

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



International
Telecommunication
Union

Documents
Document Sharing Hub

ITU-T Z-SERIES RECOMMENDATIONS
LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	Z.100–Z.109
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
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

		Page
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

	Page
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

	Page
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.

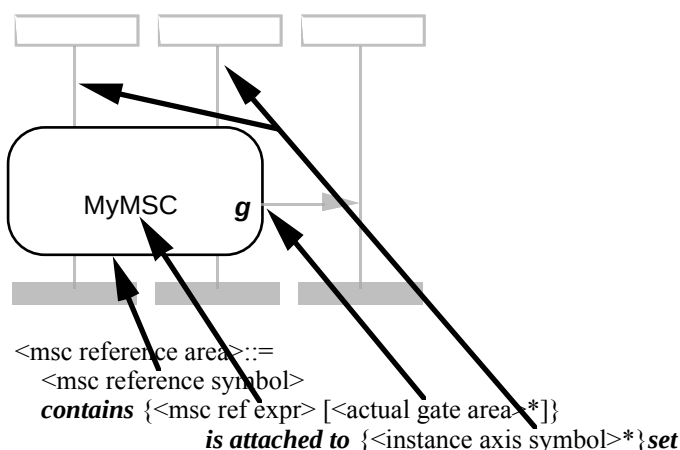


Figure 1 – Example for 'contains'

is followed by

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

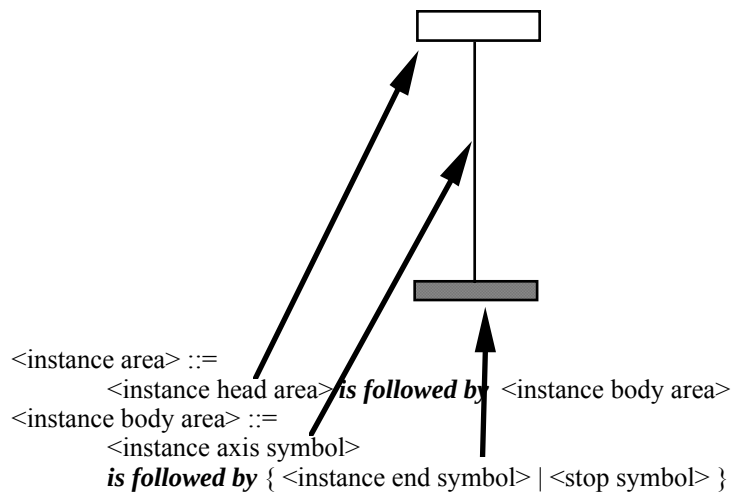


Figure 2 – Example for 'is followed by'

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.

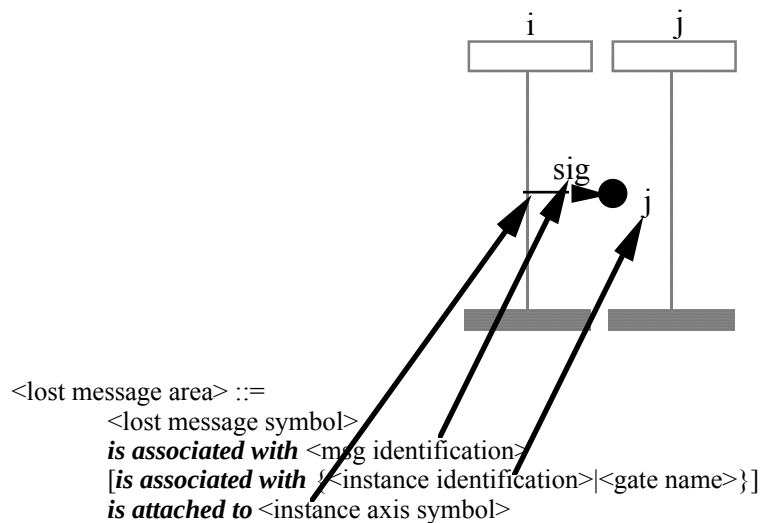


Figure 3 – Example for 'is associated with'

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.

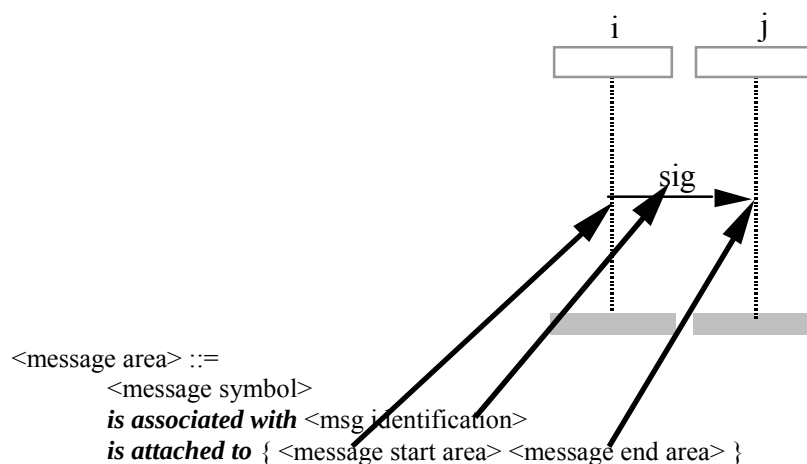


Figure 4 – Example for 'is attached to'

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:

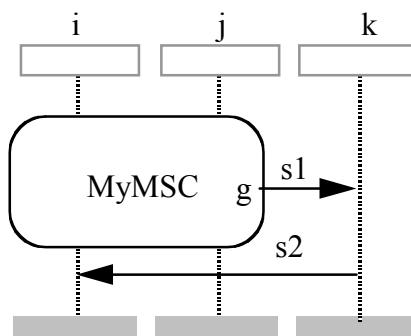
```
<message end area> ::=
  <message in area> | <actual in gate area>
    | <def out gate area> | <inline gate area>
<message in area> ::=
  <message in symbol>
    is attached to <instance axis symbol>
    is attached to <message symbol>
<message in symbol> ::=
  <void symbol>
```

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

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

<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 strings 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).

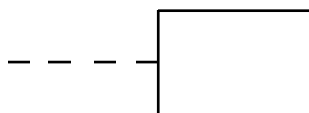
Secondly there is the *comment* which is a notation to represent informal explanations associated with symbols or text.

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 $\text{in}(m_i)$) and outputs (labelled by $\text{out}(m_i)$) of the Message Sequence Chart in Figure 6(a) we derive the following ordering relation: $\text{out}(m_2) < \text{in}(m_2)$, $\text{out}(m_3) < \text{in}(m_3)$, $\text{out}(m_4) < \text{in}(m_4)$, $\text{in}(m_1) < \text{out}(m_2) < \text{out}(m_3) < \text{in}(m_4)$, $\text{in}(m_2) < \text{out}(m_4)$ 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)).

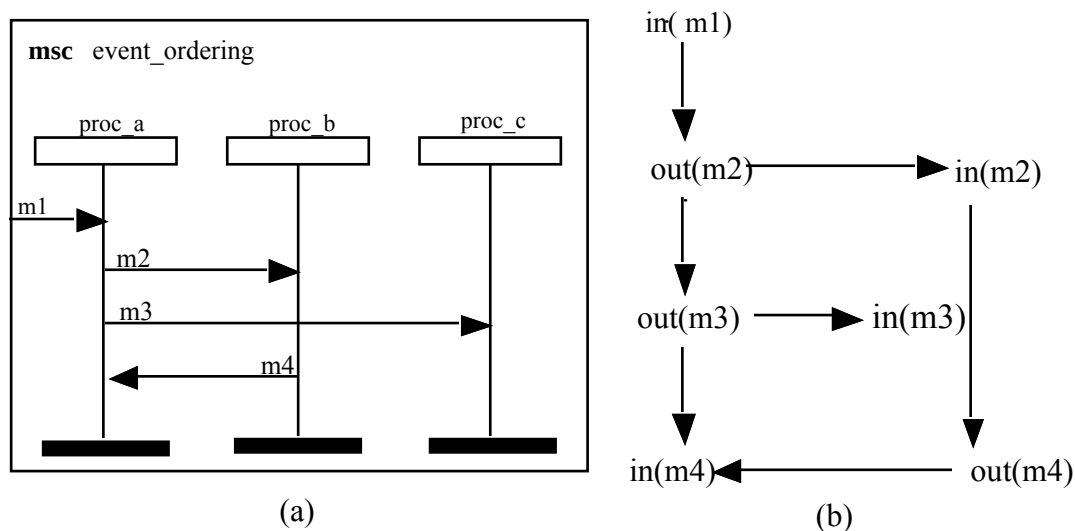


Figure 6 – Message Sequence Chart and corresponding connectivity graph

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>

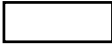
Concrete graphical grammar

<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.¹

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.

¹ 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.

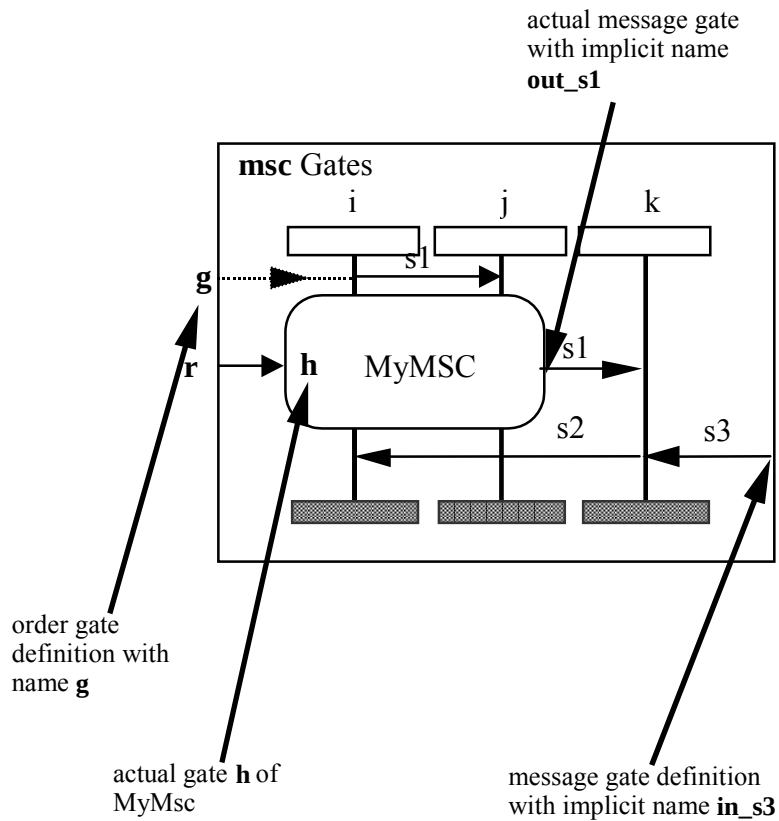


Figure 7 – Example of gates

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> }

```