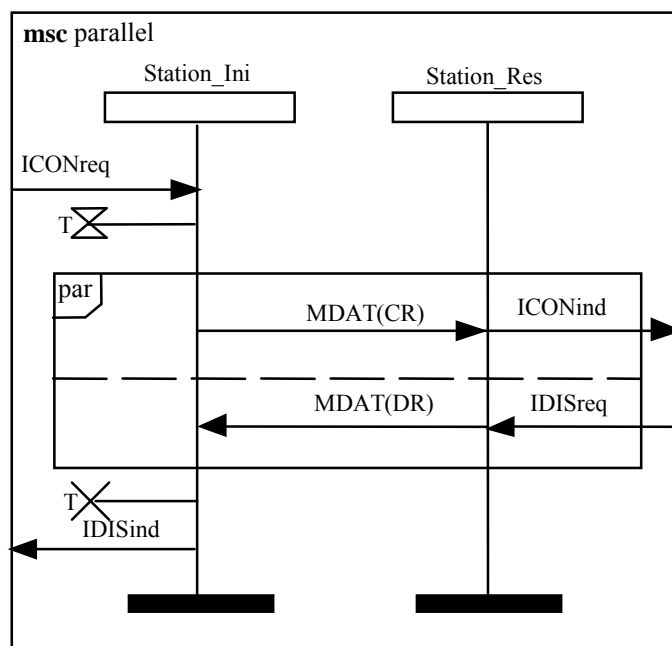


Figure 54 – Inline parallel expression

```

msc parallel;
inst Station_Ini;
inst Station_Res;
gate out ICONreq to Station_Ini;
gate in ICONind from Station_Res;
gate out IDISreq to Station_Res;
gate in IDISind from Station_Ini;

    Station_Ini: instance;
                    in ICONreq from env;
                    starttimer T;
    Station_Res:    instance;
all:
    Station_Ini:    out MDAT(CR) to Station_Res;
    Station_Res:    in MDAT(CR) from Station_Ini;
                    out ICONind to env;
                    par begin;
    Station_Res:    in IDISreq from env;
    Station_Ini:    in MDAT(DR) from Station_Res;
                    par end;
    Station_Res:    endinstance;
    Station_Ini:    stoptimer T;
                    out IDISind to env;
                    endinstance;

endmsc;

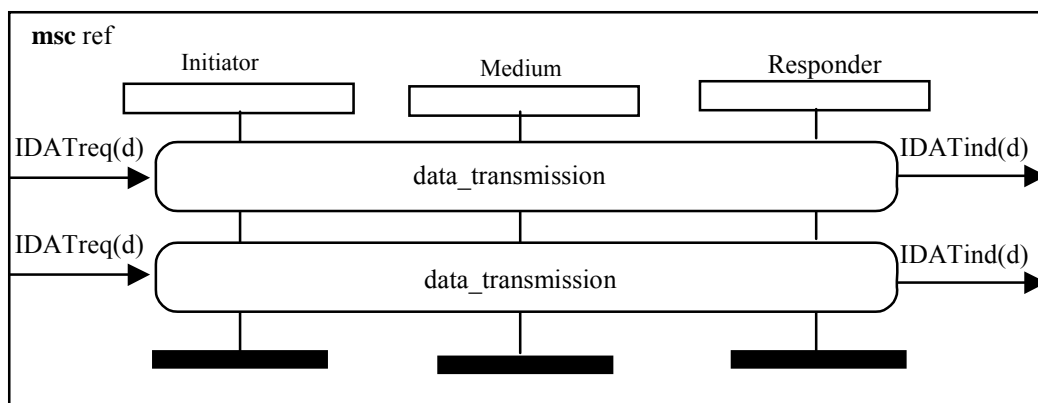
```

## 16 MSC references

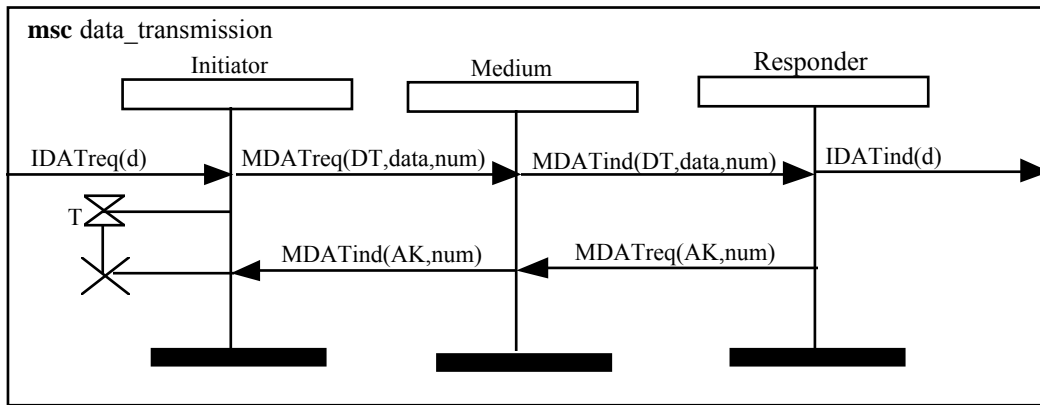
### 16.1 MSC reference

Within this example the MSC references 'data\_transmission' are employed to denote two successful data transmissions referring to the same MSC definition.

The MSC *ref* has ambiguous gates since they have implicit names, but this is all right as long as *ref* is not used as MSC reference in another diagram.



**Figure 55 – MSC including references to other MSCs**



**Figure 56 – An MSC that is referenced**

```

msc ref;
inst Initiator;
inst Medium;
inst Responder;
gate out L1 to reference data_trans1;
gate in L2 from reference data_trans1;
gate out L3 to reference data_trans2;
gate in L4 from reference data_trans2;

Initiator:      instance;
Medium:         instance;
Responder:      instance;
all:            reference data_trans1:data_transmission;
                gate in IDATreq, L1 from env;
                gate out IDATind, L2 to env
                reference data_trans2:data_transmission;
                gate in IDATreq, L3 from env
                gate out IDATind, L4 to env;

Initiator:      endinstance;
Medium:         endinstance;
Responder:      endinstance;
endmsc;

```

```

msc data_transmission;
inst Initiator;
inst Medium;
inst Responder;
gate out IDATreq(d) to Initiator;
gate in IDATind(d) from Responder;

Initiator:      instance;
Medium:         instance;
Responder:      instance;
Initiator:      in IDATreq(d) from env;
                out MDATreq(DT, data, num) to Medium;
                starttimer T;

Medium:         in MDATreq(DT, data, num) from Initiator;
                out MDATind(DT, data, num) to Responder;

Responder:      in MDATind(DT, data, num) from Medium;
                out IDATind(d) to env;
                out MDATreq(AK, num) to Medium;

Medium:         in MDATreq(AK, num) from Responder;
                out MDATind(AK, num) to Initiator;

Initiator:      in MDATind(AK, num) from Medium;
                stoptimer T;

Initiator:      endinstance;
Medium:         endinstance;
Responder:      endinstance;
endmsc;

```

## 16.2 MSC reference with gate

This example shows an MSC reference connected with the exterior by a gate. The referenced MSCs 'message\_lost' and 'time\_out' are provided in example Figure 61.

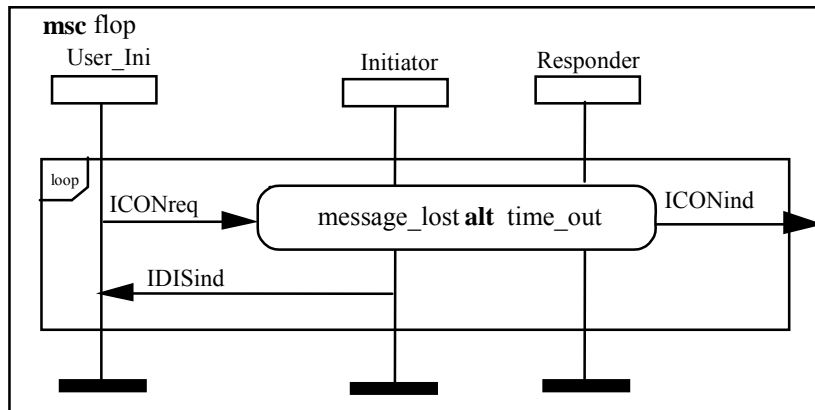


Figure 57 – Referring an MSC with message gates

```

mfc flop;
inst User_Ini;
inst Initiator;
inst Responder;
gate in ICONind from failed;

    User_Ini:      instance;
    Initiator:     instance;
    Responder:     instance;
    all:           loop begin;
        User_Ini:      out ICONreq to reference failed;
        Initiator,
        Responder:     reference failed: message_lost alt time_out;
        User_Ini:      gate in ICONreq from User_Ini;
        Responder:     gate out ICONind to env;

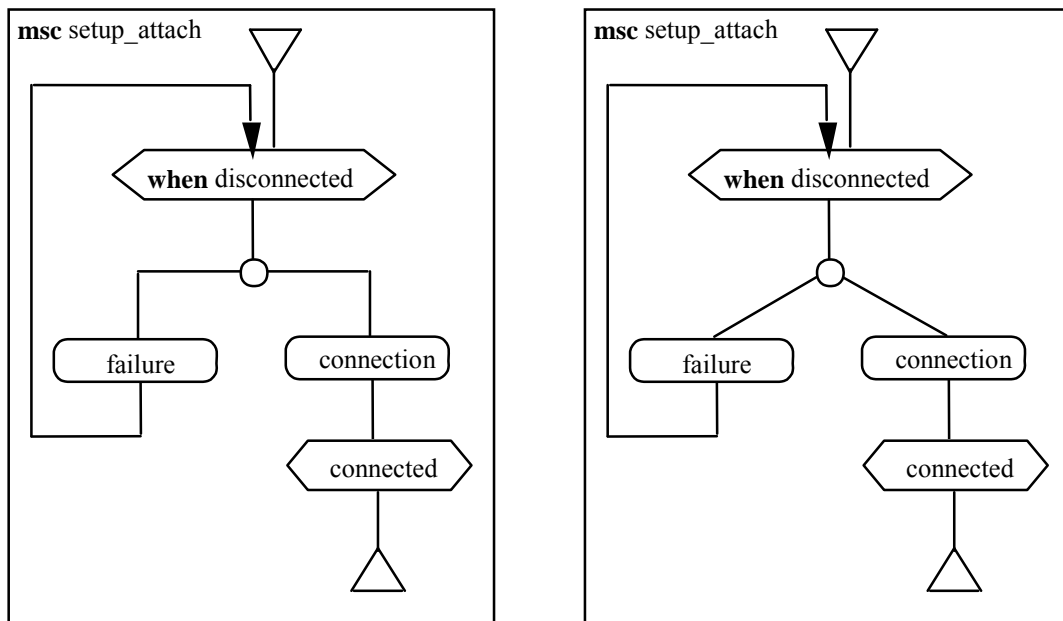
        Initiator:     out IDISind to User_Ini;
        User_Ini:      in IDISind from Initiator;

    all:           loop end;
    User_Ini:      endinstance;
    Initiator:     endinstance;
    Responder:     endinstance;
endmfc;
  
```

## 17 High-level MSC (HMSC)

### 17.1 High-level MSC with free loop

MSC 'setup\_attach' in this example shows the modelling of the connection setup by means of a free loop.



**Figure 58 – HMSC setup\_attach (two different graphical variants)**

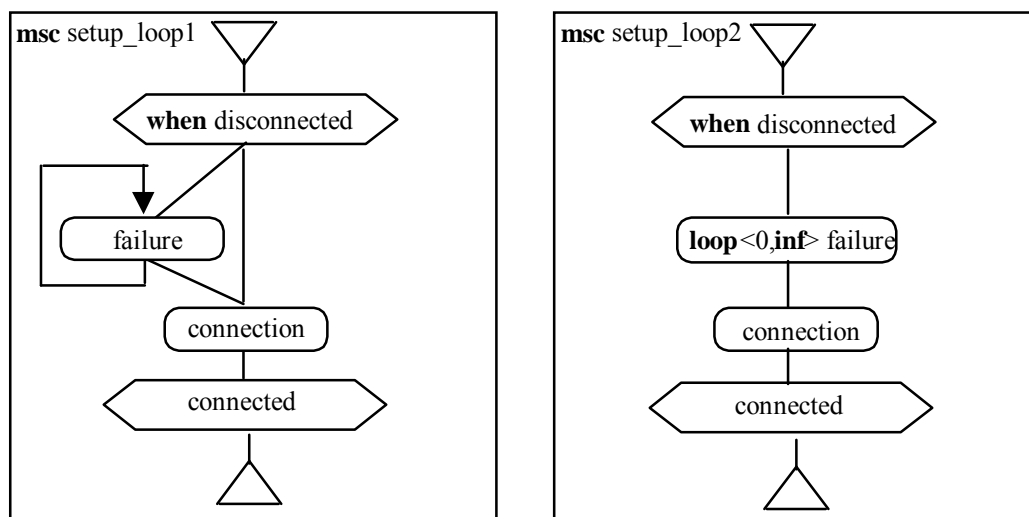
```

mhc setup_attach;
  initial connect L1;
  L1: condition when disconnected connect L2;
  L2: connect L3, L4;
  L3: reference failure connect L1;
  L4: reference connection connect L5;
  L5: condition connected connect L6;
  L6: final;
endmhc;

```

## 17.2 High-level MSC with loop

This example shows the modelling of the connection setup by means of a loop attached to the MSC reference 'failure'.



**Figure 59 – setup\_loop in two different variants**

```

mhc setup_loop1;
  initial connect L1;
  L1: condition when disconnected connect L2, L3;
  L2: reference failure connect L2, L3;
  L3: reference connection connect L4;
  L4: condition connected connect L5;

```



```

L5: final;
endmsc;
msc setup_loop2;
  initial connect L1;
  L1: condition when disconnected connect L2;
  L2: reference loop <0, inf> failure connect L3;
  L3: reference connection connect L4;
  L4: condition connected connect L5;
  L5: final;
endmsc;

```

### 17.3 High-level MSC with alternative composition

MSC 'alternative' in this example shows two methods of expressing an alternative construct. The first with correct parenthesis in which the branching has a corresponding join construct, and the second as an in-line expression, in which each alternative has its own start and end node.

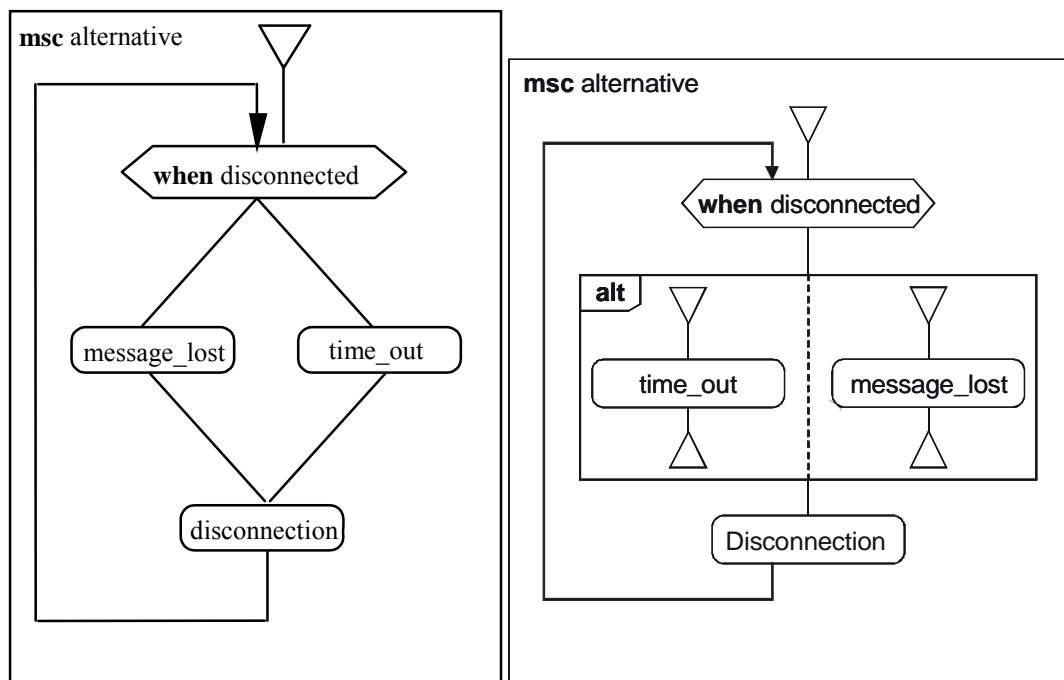
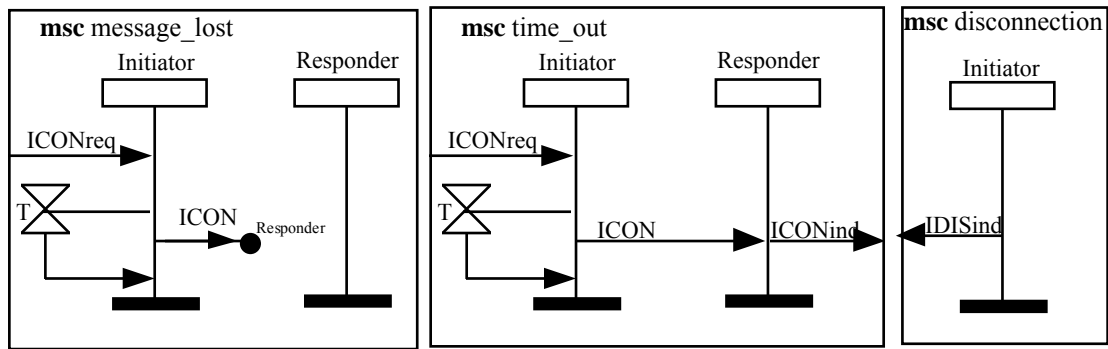


Figure 60 – HMSC with different alternative expressions

```

msc alternative1;
  initial connect L1;
  L1: condition when disconnected connect L2, L3;
  L2: reference message_lost connect L4;
  L3: reference time_out connect L4;
  L4: reference disconnection connect L1;
endmsc;
msc alternative2;
  initial connect L1;
  L1: condition when disconnected connect L2;
  L2: alt begin;
    initial connect L3;
    L3: reference time_out connect L4;
    L4: final;
  alt;
    initial connect L5;
    L5: reference message_lost connect L6;
    L6: final;
  alt end connect L7;
  L7: reference disconnection connect L1;
endmsc;

```



**Figure 61 – Simple MSCs referenced from the HMSC in Figure 60**

```

msc message_lost;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;

Initiator: instance;
in ICONreq from env;
starttimer T;
out ICON to lost Responder;
timeout T;
endinstance;
Responder: instance;
in ICON from Initiator;
endinstance;
endmsc;

```

```

msc time_out;
inst Initiator;
inst Responder;
gate out ICONreq to Initiator;
gate in ICONind from Responder;

Initiator: instance;
in ICONreq from env;
starttimer T;
out ICON to Responder;
timeout T;
endinstance;
Responder: instance;
in ICON from Initiator;
out ICONind to env;
endinstance;
endmsc;

```

```

msc disconnection;
inst Initiator;
gate in IDISind from Initiator;

Initiator: instance;
out IDISind to env;
endinstance;
endmsc;

```

## 17.4 High-level MSC with parallel composition

In this example the connection request from the 'Initiator' is merged in parallel with the disconnection request from the 'Responder' by means of the parallel HMSC in-line expression.

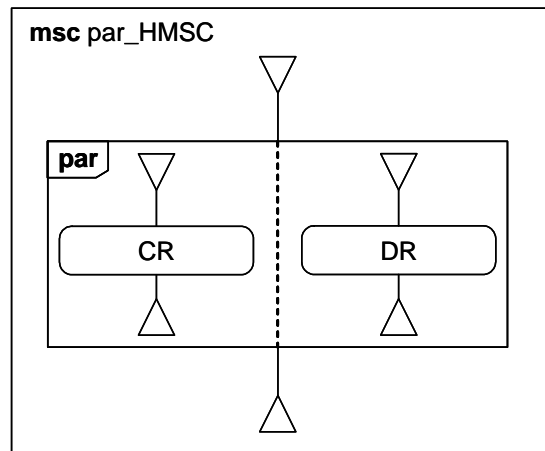


Figure 62 – HMSC with parallel expression

```

mhc par_HMSC;
  initial connect L1;
  L1: par begin;
    initial connect L2;
    L2: reference CR connect L3;
    L3: final;
  par;
    initial connect L4;
    L4: reference DR connect L5;
    L5: final;
  par end connect L6;
  L6: final;
endmhc;

```

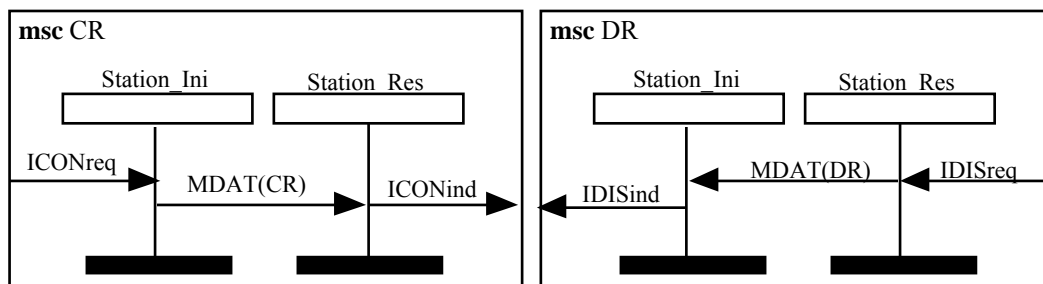


Figure 63 – The MSCs referenced from the HMSC in Figure 62

```

mhc CR;
  inst Station_Ini;
  inst Station_Res;
  gate out ICONreq to Station_Ini;
  gate in ICONind from Station_Res;

  Station_Ini: instance;
    in ICONreq from env;
    out MDAT(CR) to Station_Res;
  endinstance;
  Station_Res: instance;
    in MDAT(CR) from Station_Ini;
    out ICONind to env;
  endinstance;
endmhc;

mhc DR;
  inst Station_Ini;

```

```
inst Station_Res;  
gate out IDISreq to Station_Res;  
gate in IDISind from Station_Ini;  
  
    Station_Res:    instance;  
                    in IDISreq from env;  
                    out MDAT(DR) to Station_Ini;  
                    endinstance;  
    Station_Ini: instance;  
                in MDAT(DR) from Station_Res;  
                out IDISind to env;  
                endinstance;  
endmsc;
```

# Appendix I

## Application of MSC

(This appendix does not form an integral part of this Recommendation)

### I.1 Introduction

This appendix addresses the problem of possible applications of message sequence charts (MSCs). The main part of this Recommendation states that MSCs are meant to "describe interactions between a number of independent message passing Instances". In addition to this, MSCs are "a scenario language", "a graphical language", "a formal language", and are "widely applicable", that is not "tailored for one single application domain". These application domains are not explicitly defined in the main part of this Recommendation, which leaves ground for the following interpretation that MSCs can be used in almost any context, without restriction. This is not the case, and recent literature has shown that with their whole expressive power, several applications of MSCs were impracticable.

Among the applications of MSC, the following are frequently addressed:

- model checking,
- comparison of specifications,
- specification and implementation.

This list is not exhaustive, but has been well covered by the literature in the last decade. In particular, several publications have shown that these applications can be undecidable problems for MSCs, that is there exists no algorithm that takes as an input any MSC and that terminates with as output a correct solution. The objective of this appendix is to provide a list of known decision problems that are impracticable in general for MSCs, and a list of syntactic criteria that ensure decidability of some of the problems listed.

This appendix is organized as follows. Clause I.2 gives a more precise definition of the model checking or comparison problems that can be considered for MSCs, and of the notion of implementation of MSCs. Clause I.4 identifies syntactic subclasses of MSCs called regular MSCs, local choice MSCs and globally cooperative MSCs for which model checking, comparison, or implementation problems have a solution.

### I.2 Problems

#### I.2.1 Model checking

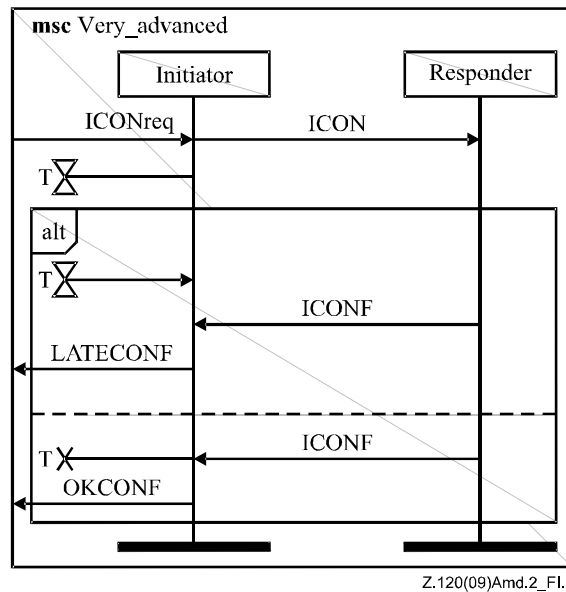
The usual definition of model checking is verifying whether a logic formula  $\phi$ , described with a specific syntax, is satisfied by a model  $M$ . This is written  $M \models \phi$ . Several popular logics exist. We can cite linear temporal logic (LTL), computational tree logic (CTL), CTL\*, alternating-time temporal logic (ATL), and the modal  $\mu$ -calculus. For an introduction to model checking and logics, interested readers may consult [b-Clarke99], and [b-Holzmann99].

Temporal logics are frequently used to ensure that modelled systems satisfy some safety or liveness properties. Logics can address properties of global states, or question the structure of the model itself, and the interpretation of a formula  $\phi$  depends on the semantics of the logic. Similarly, the usual interpretation of model checking is that the formulae address properties of runs and global states of the model.

An example of linear temporal logic (LTL) formula is:

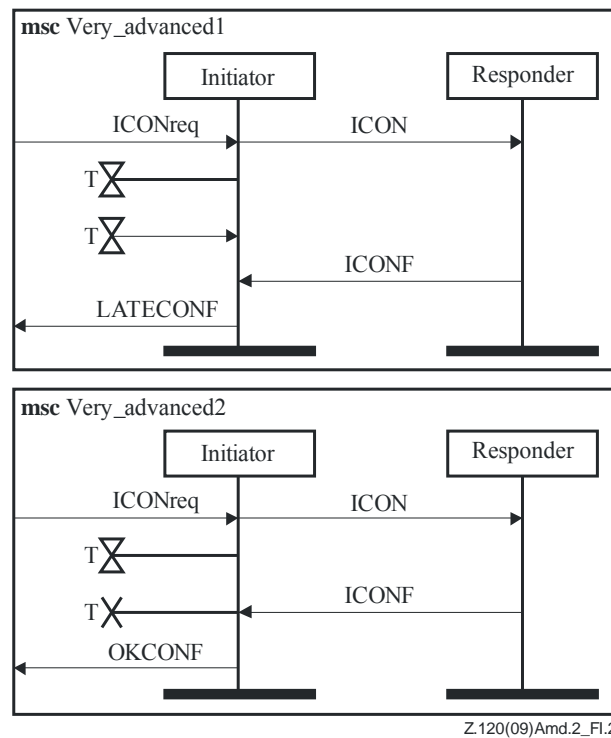
$$\phi_1 = \mathbf{G}(a \Rightarrow \mathbf{F} b)$$

With the LTL semantics, if  $a$  and  $b$  are action names, this property means that it is always true that when action  $a$  is played, then action  $b$  is eventually played. A whole description of LTL and other logics is beyond the scope of this appendix.



**Figure I.1 – An MSC description with an alternative**

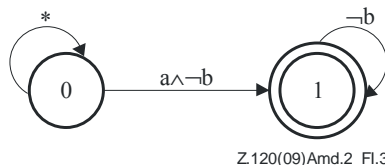
For message sequence charts, the runs of a description are defined by the semantics provided by [b-ITU-T Z.120 Annex B]. For a given MSC description  $M$ , we will denote by  $L(M)$  the set of all runs defined by  $M$ . Note also that an MSC description does not only represent a set of runs, but also a set of basic MSCs, which can be obtained by unfolding loops, replacing alternatives by a single choice, etc. Consider, for instance, the MSC description of Figure I.1. This description defines two possible MSCs that are depicted in Figure I.2.



**Figure I.2 – Basic MSC interpretation of the MSC in Figure I.1**

From now on, we will denote by  $F(M)$  the set of basic MSCs (bMSCs) described by an MSC description  $M$ .

Frequently, checking a logical formula over runs of a model  $M$  is equivalent to verifying joint properties of runs of the model and of a finite state automaton  $R_\phi$  computed from the formula. For instance, an automaton  $R_{\neg\phi_1}$  associated to the negation of formula  $\phi_1$  above is depicted in Figure I.3. This automaton describes all runs that do not satisfy  $\phi_1$ . Checking whether a MSC  $M$  satisfies  $\phi_1$  consists in verifying that the set of runs described by  $M$  and by the automaton  $R_{\neg\phi_1}$  are disjoint.



**Figure I.3 – An automaton  $R_{\neg\phi_1}$  collecting runs that satisfy  $\neg\phi_1$**

To model-check logical properties on message sequence charts specifications, tools have to provide an answer to a question of the kind:

- (Mc1)  $L(M) \subseteq R_\phi$  ?
- (Mc2)  $R_\phi \subseteq L(M)$  ?
- (Mc3)  $L(M) \cap R_\phi = \emptyset$  ?

Where  $M$  is the MSC description,  $\phi$  a logical formula,  $R_\phi$  a finite state automaton that describes sets of runs that satisfy (or do not satisfy)  $\phi$ . Here,  $Mc1$  occurs when  $R_\phi$  models all acceptable behaviours: the behaviours of the MSC specification should be contained in the behaviours of  $R_\phi$  and the user wants a positive answer.  $Mc2$  occurs when  $R_\phi$  models the bad properties of all behaviours that should not occur, and the user expects a negative answer.  $Mc3$  occurs when  $R_\phi$  models behaviours that have some undesired property, and the user expects a negative answer.

Within this setting, comparison of the runs of an MSC with a finite state automaton should be a tractable problem.

### I.2.2 Comparison of MSC descriptions

Comparison of MSC descriptions is another problem close to model checking. As there might be several ways to describe similar behaviours with message sequence charts, the questions of the equivalence of two descriptions might be interesting. For two MSC descriptions  $M1$  and  $M2$ , one may also want to verify that  $M2$  is an extension of  $M1$ , i.e., that all behaviours described by  $M1$  can be found in  $M2$ . This comparison can occur at the level of runs, or at the level of basic MSCs generated by two MSC specifications. Hence, comparing two MSC specifications  $M1$  and  $M2$  resumes to answering any of the six following questions:

- (*EquivL*)  $L(M1) = L(M2) ?$
- (*RefL*)  $L(M1) \subseteq L(M2) ?$
- (*IntL*)  $L(M1) \cap L(M2) = \emptyset ?$
- (*EquivF*)  $F(M1) = F(M2) ?$
- (*RefF*)  $F(M1) \subseteq F(M2) ?$
- (*IntF*)  $F(M1) \cap F(M2) = \emptyset ?$

### I.2.3 Specification and implementation

Message sequence charts allow for the description of interactions. It is then tempting to consider them as a specification or even as a development and programming language. However, not all MSC descriptions can be implemented. Consider, for instance, the example of Figure I.9. In this MSC, two systems are performing actions, namely *count* for instance *System1*, and *recount* for instance *System2*. Implicitly, in all bMSCs depicted by this description, the number of occurrences of actions *count* and *recount* executed by both instances should be the same. However, the two systems never communicate. Hence, without providing additional mechanisms to synchronize *System1* and *System2*, the description of Figure I.9 cannot be implemented.

If a user considers that message sequence charts present behaviours at a certain abstraction level, this kind of description is not a real problem, as using additional messages in implementations of this description is allowed. Now, if the MSC description is considered as complete, i.e., all message actions, timers and so on that will be used by any implementation appear in the description, then some MSC descriptions cannot be implemented.

A general approach to implement a MSC description is to implement the behaviour of each instance separately (for instance with SDL) [b-Khendek99]. However, it has been shown that not all MSC descriptions can be implemented this way [b-Alur05], as some additional unspecified behaviours appear in the generated implementation. When a MSC description can be implemented by separating all instances behaviours, it will be called a *realizable* MSC.

Hence, a natural question that arises for a given MSC description  $M$  is:

- (*Rez*) *is M realizable ?*

In general, there is no procedure to answer the realizability question [b-Alur05]. However, recent results [b-Genest02], [b-Genest04], and [b-Helouet00] have shown that a slight modification of the message contents can allow for the implementation of some subclasses of MSCs.



### I.3 General undecidable results

A problem whose answer can be yes or no is called *decidable* when there exists an algorithm that can compute a correct answer for any instance of this problem. If such algorithm does not exist, the problem is said to be *undecidable*.

- (Mc1)  $L(M) \subseteq R_\phi ?$
- (Mc2)  $R_\phi \subseteq L(M) ?$
- (Mc3)  $L(M) \cap R_\phi = \phi ?$
- (EquivL)  $L(M1) = L(M2) ?$
- (RefL)  $L(M1) \subseteq L(M2) ?$
- (IntL)  $L(M1) \cap L(M2) = \phi ?$
- (EquivF)  $F(M1) = F(M2) ?$
- (RefF)  $F(M1) \subseteq F(M2) ?$
- (IntF)  $F(M1) \cap F(M2) = \phi ?$
- (Rez) *is M realizable ?*

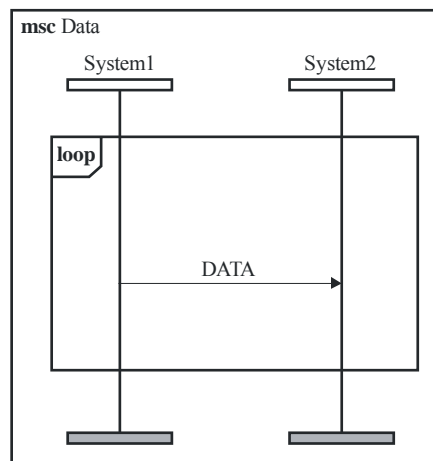
are undecidable problems. This does not mean that model checking, comparison or implementability of MSCs are always untractable problems for MSCs (i.e., they have no algorithmic solution), but rather that when considering a target application for an MSC description, users have to make sure that their specification meets some syntactical requirements. Some simple syntactic criteria allow for the characterization of several kinds of MSC descriptions (or MSC subclasses) that enable the decidability of some of the problems listed above.

### I.4 Syntactical description of MSC subclasses

Due to the undecidability results cited in clause I.3, several applications could be considered as impossible for message sequence charts. Several restrictions to the use of MSC constructs have been defined. This clause lists three of them, and for each syntactical subclass lists the possible applications.

#### I.4.1 Regular MSCs (RMSCs)

The set of runs of a regular MSC forms a regular language. This means that this set can be represented by a finite state machine, but also that usual techniques of model checking can be applied to regular MSC. An MSC description forms a regular MSC description if, in all loops that can appear in the description, all messages that are sent are acknowledged, either directly or indirectly, and if the body of the loop does not form disconnected parts of behaviours.



Z.120(09)Amd.2\_Fl.4

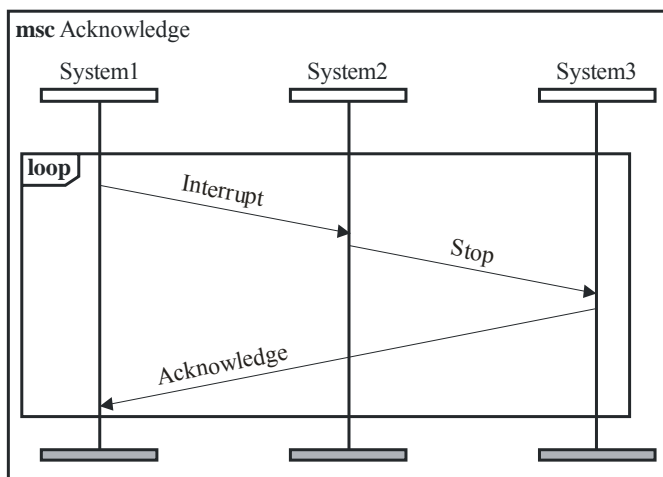
```

mhc Data;
inst System1, System2;
instance System1;
  loop begin simpleloop shared all;
    out DATA to System2;
  loop end;
endinstance;
instance System2;
  loop begin simpleloop shared all;
    in DATA from System1;
  loop end;
endinstance;
endmhc;

```

**Figure I.4 – A non-regular MSC**

Consider, for instance, the MSC description of Figure I.4. In message sequence charts, messages are considered asynchronous. This specification then describes a protocol where instance *System1* does not have to wait for an acknowledgement of *DATA* messages before sending the next message. Runs of this MSC cannot be depicted by a finite state automaton. The second condition is illustrated by the MSC *Count* of Figure I.9. In this MSC description, all MSCs in  $F(Count)$  contain the same number of occurrences of atomic actions *count* and *recount*. The runs of this MSC cannot be described with a finite state automaton. The MSC Acknowledge of Figure I.5 fulfils the conditions to be regular.



Z.120(09)Amd.2\_Fl.5

```

mhc Acknowledge;
inst System1, System2, System3;
instance System1;
  loop begin simpleloop shared all;
    out Interrupt to System2;
    in Ack from System3;
  loop end;
endinstance;
instance System2;
  loop begin simpleloop shared all;
    in Interrupt from System1;
    out Stop to System3;
  loop end;
endinstance;
instance System3;
  loop begin simpleloop shared all;
    in Stop from System2;
    out Ack to System1;
  loop end;
endinstance;
endmhc;

```

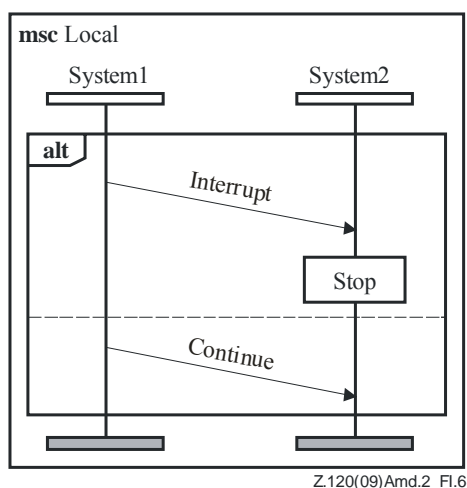
**Figure I.5 – A regular MSC**

To summarize, the following questions might be solved by appropriate algorithms for the class of regular MSCs: *Mc1*, *Mc2*, *Mc3*, *EquivL*, *RefL*, *IntL*.

#### I.4.2 Local choice MSCs (LMSCs)

An MSC description is called a local choice MSC when, for all alternatives, there is one single instance which can take the decision of how to continue interactions with other instances.

Consider the example of Figure I.6. MSC *Local* is a local choice MSC, as for both behaviours in the alternative, instance *System1* chooses how the interaction will continue, by sending a message to *System2*. MSC *Nonlocal* in Figure I.7 is not a local choice MSC, as the decision to perform one of the alternative scenarios can be taken either by *System1* or by *System3*. There is a chance that an implementation of such a scenario leads to a deadlock. A deadlock is a situation where two or more processes are waiting for each other to continue their execution. For the MSC description of Figure I.7, if the program implementing *System1* behaves as in the first part of the alternative, and the program implementing *System3* behaves as in the second part of the alternative, then a deadlock can occur.

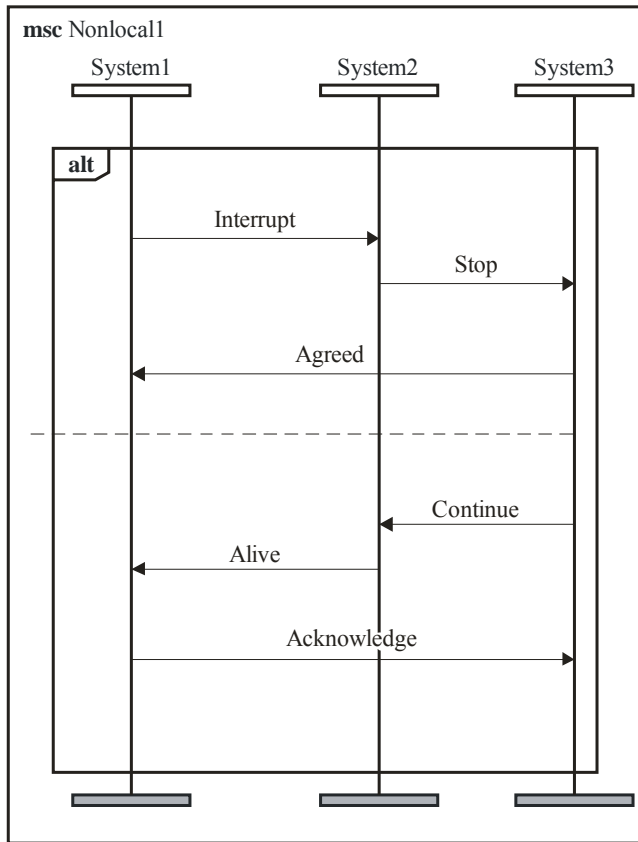


```

msc Local;
inst System1, System2;
instance System1;
  alt begin simplealt shared all
    out Interrupt to System2;
  alt;
    out Continue to System2;
  alt end;
endinstance;
instance System2;
  alt begin simplealt shared all
    in Interrupt from System1;
    action Stop;
  alt;
    in Continue from System2;
  alt end;
endinstance;
endmsc;

```

Figure I.6 – A local choice MSC



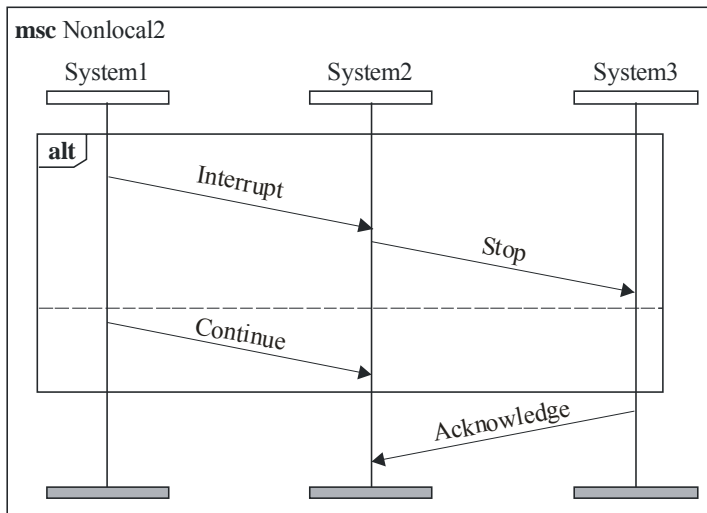
Z.120(09)Amd.2\_FI.7

```

msc Nonlocal1;
inst System1, System2, System3;
instance System1;
alt begin simplealt shared all;
  out Interrupt to System2;
  in Agreed from System3;
alt;
  in Alive from System2;
  out Acknowledge to System3;
alt end;
endinstance;
instance System2;
alt begin simplealt shared all;
  in Interrupt from System1;
  out Stop to System3;
alt;
  in Continue from System3;
  out Alive to System1;
alt end;
endinstance;
instance System3;
alt begin simplealt shared all;
  in Stop from System2;
  out Agreed to System1;
alt;
  out Continue to System2;
  in Acknowledge from System1;
alt end;
endinstance;
endmsc;
  
```

**Figure I.7 – A non-local choice MSC**

Note that local choice property is not purely local to an alternative frame. Consider, for instance, the example of Figure I.8. According to the semantics of MSCs [b-Reniers99], and [b-ITU-T Z.120 Annex B], instance *System3* can decide to send message *Acknowledge* without waiting for the decision of *System1*. However, if message *Acknowledge* is sent, this means that nothing occurs in the alt frame on instance *System3*, and then that the first behaviour of the alternative is ruled out.



Z.120(09)Amd.2\_Fl.8

```

msc Nonlocal2;
inst System1, System2, System3;
instance System1;
    alt begin simplealt shared alt
        out Interrupt to System2;
    alt;
        out Continue to System2;
    alt end;
endinstance;
instance System2;
    alt begin simplealt shared alt
        in Interrupt from System1;
        out Stop to System3;
    alt;
        in Continue from System3;
    alt end;
in Acknowledge from System3;
endinstance;
instance System3;
    alt begin simplealt shared alt
        in Stop from System2;
    alt;
    alt end;
    out Acknowledge to System2;
endinstance;
endmsc;

```

**Figure I.8 – A non-local MSC**

An important property of local choice MSCs is that they can be implemented, provided some additional control information is added to the contents of messages that are exchanged between instances. For more information, read [b-Helouet00], and [b-Genest02]. Local choice MSCs are a subclass of globally cooperative MSCs described hereafter.

### I.4.3 Globally cooperative MSCs (GCMSCs)

An MSC description is globally cooperative if, in all loops that can appear in the description, the body of the loop does not form disconnected parts of behaviours running on distinct groups of instances. MSC *Counting* in Figure I.9 is not a globally cooperative MSC: the part of the MSC enclosed in the loop frame contains two atomic actions located on different instances. MSC *Data* of Figure I.5 is globally cooperative.

For two globally cooperative MSCs  $M1$  and  $M2$ , the following properties are decidable:

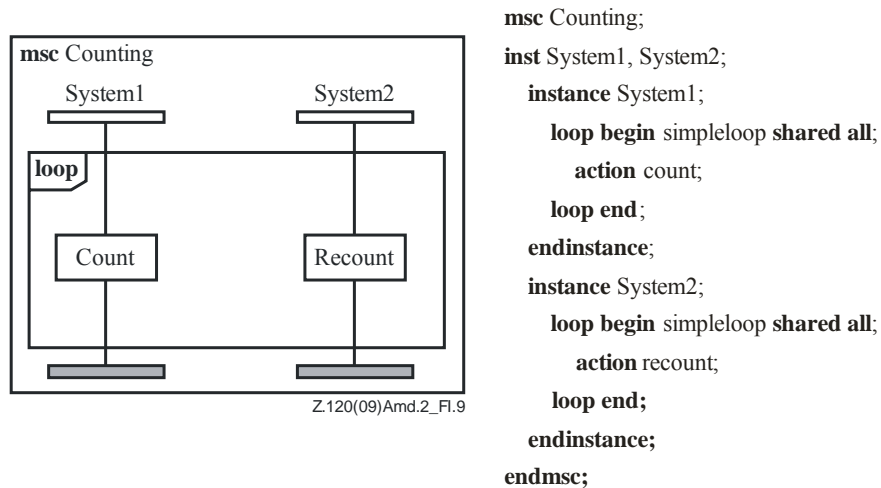
- (*EquivF*)  $F(M1) = F(M2) ?$
- (*RefF*)  $F(M1) \subseteq F(M2) ?$
- (*IntF*)  $F(M1) \cap F(M2) = \phi ?$

When considering two MSC descriptions  $M1$  and  $M2$ , whenever  $M2$  is globally cooperative, the following problems have an algorithmic solution.

$$(EquivF) \quad F(M1) = F(M2) ?$$

$$(RefF) \quad F(M1) \subseteq F(M2) ?$$

There are also generic implementation procedures for globally cooperative MSCs [b-Genest04], but the drawback is that the obtained implementations can contain deadlocks, which is in general an undesirable property of a system.



**Figure I.9 – A non-globally cooperative MSC**

### I.5 Summary of results

We recall here the relationship between different classes of MSCs. Local choice MSCs and regular MSCs are necessarily globally cooperative MSCs. Table I.1 should be read line by line, i.e., for inclusion of subclasses ( $\subseteq$ ), the class mentioned by each line is contained in the class mentioned by the column.

**Table I.1 – Comparison of syntactical subclasses of MSCs**

	RMSC	LMSC	GCMSC	MSC
RMSC	=		$\subseteq$	$\subseteq$
LMSC		=	$\subseteq$	$\subseteq$
GCMSC			=	$\subseteq$
MSC				=

Table I.2 below recalls the decidability of the problems listed in clause I.2. "Yes" means that the considered problem is decidable for the class of MSC. "No" means that the considered problem is undecidable for the class of MSC. Note that for  $Mc1$ ,  $Mc2$  and  $Mc3$ , there is no immediate answer, as the existence of a decision procedure for local choice and globally cooperative depends on the nature of the properties considered.

**Table I.2 – Decidable problems for MSC subclasses**

	<b>Mc1</b>	<b>Mc2</b>	<b>Mc3</b>	<b>EquivL</b>	<b>EquivF</b>	<b>RefL</b>	<b>RefF</b>	<b>IntL</b>	<b>IntF</b>	<b>Rez</b>
<b>MSC</b>	No	No	No	No	No	No	No	No	No	No
<b>RMSC</b>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
<b>LMSC</b>	?	?	?	Yes	Yes	Yes	Yes	Yes	Yes	No
<b>GCMSC</b>	?	?	?	Yes	Yes	Yes	Yes	Yes	Yes	No

Table I.3 below recalls the different classes of MSC that can be implemented. Implementation mechanisms can use additional information on message to ensure correctness of implementation. The notion of correctness is also subject to several interpretations. Some implementation approaches (see for instance [b-Genest04]) consider that an MSC description and an implementation are only compared according to their correct runs, and do not consider deadlocked executions of the implementation. This way, an implementation that can deadlock can be considered as correct. For each subclass of MSC in Table I.3, we indicate whether an implementation mechanism has been proposed, and the restrictions (presence of deadlocks).

**Table I.3 – Implementation of MSCs**

	<b>Implementation</b>
MSC	?
RMSC	?
LMSC	With additional control information on messages
GCMSC	With deadlocks

## **I.6 Recommendations**

We give here a list of recommendations according to the targeted application for a MSC specification.

### **I.6.1 Model checking**

If the targeted application is model-checking of message sequence charts, the MSC description should remain regular, that is, for every loop:

- All messages sent from an instance I1 to an instance I2 should be acknowledged, either directly or indirectly.
- If the loop comports two atomic actions located on different instances, then there must be a direct or indirect message exchange between the instances where these actions are located.

### **I.6.2 Comparison of MSC specifications**

If the targeted application is a comparison of specifications, then the message sequence charts used should remain globally cooperative; that is, for every loop, each instance or group of instance must either send or receive a message from the rest of the instances participating to the loop.

### **I.6.3 Implementation**

If the targeted application is implementation of specifications, then the message sequence charts used should remain local choice; that is, for every alternative, the parts of the MSC in the scope of each part of this alternative should start with events located on a single instance.

## Bibliography

- [b-ITU-T T.50] Recommendation ITU-T T.50 (1992), *International Reference Alphabet (IRA) (formerly International Alphabet No. 5 or IA5); Information technology – 7-bit coded character set for information interchange.*
- [b-ITU-T Z.120 Annex B] Recommendation ITU-T Z.120 Annex B (1998), *Formal semantics of message sequence charts.*
- [b-Alur05] Alur Rajeev, Etessami Kousha, and Yannakakis Mihalis (2005), *Realizability and verification of MSC graphs*, Theoretical Computer Science. 331(1): 97-114.
- [b-Clarke99] Clarke Edmund M., Grumberg Orna, and Peled Doron (1999), *Model Checking*, MIT Press, December.
- [b-Genest02] Genest Blaise, Muscholl Anca, Seidl Helmut, and Zeitoun Marc (2002), *Infinite-State High-Level MSCs: Model-Checking and Realizability*, Proceedings of ICALP, pp. 657-668.
- [b-Genest04] Genest Blaise, Kuske Dietrich, and Muscholl Anca (2004), *A Kleene Theorem for a Class of Communicating Automata with Effective Algorithms*, Proceedings of DLT 2004, pp. 30-48, LNCS 3340.
- [b-Helouet00] Hélouët Loïc, and Jard Claude (2000), *Conditions for synthesis of communicating automata from HMSCs*, 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Berlin, 3-4 April.
- [b-Holzmann99] Holzmann Gerard J., and Smith Margaret H. (1999), *Software Model Checking*, Proceedings of FORTE 1999, pp. 481-497.
- [b-Khendek99] Abdalla Miguel, Khendek Ferhat, and Butler Greg (1999), *New Results on Deriving SDL Specifications from MSCs*, in the Proceedings of SDL Forum'99, Elsevier Science B. V., R. Dssouli, G.v. Bochmann and Y. Lahav (eds.), Montreal, Canada, June 21-25.
- [b-Reniers99] Mauw Sjouke, and Reniers Michel A. (1999), *Operational Semantics for MSC'96*, Computer Networks and ISDN Systems 31(17):1785-1799, Elsevier Science B.V.





## **SERIES OF ITU-T RECOMMENDATIONS**

Series A	Organization of the work of ITU-T
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	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Terminals and subjective and objective assessment methods
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, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
<b>Series Z</b>	<b>Languages and general software aspects for telecommunication systems</b>