



INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.120

(10/96)

SERIES Z: PROGRAMMING LANGUAGES

Criteria for the use and applicability of formal Description
Techniques

Message Sequence Chart (MSC)

ITU-T Recommendation Z.120

(Previously CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS
PROGRAMMING LANGUAGES

Specification and Description Language (SDL)	Z.100–Z.109
Criteria for the use and applicability of formal Description Techniques	Z.110–Z.199
ITU-T High Level Language (CHILL)	Z.200–Z.299
MAN-MACHINE LANGUAGE	Z.300–Z.499
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.399
Miscellaneous	Z.400–Z.499

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

FOREWORD

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

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

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1 (Helsinki, March 1-12, 1993).

ITU-T Recommendation Z.120 was revised by ITU-T Study Group 10 (1993-1996) and was approved by the WTSC (Geneva, 9-18 October 1996).

NOTE

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

© ITU 1997

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

CONTENTS

	<i>Page</i>
1	Introduction 1
1.1	Introduction to MSC 1
1.2	Meta-language for textual grammar..... 2
1.3	Meta-language for graphical grammar..... 2
2	General rules 7
2.1	Lexical rules..... 7
2.2	Visibility and naming rules..... 11
2.3	Comment..... 11
2.4	Drawing rules 12
2.5	Paging of MSCs 13
3	Message Sequence Chart document 13
4	Basic MSC 14
4.1	Message Sequence Chart 14
4.2	Instance 19
4.3	Message 20
4.4	Environment and gates 24
4.5	General ordering 30
4.6	Condition 31
4.7	Timer..... 33
4.8	Action 36
4.9	Instance creation 36
4.10	Instance stop 37
5	Structural concepts 37
5.1	Coregion 37
5.2	Instance decomposition..... 33
5.3	Inline expression..... 33
5.4	MSC reference 36
5.5	High-level MSC (HMSC)..... 39
6	Message Sequence Chart examples..... 43
6.1	Standard message flow diagram 43
6.2	Message overtaking 44
6.3	MSC basic concepts..... 45
6.4	MSC-composition/MSC-decomposition..... 46
6.5	MSC with time supervision 48
6.6	MSC with message loss 49
6.7	Local conditions..... 50
6.8	Shared condition and messages with parameters 51
6.9	Creating and terminating processes 52
6.10	Coregion 52
6.11	Generalized ordering within a coregion..... 53
6.12	Generalized ordering between different instances 54
6.13	Instance decomposition..... 54
6.14	Inline Expression with alternative composition..... 55
6.15	Inline Expression with gates 57
6.16	Inline Expression with parallel composition..... 58
6.17	MSC reference 59

	<i>Page</i>
6.18 MSC reference with gate	60
6.19 High-level MSC with free loop	61
6.20 High-level MSC with loop.....	61
6.21 High-level MSC with alternative composition	62
6.22 High-level MSC with parallel composition	63
Annex A – Index	65

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

The main 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

The status of basic MSCs is stable, including the constructs for instance, instance creation and termination, message exchange, action, timer handling and condition. For the new structural concepts – generalized coregion, inline expression, MSC reference, HMSC – further developments may be expected.

ASSOCIATED WORK

- ITU-T Recommendation Q.65 (1997), *The unified functional methodology for the characterization of services and network capabilities*.
- ITU-T Recommendation X.210 (1993), *Information technology – Open Systems Interconnection – Basic Reference Model: Conventions for the definition of OSI services*. (Common text with ISO/IEC.)
- ITU-T Recommendation Z.100 (1993), *CCITT Specification and Description Language (SDL)*.

MESSAGE SEQUENCE CHART (MSC)

(revised in 1996)

1 Introduction

1.1 Introduction to MSC

A Message Sequence Chart (MSC) shows sequences of messages interchanged between system components and their environment. In SDL (Recommendation Z.100, ITU 1996) the system components are services, processes and blocks. MSCs have been used for a long time by ITU-T Study Groups (former CCITT) in their Recommendations and within industry, according to different conventions and under various names such as Signal Sequence Chart, Information Flow Diagram, Message Flow and Arrow Diagram.

The reason to standardize MSCs is to make it possible to provide tool support for them, to exchange MSCs between different tools, to ease the mapping to and from SDL specifications and to harmonize the use within ITU.

One part of the standardization work is to provide a clear definition of the meaning of an MSC. This is done in this Recommendation [Annex B (1995)] by providing a formal semantics based on process algebra.

Another way to explain the meaning of MSCs may be by relating them to SDL specifications, as follows: An MSC describes one or more traces of an SDL system specification. Accordingly, an MSC can be derived from an existing SDL system specification, and may then be used, for example, to record the result of an animation.

Due to the standardization, the importance of MSCs for system engineering has increased considerably. Accordingly MSCs may serve as:

- a) an overview of a service as offered by several entities;
- b) a statement for requirements specification;
- c) a basis for elaboration of SDL specifications;
- d) a basis for system simulation and validation;
- e) a basis for selection and specification of test cases;
- f) a specification of communication;
- g) an interface specification;
- h) a formalization of use cases within object-oriented design and analysis.

Since an MSC usually only covers a partial behaviour, the selection of partial behaviours is a crucial task. The candidates for MSCs are primarily the 'standard' cases. Further cases are generally built on them and cover exceptional behaviours, e.g. caused by errors of various kinds.

In the following, a syntax for Message Sequence Charts is presented in textual and graphical representation. A corresponding informal (verbal) semantics description is provided.

This Recommendation is structured in the following manner: In clause 2, general rules concerning syntax, drawing and paging are outlined. In clause 3, a syntax definition for the Message Sequence Chart document which is a collection of Message Sequence Charts is provided. Clause 4 contains the syntax definition for Message Sequence Charts and the syntax rules for the basic constituents, i.e. instance, message, environment, general ordering, condition, timer, action, instance creation and termination. In Clause 5, 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 (5.1) and instance decomposition (5.2). The most advanced concepts – inline expression (5.3), MSC reference (5.4), high level MSC (5.5) – permit MSC composition and reusability of (parts of) MSCs on different levels. In clause 6, examples are provided for all MSC-constructs. Annex A contains an index for the <keyword>s and non-terminals.

1.2 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>. Note that the two special terminals <name> and <character string> may also have semantics stressed as defined below.

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 graphical grammar. For example:

```
<msc inst interface> ::=
    inst <instance 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, <msc inst interface> and <instance list> in the example above are non-terminals; **inst** is a terminal symbol.

Sometimes the 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:

```
<msc document body> ::=
    { <message sequence chart> | <msc diagram> }*
```

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 an <msc document body> may be empty but may also contain any number of <message sequence chart>s and/or <msc diagram>s.

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>.

1.3 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 outlooks 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.

contains:

"<area1> **contains** <area2>" means that <area2> is contained within <area1>, in principle geometrically, but also logically. See Figure 1-1.

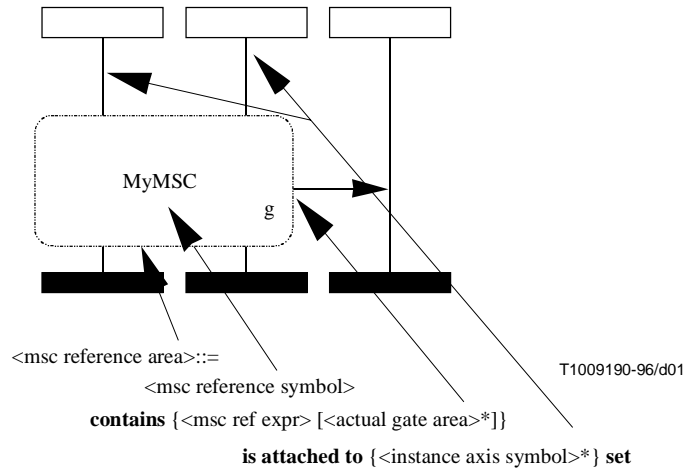


Figure 1-1/Z.120 – Example for ‘contains’

is followed by:

"<area1> **is followed by** <area2>" means that <area2> is logically and geometrically related to <area1>. See Figure 1-2.

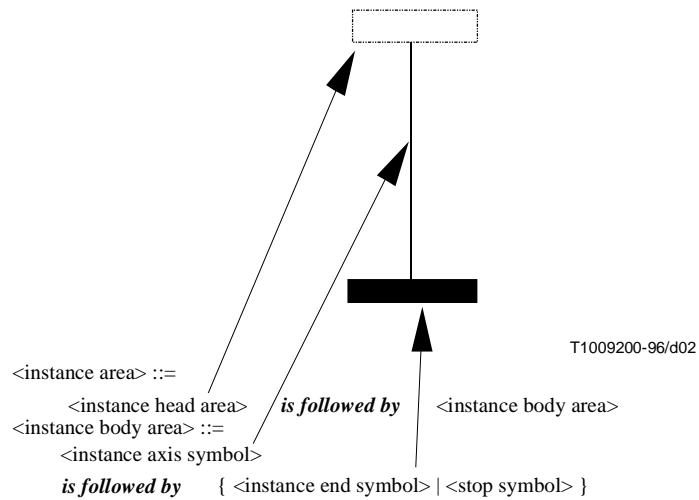


Figure 1-2/Z.120 – Example for ‘is followed by’

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. See Figure 1-3.

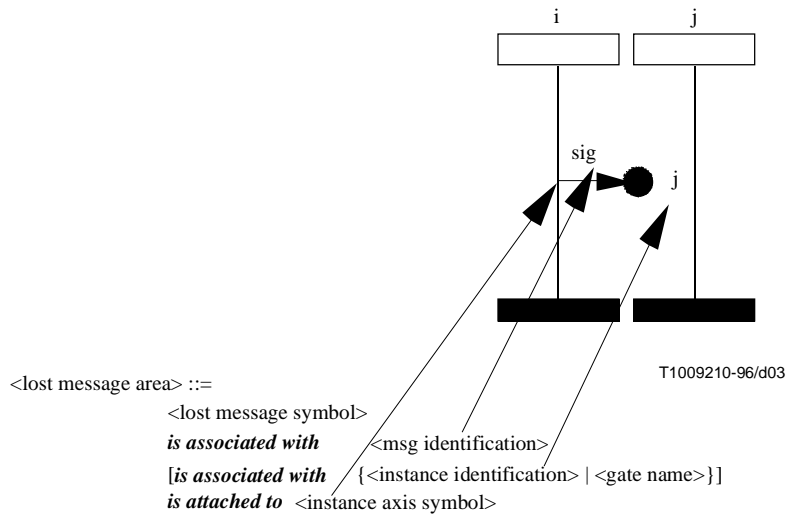


Figure 1-3/Z.120 – Example for 'is associated with'

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 "<P>::=<area1> **is attached to** <area2>" is that, if non-terminal <P> 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. See Figure 1-4.

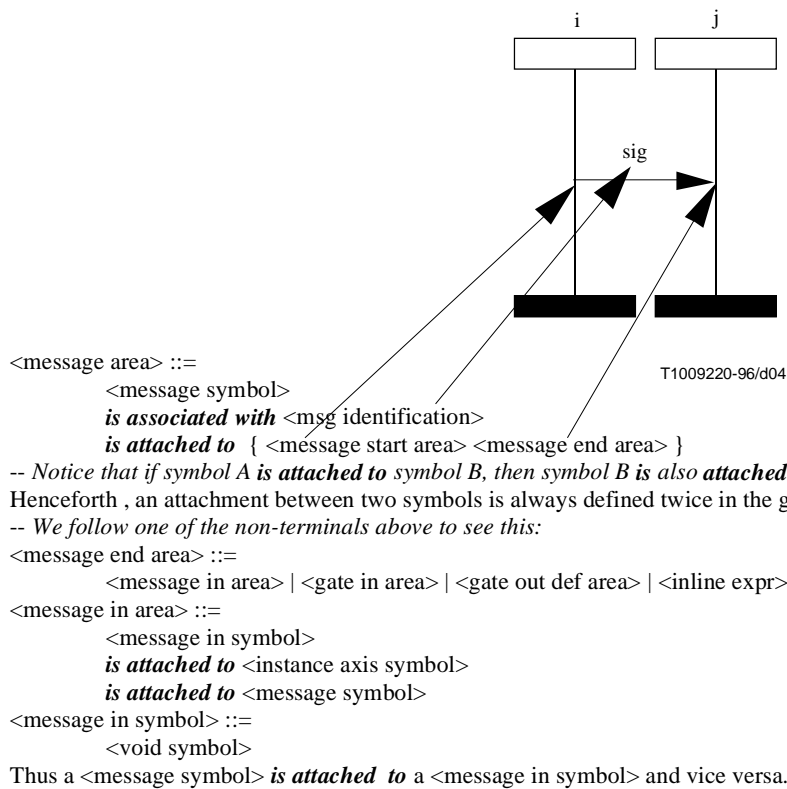
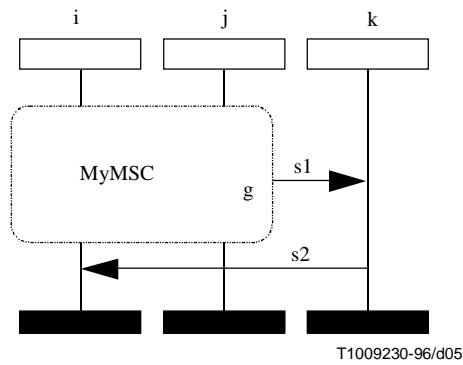


Figure 1-4/Z.120 – Example for 'is attached to'

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. See Figure 1-5.



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

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

The above figure 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.

Figure 1-5/Z.120 – Example for 'above'

set:

The *set* metasymbol is a postfix operator operating on the immediately preceding syntactic elements within curly brackets, and indicating an (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.

Examples

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

```

<word> ::=      {<alphanumeric> | <full stop>}*
               <alphanumeric>
               {<alphanumeric> | <full stop>}*

<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 square bracket>
                  | <right square bracket>
                  | <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> ::= }
<overline> ::= ~
<upward arrow head> ::= ^
<full stop> ::= .
<underline> ::= _

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

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

<apostrophe> ::= '
<other character> ::= ?   | &   | %   | +   | -   | !   | /   | >   | *   | "   | <   | =
<special> ::=      (   | )   | ,   | ;   | :

```

```

<composite special> ::= << | >>
<note> ::= /* <text> */
<name> ::= <word> { <underline> <word> }*
<keyword> ::=
    action
    | all
    | alt
    | as
    | before
    | begin
    | block
    | by
    | comment
    | concurrent
    | condition
    | connect
    | create
    | decomposed
    | empty
    | end
    | endconcurrent
    | endinstance
    | endmsc
    | endexpr
    | env
    | exc
    | expr
    | external
    | found
    | from
    | gate
    | in
    | inf
    | inline
    | inst
    | instance
    | loop
    | lost
    | msc
    | mscdocument
    | msg
    | opt
    | order
    | out
    | par
    | process
    | reference
    | related
    | reset
    | service
    | seq
    | set
    | shared
    | stop
    | subst
    | system

```

	text
	timeout
	to
	via

The <national> characters are represented above as in the International Reference Version of CCITT Alphabet No. 5 (Recommendation T.50). The responsibility for defining the national representations of these characters lies with national standardization bodies.

Control characters are defined as in Recommendation 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 CCITT Alphabet No. 5 character for a space.

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 except <character string>, <letter>s are always treated as if uppercase. (The treatment of <national>s may be defined by national standardization bodies.)

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>.

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.

The scope unit for MSC is the MSC document. No two entities within this scope unit and belonging to the same entity class can have the same name. Different occurrences of a condition name, timer name and message name denote the same entity. The name of an entity is visible within the enclosing scope unit, but not outside. Only visible names can be used when referencing entities.

Gate names are always associated with a specific MSC. Therefore the same gate names may be applied to different MSCs.

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> ::=



T1009240-96/d06

A <comment symbol> can be attached to the following graphical symbols: <text symbol>, <instance head symbol>, <instance end symbol>, <message out symbol>, <message in symbol>, <lost message symbol>, <found message symbol>, <general order symbol>, <condition symbol>, <set symbol>, <reset symbol>, <timeout symbol>, <action symbol>, <createtime symbol>, <stop symbol>, <coregion symbol>, <inline expression symbol>, <separator symbol>, <msc reference symbol>, <hmsc start symbol>, <hmsc end symbol>, <par frame symbol>, <connection point symbol>.

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> ::=



T1009250-96/d07

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 of message symbol and general ordering symbol with message symbol, general ordering symbol, lines in timer symbols, createline symbol, instance axis symbol, dashed line in comment symbol and condition symbol;
- for the crossing of lines in timer symbols with message symbol, general ordering symbol, lines in timer symbols, createline symbol, dashed line in comment symbol and condition symbol;
- for the crossing of createline symbol with message symbol, general ordering symbol, lines in timer symbols, instance axis symbol and dashed line in comment symbol;

- d) for the crossing of condition symbol with instance axis symbol, message symbol, general ordering symbol, lines in timer symbols;
- e) for the crossing of inline expression symbol with instance axis symbol;
- f) for the crossing of reference symbol with instance axis symbol;
- g) for the crossing of action symbol with instance axis symbol in line form, and the overlap of action symbol with instance axis symbol in column form;
- h) for the crossing of hmsc line symbol with 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 4.6) crosses an instance which is not involved in this condition, the instance axis is drawn through.

If a shared reference (see 5.4) crosses an instance which is not involved in this reference, the instance axis is drawn through.

If a shared inline expression (see 5.3) 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 and time-out. The following events are outgoing events: message output, lost message, timer set, timer reset 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 clause 5).

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 header contains the document name and optionally, following the keyword **related to**, the identifier (pathname) of the SDL-document to which the MSCs refer.

Concrete grammar

```
<msc document> ::=
    <msc document head> <msc document body>
```

<msc document head> ::=
 <document head> | <document head area>

<msc document body> ::=
 { <message sequence chart> | <msc diagram> }*

Concrete textual grammar

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

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

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

<qualifier> ::= << <text> >>

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

Concrete graphical grammar

<document head area> ::=
 <frame symbol> **contains** <document head>

Semantics

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

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.

The Message Sequence Chart heading consists of the Message Sequence Chart name and (optionally) a list of the instances being contained in the MSC and a list of the gates of the MSC.

An MSC is described either by instances and events or by an expression that relates MSC references without explicitly mentioning the instances [see 5.5, High-level MSC (HMSC)].

Concrete textual grammar

<message sequence chart> ::=
 msc <msc head> { <msc body> | **expr** <msc expression> } **endmsc** <end>

<msc head> ::= <msc name> <end> [<msc interface>]

<msc interface> ::= [<msc inst interface>] [<msc gate interface>]

<msc inst interface> ::=
 inst <instance list> <end>

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

```

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

<kind denominator> ::=
    system | block | process | service | <name>

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

<msc gate def> ::= gate { <msg gate> | <order gate> } <end>

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

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

<msc body> ::= <msc statement> *

<msc statement> ::=
    <text definition> | <event definition> | <old instance head statement> <instance event list>

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

<instance event list> ::=
    { <instance event> <end> } +

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

<orderable event> ::=
    [ <event name> ]
    { <message event> | <incomplete message event> | <create> | <timer statement> | <action> }
    [ before <event name list> ]

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

<event name list> ::=
    { <event name> | <gate name> } [ , <event name list> ]

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

<non-orderable event> ::=
    <coregion> | <shared condition> | <shared msc reference> | <shared inline expr>
    | <instance head statement> | <instance end statement> | <stop>

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

<multi instance event list> ::=
    { <multi instance event> <end> } +

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

<old instance head statement> ::=
    instance <instance name> [ [ : ] <instance kind> ] [ <decomposition> ] <end>

```

The non-terminal <old instance head statement> is only intended for backward compatibility with existing MSC-92 descriptions.

For each <instance head statement> or <old instance head statement> there must also be a corresponding <instance end statement> or a <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>. For each instance there must be no <instance head statement> after <old instance head statement>.

The <instance list> in the <msc interface>, if present, must contain the same instances as specified in the <msc body>.

Concrete graphical grammar

<msc diagram> ::= <msc symbol> **contains** { <msc heading> { <msc body area> | <mscexpr area> } }

<msc symbol> ::= <frame symbol>
is attached to { <def gate area>* } **set**

<frame symbol> ::=



T1009260-96/d08

<msc heading> ::= **msc** <msc name>

<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>* } **set**

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

<instance event area> ::=
 <message event area>
 | <timer area>
 | <concurrent area>
 | <action area>

<shared event area> ::=

- | <condition area>
- | <msc reference area>
- | <inline expression area>

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. Although the behaviour of the environment is non-deterministic, it is assumed to obey the constraints given by the Message Sequence Chart.

No global time axis 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 5.1, 5.30), 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 so called 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 reflexive is called partial order.

For the message inputs [labelled by in(mi)] and outputs [labelled by out(mi)] of the Message Sequence Chart in Figure 4-1 a), we derive the following ordering relation: out(m2) < in(m2), out(m3) < in(m3), out(m4) < in(m4), in(m1) < out(m2) < out(m3) < in(m4), in(m2) < out(m4) 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 4-1 b)].

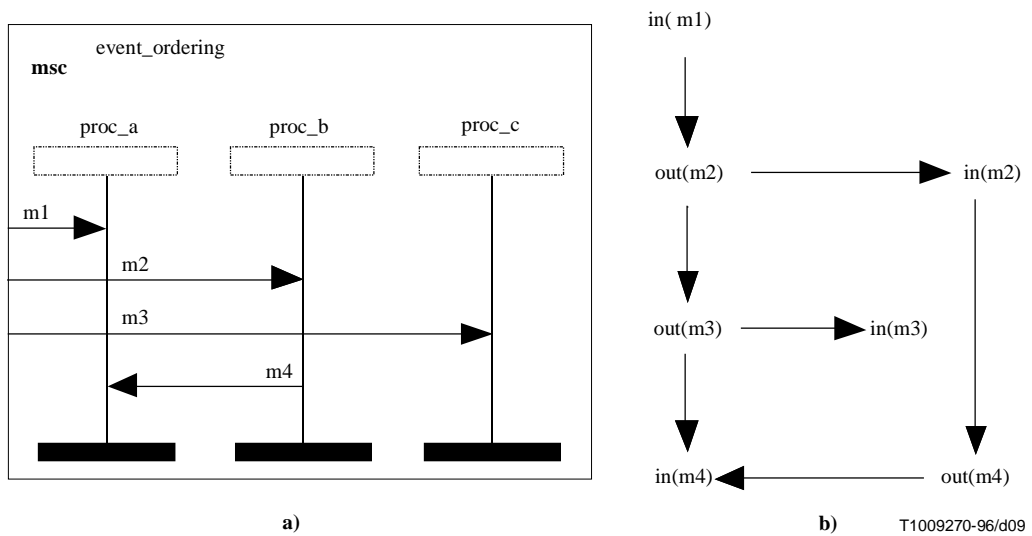


Figure 4-1/Z.120 – Message Sequence Chart and corresponding connectivity graph

A formal semantics of MSCs based on process algebra is provided in Annex B (1995). 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 <event list> or by <instance name list> followed by an attached <multi instance event> whereby a colon symbol serves as separator. The non-terminal <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 optional <msc interface> which describes the interface of the MSC with its environment consists of the <msc inst interface> and the <msc gate interface>. The <msc inst interface> provides a declaration of the instances, i.e. <instance name> and optionally <instance kind>. Since normally one MSC only consists of a section of a system run the <msc inst interface> describes the connection points of instances to the environment. 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 of entities. An instance of an entity is an object which has the properties of this entity. Related to SDL, an entity may be an SDL-process, block or service. Within the instance heading the entity 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>]

<instance end statement> ::=

endinstance

Concrete graphical grammar

<instance 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>]

<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> ::=



T1009280-96/d10

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. In the latter case, the colon symbol is optional (and has to occur above if present).

Semantics

Within the Message Sequence Chart body the instances are defined. The instance end symbol determines the end of the description of the instance within this MSC. It does not describe the termination of the instance (see 4.10, instance stop). Correspondingly, the instance head symbol determines the start of the description of the instance within the MSC. It does not describe the creation of the instance (see 4.9, instance creation).

In the context of SDL an instance may refer to a process (keyword **process**), service (keyword **service**) or block (keyword **block**). Outside of SDL, it may refer to any kind of entity. The instance definition provides an event description for message inputs and message outputs, actions, shared and local conditions, timer, instance creation, instance stop. Outside of coregions (see 5.1) and inline expressions (see 5.3), a total ordering of events is assumed along each instance-axis. Within coregions no time 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, and an input is to either the environment (a gate) or an instance.

An incomplete message occurs if either the output or the input is lost and is represented by only one event.

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 4-1 a) can be split into out(m2) (output) and in(m2) (input). In the textual representation the message input is represented by the keyword **in**, the message output by the keyword **out**, both followed by the message name and optionally a message instance name. To a message, parameters may be assigned between parentheses.

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, may be 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, can be defined by the keyword **found** in the textual representation and by a white hole in the graphical representation.

By means of the keyword **before** in the textual representation, a time ordering of message events on different instances may be defined. In the graphical representation, synchronization constructs in form of connection lines define these generalized ordering concepts.

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> ] [ (<parameter list>) ]

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

<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 only the keyword **env** is used 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 only the keyword **env** is used 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**).

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 <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. This is the case if the connectivity graph (see 4.1) 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>s have to be identical.

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>

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

<void symbol> ::= .

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

NOTE 1 – 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>

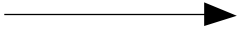
<message in symbol> ::=
 <void symbol>

NOTE 2 – 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>
 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> ::=
 
 T1009290-96/d11

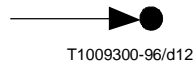
NOTE 3 – The mirror image of the <message symbol> is allowed. The point of the arrow head should be on the instance axis.

NOTE 4 – 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> ::=

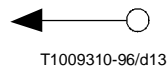


NOTE 5 – The <lost message symbol> describes the event of the output side, i.e. the solid line starts on the <message start area> where the event occurs. The optional intended target of the message can be given by an identifier associated with the symbol. The target identification should be written close to the black circle, while the message identification should be written close to the arrow.

NOTE 6 – 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> ::=



NOTE 7 – The <found message symbol> describes the event of the input side (the arrowhead) which should 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 part.

NOTE 8 – The mirror image of the symbol may also be used.

Semantics

For an MSC the message-output denotes the message sending (corresponding to SDL-output), the message-input, the message consumption (corresponding to SDL-input). No special construct is provided for message reception (input into the buffer). No type definition is attached to parameters within the parameter list.

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

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

4.4 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.

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 incomplected 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.

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).

Gates which are not connected on an MSC reference are implicitly defined to propagate to the next enclosing frame (either MSC frame or inline expression frame). A propagated gate will have the same gate name as the gate it propagated from. This is a practical shorthand which saves a connection line which otherwise would contribute to the cluttering of the diagram.

Gates on inline expressions are merely transit points on the frame of the inline expression. If the gate is not continued outside the frame, the following implicit rules apply:

- 1) If there are other gates with the same name of the same inline expression, the continuation given for one of the gates holds for all.
- 2) If there are no other gates with the same name and no continuation exists, an implicit continuation to the next enclosing frame (either MSC frame or inline expression frame) is assumed. See Figure 4-2.

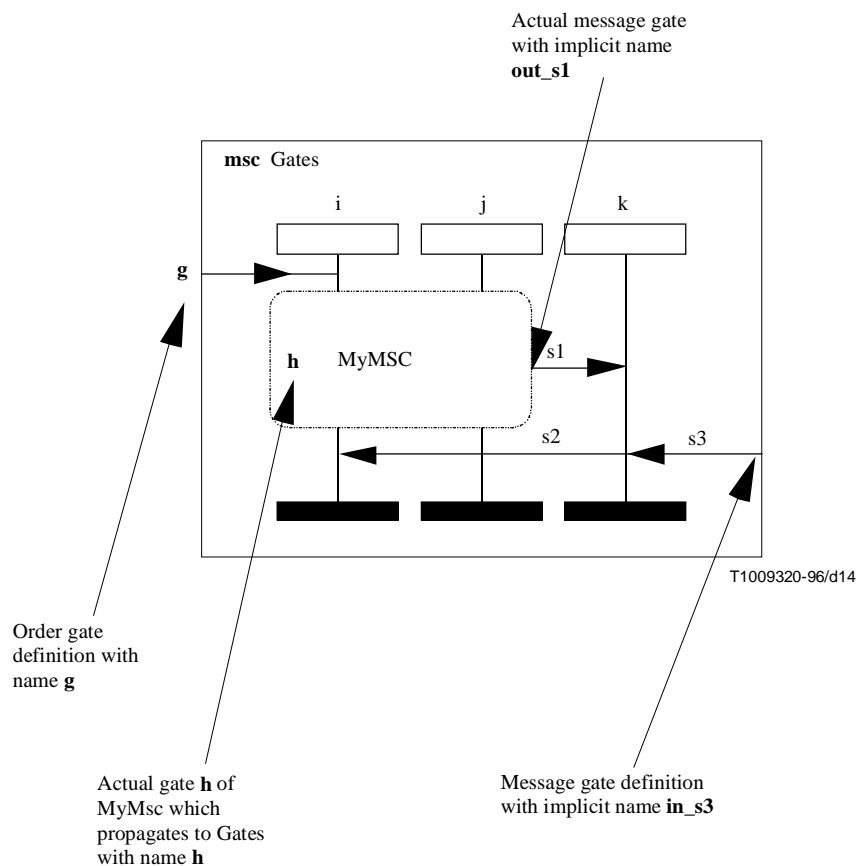


Figure 4-2/Z.120 – Example of gates

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>

<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>

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 order out gate> ::=

<gate name>
[**external before** <order dest>]

<inline order in gate> ::=

<gate name> **before** <order dest>
[**external**]

Concrete graphical grammar

<inline gate area> ::=

{ <inline out gate area> | <inline in 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> } ]

```

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. When there is no message attached outside the gate, this means that the gate proliferates to the next frame.

```

<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> }
    is attached to <msc symbol>

```

```

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

```

NOTE 1 – The <message symbol> or <found message symbol> should 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> }

```

NOTE 2 – The <message symbol> or <lost message symbol> should have its open end attached to the <def in 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>

<gate identification> ::=

<gate name>

<actual gate area> ::=

<actual out gate area> | <actual 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> }]

NOTE 3 – 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> }]

NOTE 4 – The <actual in gate area> is attached to the arrow head end of the <message symbol> or <found message> 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 Semantics

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

4.5 General ordering

General ordering is used to describe when two events are ordered in time, but where this cannot be deduced from the ordering imposed by the messages.

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> ::=



NOTE: The <general order symbol1> is allowed to have a staircase like shape. 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 <coregion symbol2> (column form). The connection lines from the <general order symbol1> to the <ordered event area> s that it orders should be horizontal.

<general order symbol2> ::=



T1009330-96/d15

<general order symbol1>

NOTE 2 – The <general order symbol2> may have any orientation and also be bent.

<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>
- | <timer area>
- | <create area>
- | <action symbol>

Static Semantics

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

Semantics

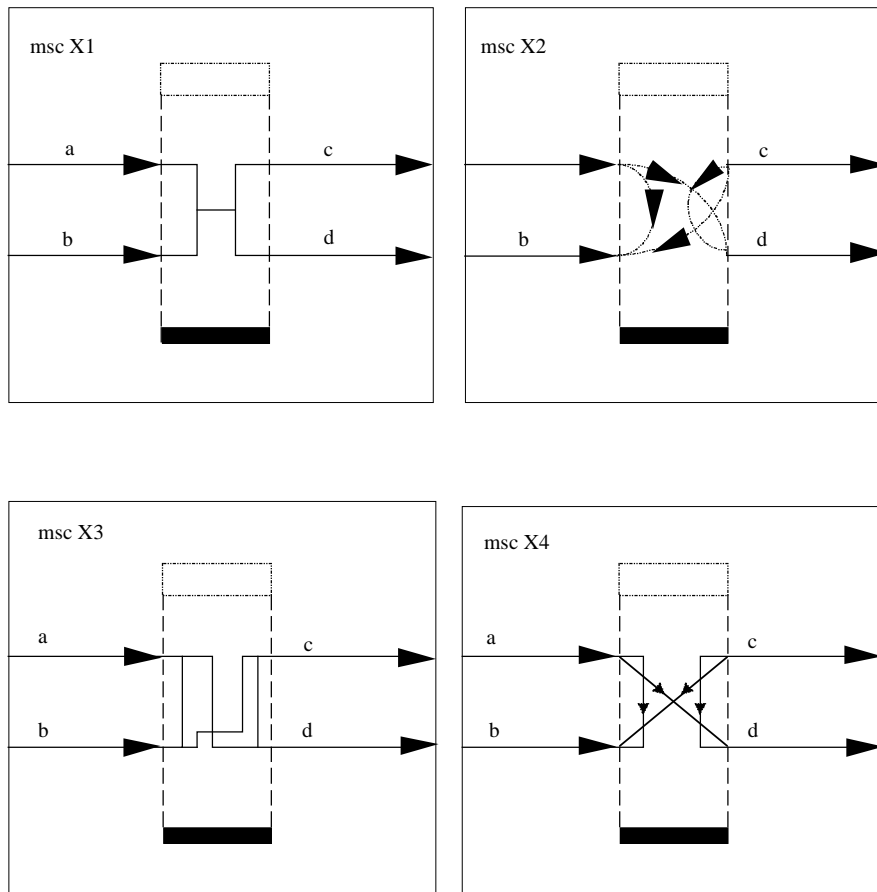
The general order symbols describe orders between events which otherwise would not be ordered. This is particularly useful when events in a coregion should be more ordered than all possible permutations.

Example of general ordering

Figure 4-3 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.

4.6 Condition

A condition describes either a global system state (global condition) referring to all instances contained in the MSC or a state referring to a subset of instances (non-global condition). In the second case the condition may be local, i.e. attached to just one instance. 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 several instances then the keyword **shared** together with the instance list denotes the set of instances by which the condition is shared. A global condition referring to all instances may be defined by means of the keyword **shared all**.



T1009340-96/d16

Figure 4-3/Z.120 – Example for general ordering

Global conditions can be used to restrict how MSCs can be composed in High-level MSCs. This is further discussed in 5.5 'High-level MSC (HMSC)'.

Concrete textual grammar

<shared condition> ::=

 <condition identification> <shared>

<condition identification> ::=

condition <condition name list>

<condition name list> ::=

 <condition name> { , <condition name> }*

<shared> ::=

shared { [<shared instance list>] | **all** }

<shared instance list> ::=

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

<condition> ::=

 <condition identification>

To 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

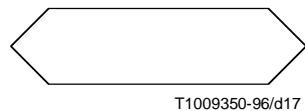
<condition> attached to instance *b*. If instance *a* and instance *b* share the same <condition> then for each message exchanged between these instances, the <message input> and <message output> must be placed both before or both after the <condition>. If two conditions are ordered directly (because 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.

Concrete graphical grammar

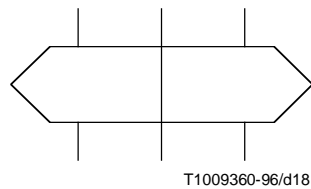
<condition area> ::=

<condition symbol> **contains** <condition name list>
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:



Semantics

Global conditions, representing global system states, refer to all instances involved in the MSC. For each Message Sequence Chart:

- an initial global condition (global initial state);
- a final global condition (global final state); and
- intermediate global conditions (global intermediate states),

may be specified using the keyword **shared all** in the textual representation.

Initial, intermediate and final global conditions are not introduced merely for documentation purposes in the sense of comments or illustrations. Global conditions indicate possible continuations of Message Sequence Charts containing the same set of instances by means of condition name identification. Conditions may thus be used to restrict possible compositions of MSCs. The actual composition of MSCs is defined in form of HMSCs (see 5.5). The prescribed compositions have to be in agreement with the conditions contained in the MSCs according to the definition of allowed continuations.

4.7 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 reset (time supervision) may be specified. In addition, the individual timer constructs – timer setting, reset/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 set symbol has the form of an hour glass connected with the instance axis by a (bent) line symbol. Time-out is described by a message arrow pointing at the instance which is attached to the hour glass symbol. The reset symbol has the form of a cross symbol which is connected with the instance by a (bent) line.

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

Concrete textual grammar

```

<timer statement> ::=
    <set> | <reset> | <timeout>

<set> ::=
    set <timer name> [ , <timer instance name> ] [ (<duration name>) ]

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

<timeout> ::=
    timeout <timer name> [ , <timer instance name> ]
    
```

In case where the <timer name> is not sufficient for a unique mapping the <timer instance name> has to be employed.

Concrete graphical grammar

```

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

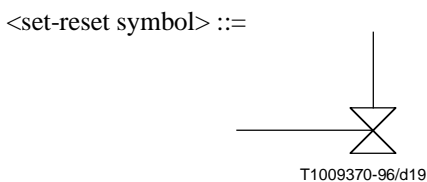
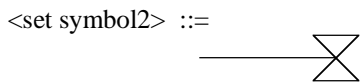
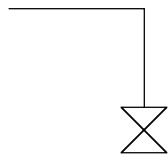
<timer set area> ::= <timer set area1> | <timer set area2>

<timer set area1> ::=
    <set symbol> is associated with <timer name> [ (<duration name>) ]
    is attached to <instance axis symbol>
    [ is attached to { <set-reset symbol> | <reset symbol2> | <timeout symbol3> } ]

<timer set area2> ::=
    <set-reset symbol> is associated with <timer name> [ (<duration name>) ]
    is attached to <instance axis symbol>
    is attached to <set symbol>
    [ is attached to { <reset symbol2> | <timeout symbol3> } ]

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

<set symbol1> ::=
    
```



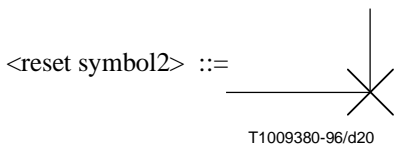
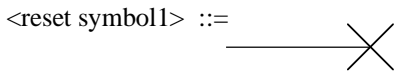
```

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

<timer reset area1> ::=
    <reset symbol1> is associated with <timer name> [ (<duration name>) ]
    is attached to <instance axis symbol>
    
```

<timer reset area2> ::=
 <reset symbol2> *is associated with* <timer name> [(<duration name>)]
 is attached to <instance axis symbol>
 is attached to { <set symbol> | <set-reset symbol> }

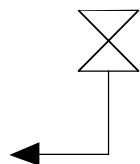
<reset symbol> ::= <reset symbol1> | <reset symbol2>



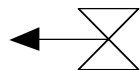
<timeout area> ::= <timeout area1> | <timeout area2>

<timeout area1> ::= <timeout symbol> *is associated with* <timer name> [(<duration name>)]
 is attached to <instance axis symbol>

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



<timeout symbol2> ::=



<timeout area2> ::=

<timeout symbol3>
 is attached to <instance axis symbol>
 is attached to { <set symbol> | <set-reset symbol> }

<timeout symbol3> ::=



Semantics

Set and reset are timer constructs taken over from SDL. Set denotes setting the timer and reset denotes resetting of the timer. Time-out corresponds to the consumption of the timer signal in SDL.

The setting of a timer always occurs before the corresponding reset or time-out.

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

4.8 Action

In addition to message exchange, the actions may be specified in MSCs. An informal text is attached to the actions.

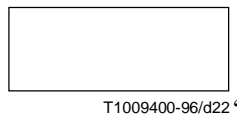
Concrete textual grammar

<action> ::= **action** <action character string>

Concrete graphical grammar

<action area> ::= <action symbol>
is attached to <instance axis symbol>
{ *is followed by* <general order area> }*
{ *is attached to* <general order area> }*
contains <action text>

<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 activity of an instance.

4.9 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> [(<parameter list>)]

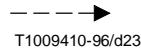
To each <create> there must be a corresponding instance with the specified name. The <instance name> has to refer to an instance with type process if its type is specified. An instance can be created only once, i.e. within one *simple* MSC, two or more <create>s with the same name must not appear.

Concrete graphical grammar

<create area> ::= <createline symbol> [*is associated with* <parameter list>]
is attached to { <instance axis symbol> <instance head area> } *set*
{ *is followed by* <general order area> }*
{ *is attached to* <general order area> }*

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. 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.

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

4.10 Instance stop

The instance stop in a sense is the counterpart to the instance creation. However, an instance can only stop itself whereas an instance is created by another instance.

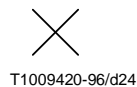
Concrete textual grammar

<stop> ::= **stop**

The <stop> at the end of an instance is allowed only for instances of type process if its type is specified.

Concrete graphical grammar

<stop symbol> ::=



Semantics

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

5 Structural concepts

In this clause, higher level concepts are introduced referring to generalized time ordering (coregion), composition and decomposition of instances, MSC reference concepts and composition of MSCs based on process algebra.

5.1 Coregion

The total ordering of events along each instance (see 4.1) in general may be not appropriate for entities referring to a higher level than SDL-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. A generalized ordering can be defined by means of general ordering relations.

Concrete textual grammar

<coregion> ::= **concurrent** <end> <coevent>* **endconcurrent** <coevent> ::=
<orderable event> <end>

Concrete graphical grammar

<concurrent area> ::=

<coregion symbol>
is attached to <instance axis symbol>
contains <coevent layer>

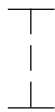
<coregion symbol> ::=

<coregion symbol1> | <coregion symbol2> <coevent layer> ::=
 <coevent area> | <coevent area> *above* <coevent layer>

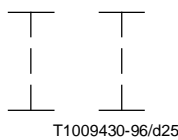
<coevent area> ::=

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

<coregion symbol1> ::=



<coregion symbol2> ::=



Drawing rule: The statement that the <coregion symbol> is attached to the <instance axis symbol> means that the <coregion symbol> should overlap the <instance axis symbol> as Figure 5-1:

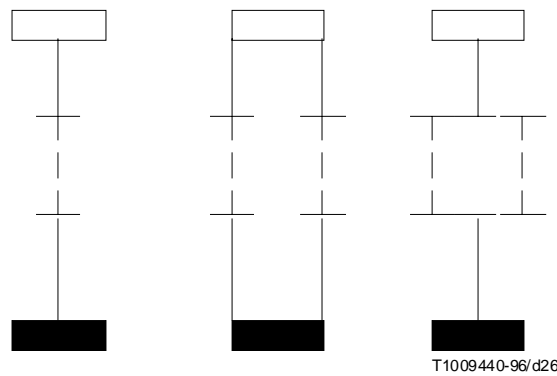


Figure 5-1/Z.120 – Different form of coregion

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

Semantics

For MSCs a total time 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 in time if no further synchronization constructs in form of general order relations are prescribed.

If a timer set is contained in a coregion together with the corresponding time-out or reset, then ordering relations between these events are preserved.

5.2 Instance decomposition

Instances in MSCs may refer to entities of different level of abstraction as indicated already by the keywords (**system**, **block**, **service**, **process**). Corresponding decomposition operations on instances can be defined determining the transition between different levels of abstraction. If instances are composed into one instance, then the total ordering of events along this instance may be relaxed in order to preserve the externally observable behaviour.

By means of the keyword **decomposed** and the optional <substructure reference>, a refining Message Sequence Chart may be attached to an instance.

Concrete textual grammar

```
<decomposition> ::=  
    decomposed [ <substructure reference> ]
```

```
<substructure reference> ::=  
    as <message sequence chart name>
```

To each instance containing the keyword **decomposed** a corresponding refining MSC with the specified name has to be specified. To each <message output> on a decomposed instance a corresponding <message input> sent to the exterior of the refining MSC must be specified. An analogous correspondence must hold for incoming messages.

Inline expressions and MSC-references must not be attached to decomposed instances.

Concrete graphical grammar

The graphical grammar is given in 4.2.

Semantics

By means of the keyword **decomposed**, a refining Message Sequence Chart may be attached to an instance. The name of the refining MSC may be defined explicitly by the optional <substructure reference>, otherwise it is identical with the name of the decomposed instance. The refining MSC represents a decomposition of this instance without affecting its observable behaviour. In the textual representation the messages addressed to and from the exterior of the refining MSC are characterized by the address **env**, in the graphical representation by the connection with the refining MSC border (frame symbol). Their connection with the external instances is provided by the messages sent and consumed by the decomposed instance, using message name identification. It must be possible to map the external behaviour of the refining MSC to the messages of the decomposed instance. The ordering of message events specified on a decomposed instance must be preserved in the refining MSC. Actions and conditions within the refining MSC may be looked at as a refinement of actions and conditions in the decomposed instance. Contrary to messages, however, no formal mapping to the decomposed instance is assumed, i.e. the refinement of actions and conditions need not obey formal rules.

5.3 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 composition, iteration, exception and optional regions.

Concrete textual grammar

```
<shared inline expr> ::=  
    { <shared loop expr> | <shared opt expr> | <shared alt expr> |  
    <shared par expr> | <shared exc expr> }
```

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

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

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

<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

<inline expr> ::=
 <loop expr> | <opt expr> | <alt expr> | <par expr> | <exc expr>

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

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

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

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

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

<loop boundary> ::=
 '< <inf natural> [, <inf natural>] >'

<inf natural> ::=
 inf | <natural name>+

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

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

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

Concrete graphical grammar

<inline expression area> ::=
 <loop area> | <opt area> | <par area> | <alt area> | <exc area>

<loop area> ::=
 <inline expression symbol> *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> *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> *contains*
 { **exc** <operand area> }
is attached to { <instance axis symbol>* } *set*
is attached to { <inline gate area>* | <inline order gate area>* } *set*

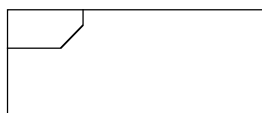
<par area> ::=
 <inline expression symbol> *contains*
 { **par** <operand area>
 { *is followed by* <separator area> *is followed by* <operand area> }* }
is attached to { <instance axis symbol>* } *set*
is attached to { <inline gate area>* | <inline order gate area>* } *set*

<alt area> ::=
 <inline expression symbol> *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> }*

<separator area> ::=
 <separator symbol>

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

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

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

Inline expressions must not be attached to decomposed instances.

Semantics

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

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.

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.

The **loop** construct can have several forms. The most basic form is "**loop** <n,m>" where n and m are natural numbers. This means that the operand may be executed at least n times and at most m times. The naturals 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 **opt** operator is the same as an alternative where the second operand is the empty MSC.

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.

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.

5.4 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**, **exc** and **subst**, and MSC references.

The **alt**, **par**, **loop**, **opt** and **exc** operators are described in 5.3. The **seq** operator denotes the weak sequencing operation where only events on the same instance are ordered.

The **subst** operation is a substitution of concepts inside the referenced MSC. Message names are substituted by message names, instance names by instance names and MSC names by MSC names.

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.

Concrete textual grammar

<shared msc reference> ::=
 reference [<msc reference identification>:] <msc ref expr>
 <shared> [<reference gate interface>]

<msc reference> ::=
 reference [<msc reference identification>:] <msc ref expr>
 [<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 exc expr> { **seq** <msc ref exc expr> }*

<msc ref exc expr> ::=
 [**exc**] <msc ref opt expr>

<msc ref opt expr> ::=
 [**opt**] <msc ref loop expr>

<msc ref loop expr> ::=
 [**loop**] [<loop boundary>]
 { **empty** | <msc name> [<parameter substitution>] | (<msc ref expr>) }

<parameter substitution> ::=
 subst <substitution list>

<substitution list> ::=
 <substitution> [, <substitution list>]

<substitution> ::=
 <replace message> | <replace instance> | <replace msc>

<replace message> ::=
 [**msg**] <message name> **by** <message name>

<replace instance> ::=
 [**inst**] <instance name> **by** <instance name>

<replace msc> ::=
 [**msc**] { **empty** | <msc name> } **by** { **empty** | <msc name> }

<reference gate interface> ::=
 { <end> **gate** <ref gate> }*

<ref gate> ::=
 <actual out gate> | <actual in gate> | <actual order out gate> | <actual order in gate>

If there are ambiguities regarding names in substitutions, the optional keywords **msg**, **inst** and **msc** should be used to eliminate the ambiguity.

The substitution of an MSC name must share the same gate interface as what it replaces.

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.

An msc reference may be attached to instances which are not contained in the referenced 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 refers to 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).

Concrete graphical grammar

<msc reference area>::=

```

    <msc reference symbol>
    contains { <msc ref expr> [ <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.

MSC references must not be attached to decomposed instances.

MSC references must not directly or indirectly refer to their enclosing MSC (recursion).

An MSC containing references and substitutions is illegal if, after repeatedly expanding the references and applying the substitutions, 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 message-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.

In general, the MSC reference may refer to a textual MSC expression possibly modified by substitution. In the simple case, where the MSC expression consists of an MSC name, 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. The result of a reference with a substitution is that the referenced MSC definition will be modified by the replacements of the substitution. **Subst x by y** denotes that x is replaced by y. All substitutions in a <substitution list> are thought to be applied in parallel. Thus, the order in which the substitutions take place is not relevant.

If an MSC definition, to which substitution is applied, contains MSC references, then the substitution should be applied also to the MSC definitions corresponding to these MSC references.

In the textual representation, the optional <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.

5.5 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
- a parallel 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 upper edge of the node symbols whereas the outgoing flow lines are connected to the lower 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 and impose restrictions on the MSCs that are referenced in the HMSC.

The parallel frames contain one or more small HMSCs and indicate that the small HMSCs are the operands of a parallel operator, i.e. the events in the different small HMSCs can be interleaved.

The connection points are introduced to simplify the layout of HMSCs and have no semantical meaning.

Concrete textual grammar

```
<msc expression> ::=
    <start> <node expression> *

<start> ::=
    <label name> { alt <label name> } * <end>

<node expression> ::=
    <label name> : { <node> seq (<label name> { alt <label name> } * ) | end } <end>

<node> ::=
    (<msc ref expr>)
    | empty | <msc name>
    | <par expression>
    | condition <condition name list>
    | connect

<par expression> ::=
    expr <msc expression> endexpr { par expr <msc expression> endexpr } *
```

Static Semantics

All referenced <label name>s in <start> or <node expression> must be defined in the HMSC, i.e. occur as "<label name>:" in a <node expression>. Every node in the HMSC graph must be reachable from the <start>, i.e. the graph must be connected.

Concrete graphical grammar

```
<mscexpr area> ::=
    { <start area> <node expression area> * <hmsc end area> * } set

<start area> ::=
    <hmsc start symbol> is followed by { <alt op area> + } set
```

<hmsc start symbol> ::=



<hmsc end area> ::=

<hmsc end symbol> *is attached to* { <hmsc line symbol>+ } *set*

<hmsc end symbol> ::=



<hmsc line symbol> ::=

<hmsc line symbol1> | <hmsc line symbol2>

<hmsc line symbol1> ::=



<hmsc line symbol2> ::=



<alt op area> ::=

<hmsc line symbol> *is attached to* { <node area> | <hmsc end symbol> }

<node expression area> ::=

<node area> *is followed by* { <alt op area>+ } *set*
is attached to { <hmsc line symbol>+ } *set*

<node area> ::=

<hmsc reference area>
| <connection point symbol>
| <hmsc condition area>
| <par expr area>

<hmsc reference area> ::=

<msc reference symbol> *contains* <msc ref expr>

<connection point symbol> ::=



T1009470-96/d29

<hmsc condition area> ::=

<condition symbol> *contains* <condition name list>

<par expr area> ::=

<par frame symbol> *contains* { <mscexpr area>+ } *set*

<par frame symbol> ::=

<frame symbol>

<hmsc end area>

Drawing rule

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 should be attached to the lower corner of the <hmsc start symbol> as illustrated in Figure 5-2a where two lines <hmsc line symbol>s follow an <hmsc start symbol>:

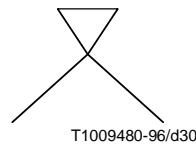


Figure 5-2a/Z.120 – HMSC drawing rules

That the <hmsc line symbol> is attached to another symbol in the <alt op area> production rule means that the <hmsc line symbol>s should 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 Figure 5-2b.

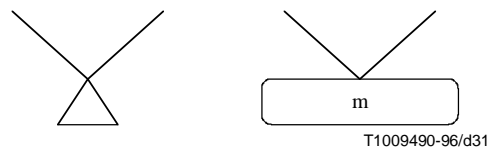


Figure 5-2b/Z.120 – HMSC drawing rules

That the <node area> is followed by an <alt op area> in the <node expression area> production rule means that the <hmsc line symbol>s should be attached to the lower edge of the symbol in the <node area> as illustrated in Figure 5-2c that shows how an <msc reference symbol> is followed by an <alt op area>:

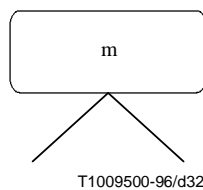


Figure 5-2c/Z.120 – HMSC drawing rules

Static Semantics

The composition of MSCs described using HMSCs can be guarded by conditions in the HMSCs. These conditions can be used to indicate global system states. Note that only global conditions can be used to restrict the composition of MSCs. All conditions in HMSCs are considered to be global conditions. In a simple MSC (an MSC described using instances, messages, etc.) only conditions that are shared by all instances in the MSC are considered global and can be used to restrict the composition in HMSCs. Non-global conditions in simple MSCs are considered as comments.

Each simple MSC, HMSC, parallel frame (in HMSC) and MSC expression (in MSC reference) has a set of initial conditions and a set of final conditions according to the following rules:

The set of initial conditions for a simple MSC (i.e. an MSC not described by an HMSC diagram) is defined as follows:

- If the MSC contains a global condition that precedes everything except the instance heads, then the set of initial conditions is defined to be the <condition name>s that are defined by this condition.
- If the MSC has no such global condition then the set of initial conditions is defined to be the set of all possible <condition name>s. This also applies to the 'empty' MSC.

The set of final conditions for a simple MSC is defined as follows:

- If the MSC contains a global condition that is not followed by anything except the instance ends, then the set of final conditions is defined to be the <condition name>s that are defined by this condition.
- If the MSC has no such global condition then the set of final conditions is defined to be the set of all possible <condition name>s. This also applies to the 'empty' MSC.

The set of initial conditions of an HMSC is defined as follows:

- If the <hmsc start symbol> is immediately followed by one or more <condition symbol>s, then the set of initial conditions is defined as the intersection of the sets of <condition name>s defined by each of the <condition symbol>s. Note that if an outgoing branch from the <start symbol> starts with more than one <condition symbol>, then only the first is considered when computing the initial conditions of the HMSC.
- If the <start symbol> is not immediately followed by a <condition symbol>, then the set of initial conditions is defined to be the complete set of all possible <condition name>s.

The set of final conditions of an HMSC is defined as follows:

- If the HMSC contains one or more <hmsc end symbol>s that are immediately preceded by <condition symbol>s, then the set of final conditions is defined as the intersection of the sets of <condition name>s defined by each of the <condition symbol>s.
- If none of the <hmsc end symbol>s in the HMSC is immediately preceded by a <condition symbol>, then the set of final conditions is defined to be the complete set of all possible <condition name>s.

The set of initial conditions of an <msc ref expression> (i.e. the expression associated with an <msc reference>) is defined as follows (where m1 is the name of an MSC and e1 and e2 are MSC expressions):

- The set of initial conditions for the expression 'm1' is the set of initial conditions of the MSC m1.
- The set of initial conditions for 'e1 seq e2' is the set of initial conditions for e1.
- The set of initial conditions for 'e1 alt e2' is the intersection of the sets of initial conditions for e1 and e2.
- The set of initial conditions for 'e1 par e2' is the intersection of the sets of initial conditions for e1 and e2.
- The set of initial conditions for 'opt e1' is the set of initial conditions for e1.
- The set of initial conditions for 'loop e1' is the set of initial conditions for e1.

The set of final conditions of an MSC expression in an MSC reference is defined as follows (where m1 is the name of an MSC and e1 and e2 are MSC expressions):

- The set of final conditions for the expression 'm1' is the set of final conditions of the MSC m1.
- The set of final conditions for 'e1 seq e2' is the set of final conditions for e2.
- The set of final conditions for 'e1 alt e2' is the intersection of the sets of final conditions for e1 and e2.
- The set of final conditions for 'e1 par e2' is the intersection of the sets of final conditions for e1 and e2.
- The set of final conditions for 'opt e1' is the set of final conditions for e1.
- The set of final conditions for 'loop e1' is the set of final conditions for e1.

The initial and final conditions for <par expr area> in HMSCs are defined in the same way as for **par** expressions:

- The set of initial conditions for the <par expr area> is the intersection of the set of initial conditions for the operands.
- The set of final conditions for the <par expr area> is the intersection of the set of final conditions for the operands.

Four static restrictions are related to conditions in HMSCs:

- If an <msc reference> is immediately preceded by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of initial conditions of the <msc ref expression> associated with the <msc reference>.
- If an <msc reference> is immediately followed by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of final conditions of the <msc ref expression> associated with the <msc reference>.

- If a <par expr area> is immediately preceded by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of initial conditions of the <par expr area>.
- If a <par expr area> is immediately followed by a <condition symbol>, with an associated set of <condition name>s, then this set must be a subset of the set of final conditions of the <par expr area>.

Semantics

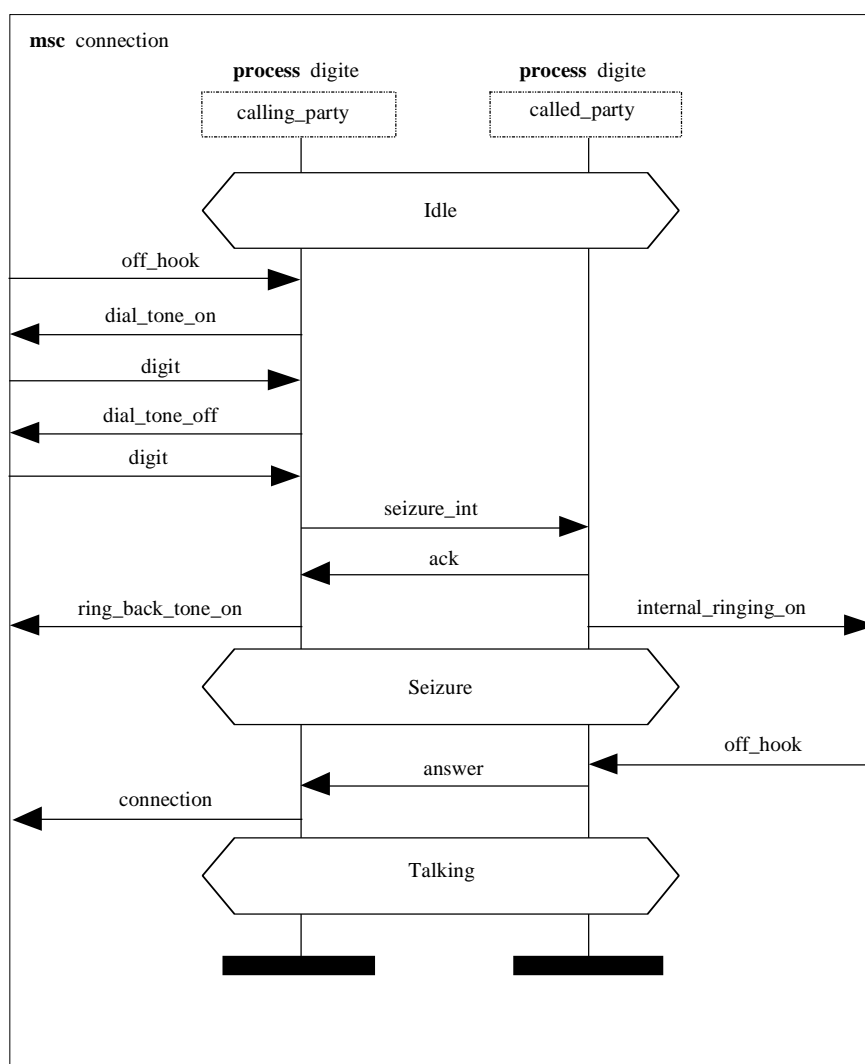
The graph describing the composition of MSCs within an HMSC is interpreted in an operational way as follows. Execution starts at the <hmsc start symbol>. Next, it continues with a node that follows one of the outgoing edges of this symbol. These nodes are considered to be operands of an **alt** operator (see 5.3). 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 5.4. Execution of a condition or a connection point is an empty operation. Execution of a parallel frame consists of executing the operands of the parallel frame in parallel, as described in 5.3 for the **par** operator. A sequential execution of two nodes that are related by an edge is described by the **seq** operator (see 5.4).

6 Message Sequence Chart examples

This clause is informative rather than normative.

6.1 Standard message flow diagram

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.



T1009510-96/d33

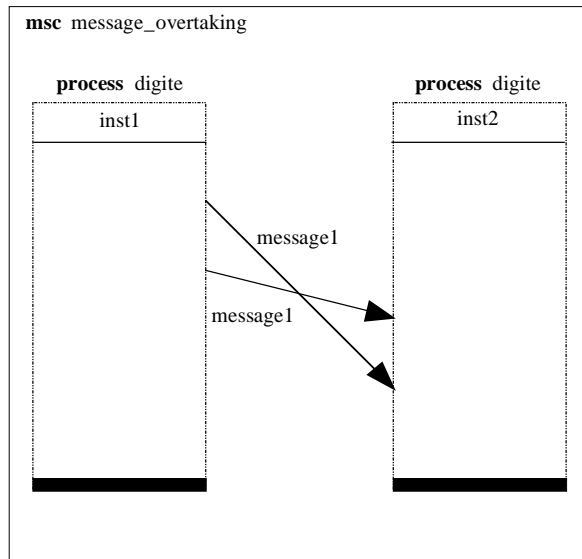
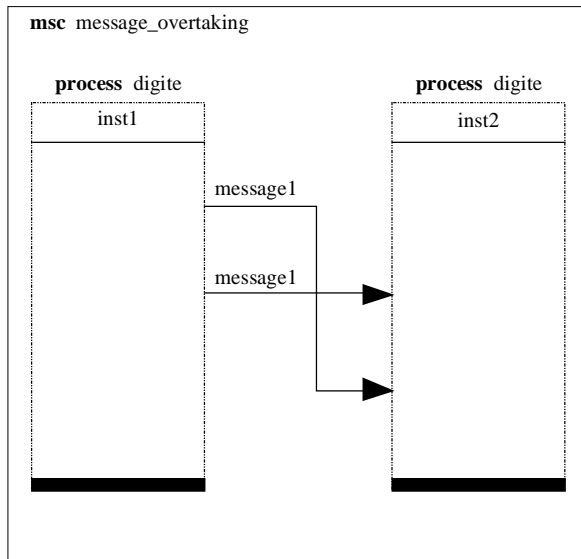
```

msc connection; inst calling_party: process digite, called_party: process digite;
  instance calling_party: process digite;
    condition 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;
  instance called_party: process digite;
    condition 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;

```

6.2 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.



T1009520-96/d34

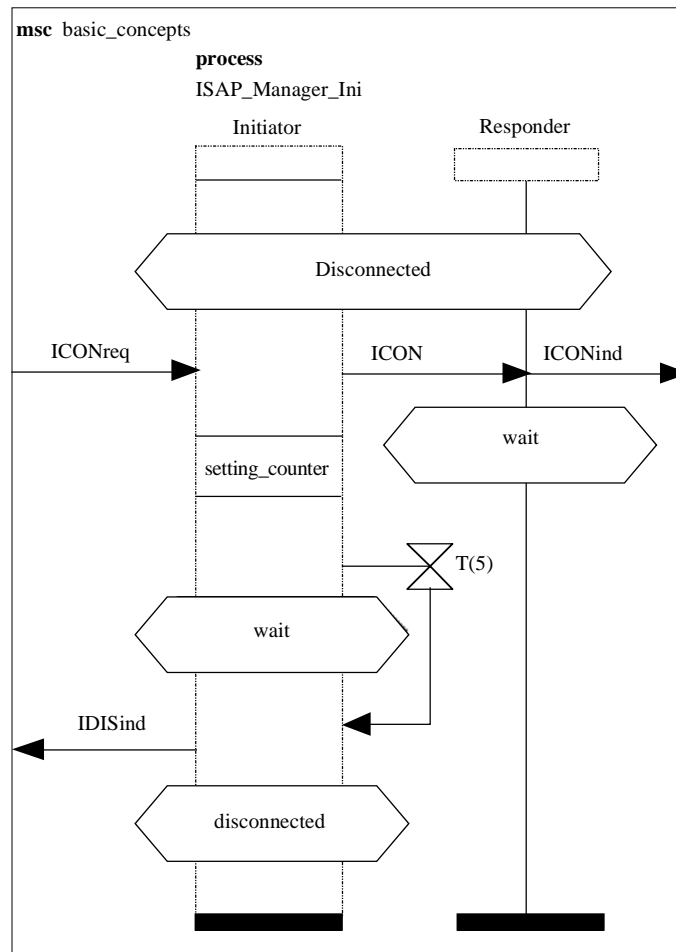
```

msc message_overtaking; inst inst1, inst2;
  instance inst1: process digite;
    out message1, a to inst2;
    out message1, b to inst2;
  endinstance;
  instance inst2: process digite;
    in message1, b from inst1;
    in message1, a from inst1;
  endinstance;
endmsc;

```

6.3 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.



T1009530-96/d35

Instance-oriented textual syntax

```

mhc basic_concepts; inst Initiator: process ISAP_Manager_Ini, Responder;
    instance Initiator: process ISAP_Manager_Ini;
        condition Disconnected shared all;
        in ICONreq from env;
        out ICON to Responder;
        action setting_counter;
        set T(5);
        condition wait shared;
        timeout T;
        out IDISind to env;
        condition disconnected shared;
    endinstance;
    instance Responder;
        condition Disconnected shared all;
        in ICON from Initiator;
        out ICONind to env;
        condition wait shared;
    endinstance;
endmhc;

```

Event-oriented textual syntax

```

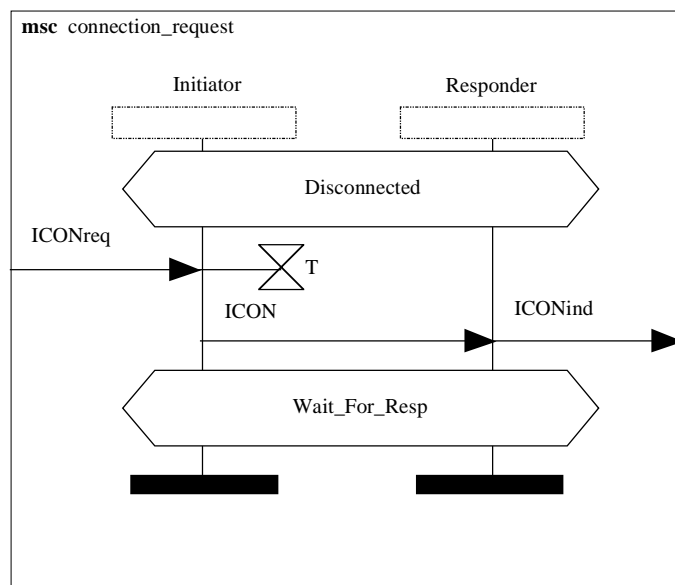
msc basic_concepts; inst Initiator: process ISAP_Manager_Ini, Responder;
Initiator: instance process ISAP_Manager_Ini;
Responder: instance;
all: condition Disconnected;
Initiator: in ICONreq from env;
           out ICON to Responder;
Responder: in ICON from Initiator;
           out ICONind to env;
           condition wait;
           endinstance;
Initiator: action setting_counter;
           set T(5);
           condition wait;
           timeout T;
           out IDISind to env;
           condition disconnected;
           endinstance;
endmsc;

```

6.4 MSC-composition/MSC-decomposition

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 (see example in 6.5).

The composition is defined by means of the HMSC 'con_setup'.

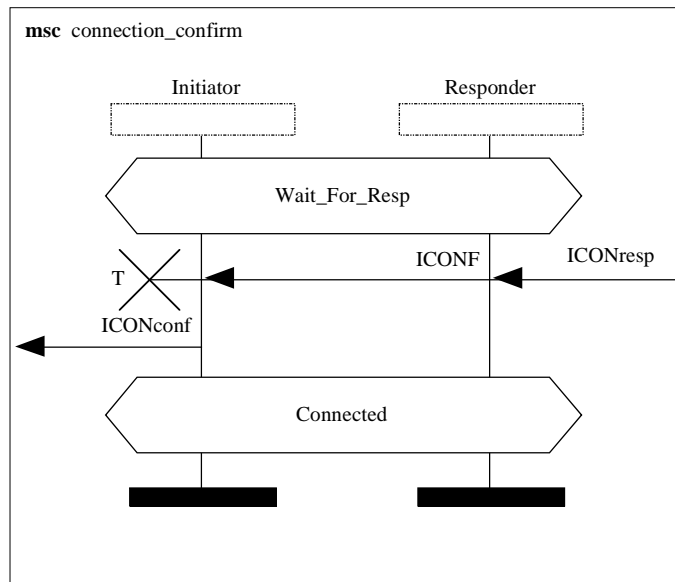


T1009540-96/d36

```

msc connection_request; inst Initiator, Responder;
           instance Initiator;
           condition Disconnected shared all;
           in ICONreq from env;
           set T;
           out ICON to Responder;
           condition Wait_For_Resp shared all;
           endinstance;
           instance Responder;
           condition Disconnected shared all;
           in ICON from Initiator;
           out ICONind to env;
           condition Wait_For_Resp shared all;
           endinstance;
endmsc;

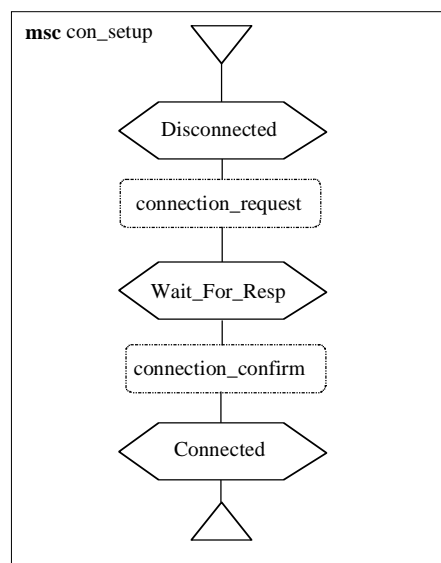
```



T1009550-96/d37

```

msc connection_confirm; inst Initiator, Responder;
instance Initiator;
condition Wait_For_Resp shared all;
in ICONF from Responder;
reset T;
out ICONconf to env;
condition Connected shared all;
endinstance;
instance Responder;
condition Wait_For_Resp shared all;
in ICONresp from env;
out ICONF to Initiator;
condition Connected shared all;
endinstance;
endmsc;
  
```



T1009560-96/d38

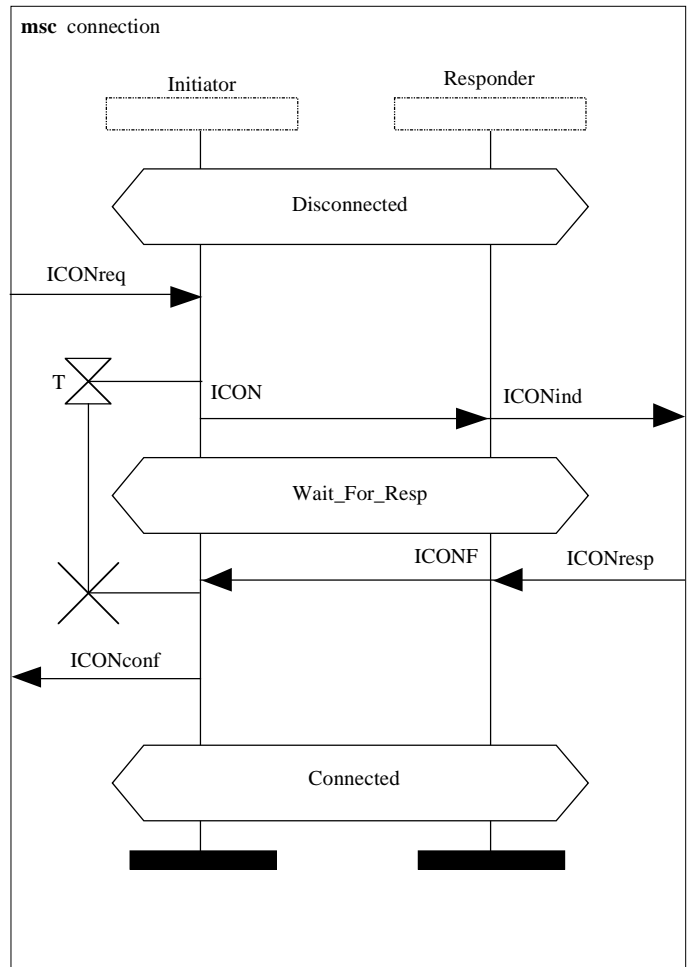
```

mhc con_setup;
expr L1;
    L1: condition Disconnected seq (L2);
    L2: connection_request seq L3;
    L3: condition Wait_For_Resp seq (L4);
    L4: connection_confirm seq (L5);
    L5: condition Connected seq (L6);
    L6: end;
endmhc;

```

6.5 MSC with time supervision

The MSC 'connection' in this example contains a timer reset.



T1009570-96/d39

```

mhc connection; inst Initiator, Responder;
instance Initiator;
    condition Disconnected shared all;
    in ICONreq from env;
    set T;
    out ICON to Responder;
    condition Wait_For_Resp shared all;
    in ICONF from Responder;
    reset T;
    out ICONconf to env;
    condition Connected shared all;

```

```

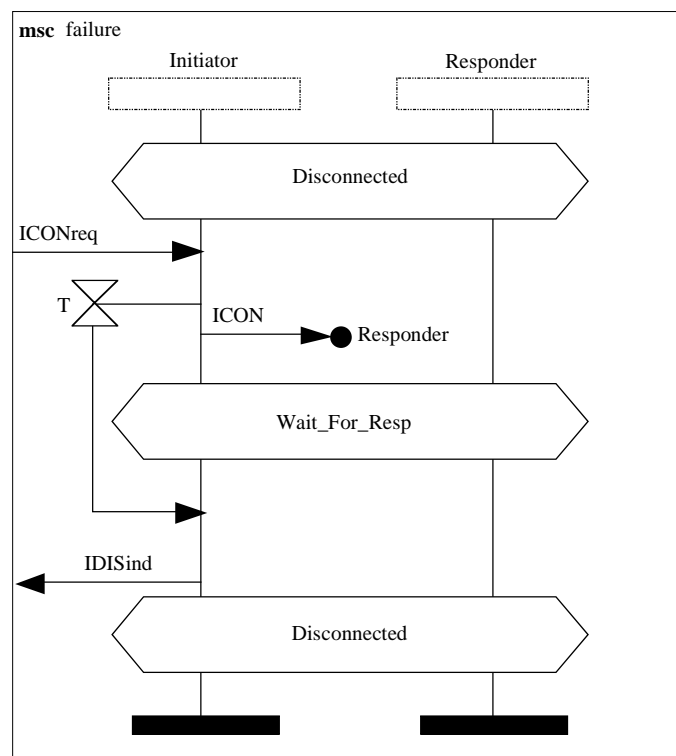
endinstance;
instance Responder;
  condition Disconnected shared all;
  in ICON from Initiator;
  out ICONind to env;
  condition Wait_For_Resp shared all;
  in ICONresp from env;
  out ICONF to Initiator;
  condition Connected shared all;
endinstance;

```

endmsc;

6.6 MSC with message loss

The MSC 'failure' in this example contains a timer expiration due to a lost message.



T1009580-96/d40

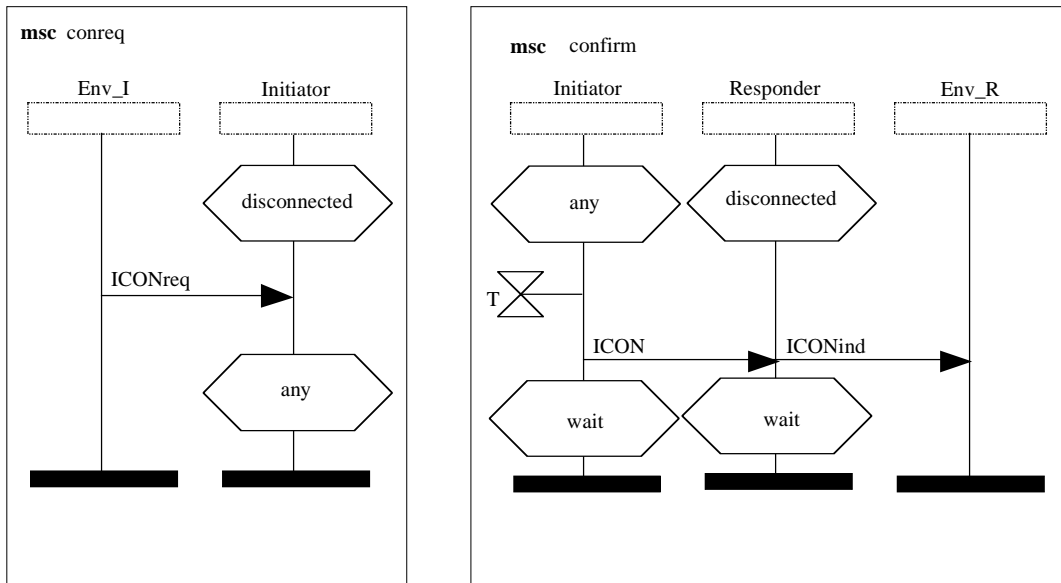
```

msc failure; inst Initiator, Responder;
  instance Initiator;
    condition Disconnected shared all;
    in ICONreq from env;
    set T;
    out ICON to lost Responder;
    condition Wait_For_Resp shared all;
    timeout T;
    out IDISind to env;
    condition Disconnected shared all;
  endinstance;
  instance Responder;
    condition Disconnected shared all;
    in ICON from Initiator;
    condition Wait_For_Resp shared all;
    condition Disconnected shared all;
  endinstance;
endmsc;

```

6.7 Local conditions

In this example local conditions referring to one instance are employed to indicate local states of this instance.



T1009590-96/d41

```

mhc conreq; inst Env_I, Initiator;
  instance Env_I;
    out ICONreq to Initiator;
  endinstance;
  instance Initiator;
    condition disconnected shared;
    in ICONreq from Env_I;
    condition any shared;
  endinstance;
endmhc;

```

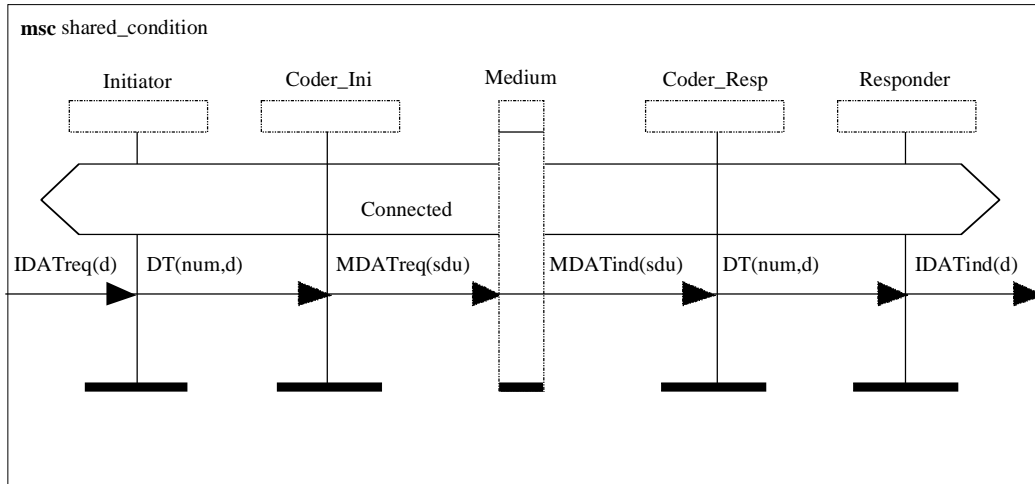
```

mhc confirm; inst Initiator, Responder, Env_R;
  instance Initiator;
    condition any shared;
    set T;
    out ICON to Responder;
    condition wait shared;
  endinstance;
  instance Responder;
    condition disconnected;
    in ICON from Initiator;
    out ICONind to Env_R;
    condition wait shared;
  endinstance;
  instance Env_R;
    in ICONind from Responder;
  endinstance;
endmhc;

```


6.8 Shared condition and messages with parameters

This example contains the shared condition 'connected'. This condition is shared by the instances 'Initiator' and 'Responder'. The instances 'Coder_Ini', 'Medium', 'Coder_Resp' are not involved. In the textual representation the keyword **shared** together with a list of instances indicates the instances to which the condition is attached.



T1009600-96/d42

```
mhc shared_condition; inst Initiator, Coder_Ini, Medium, Coder_Resp, Responder;
```

```
instance Initiator;
```

```
condition connected shared Responder;
```

```
in IDATreq(d) from env;
```

```
out DT(num,d) to Coder_Ini;
```

```
endinstance;
```

```
instance Coder_Ini;
```

```
in DT(num,d) from Initiator;
```

```
out MDATreq(sdu) to Medium;
```

```
endinstance;
```

```
instance Medium;
```

```
in MDATreq(sdu) from Coder_Ini;
```

```
out MDATind(sdu) to Coder_Resp;
```

```
endinstance;
```

```
instance Coder_Resp;
```

```
in MDATind(sdu) from Medium;
```

```
out DT(num,d) to Responder;
```

```
endinstance;
```

```
instance Responder;
```

```
condition connected shared Initiator;
```

```
in DT(num,d) from Coder_Resp;
```

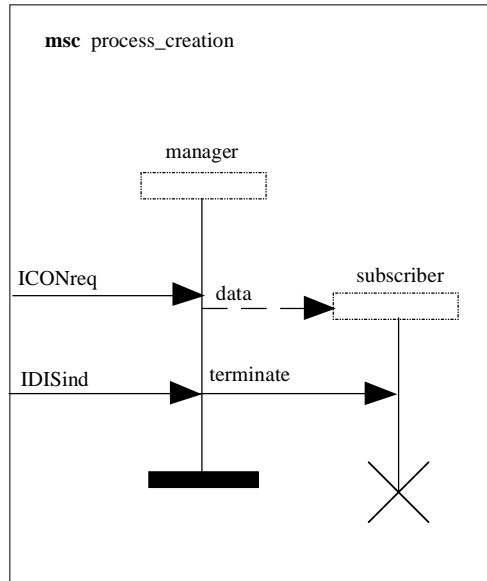
```
out IDATind(d) to env;
```

```
endinstance;
```

```
endmhc;
```

6.9 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.



T1009610-96/d43

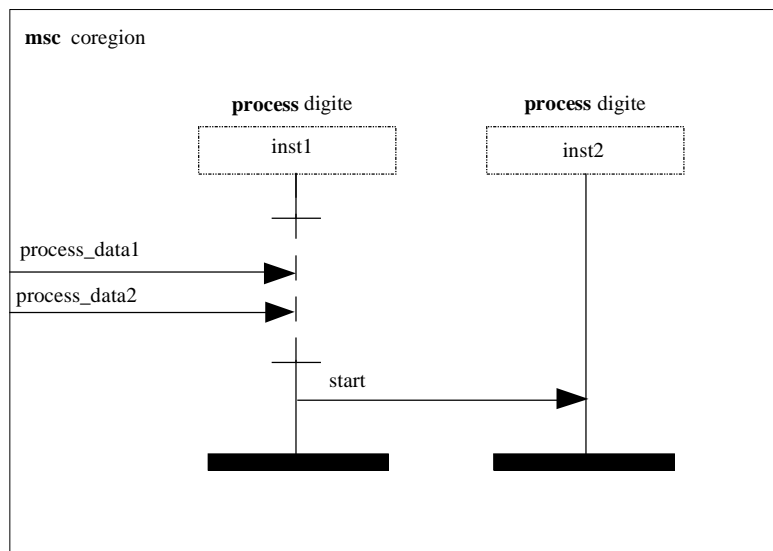
```

msc process_creation; inst manager, subscriber;
    instance manager;
        in ICONreq from env;
        create subscriber(data);
        in IDISind from env;
        out terminate to subscriber;
    endinstance;
    instance subscriber;
        in terminate from manager;
        stop;
    endinstance;
endmsc;

```

6.10 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.



T1009620-96/d44

```

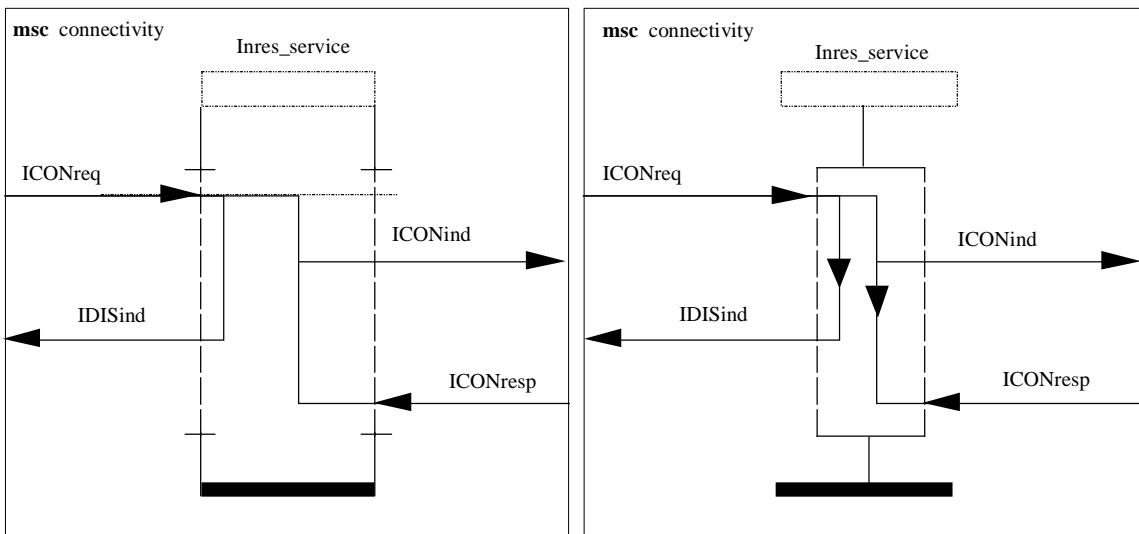
msc coregion; inst inst1, inst2;
    instance inst1: process digite;
        concurrent;
            in process_data1 from env;
            in process_data2 from env;
        endconcurrent;
        out start to inst2;
    endinstance;
    instance inst2: process digite;
        in start from inst1;
    endinstance;
endmsc;

```

6.11 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 $ICONresp$.



T1009630-96/d45

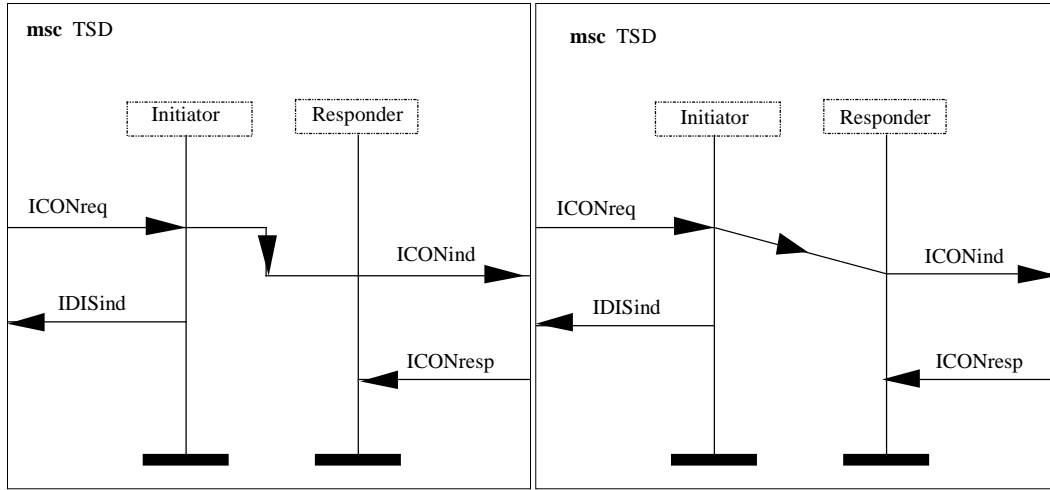
```

msc connectivity; inst Inres_service;
    instance Inres_service;
        concurrent;
            in ICONreq from env before Label1, Label2;
            Label1 out ICONind to env before Label3;
            Label2 out IDISind to env;
            Label3 in ICONresp from env;
        endconcurrent;
    endinstance;
endmsc;

```

6.12 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 $ICONreq < ICONind$.



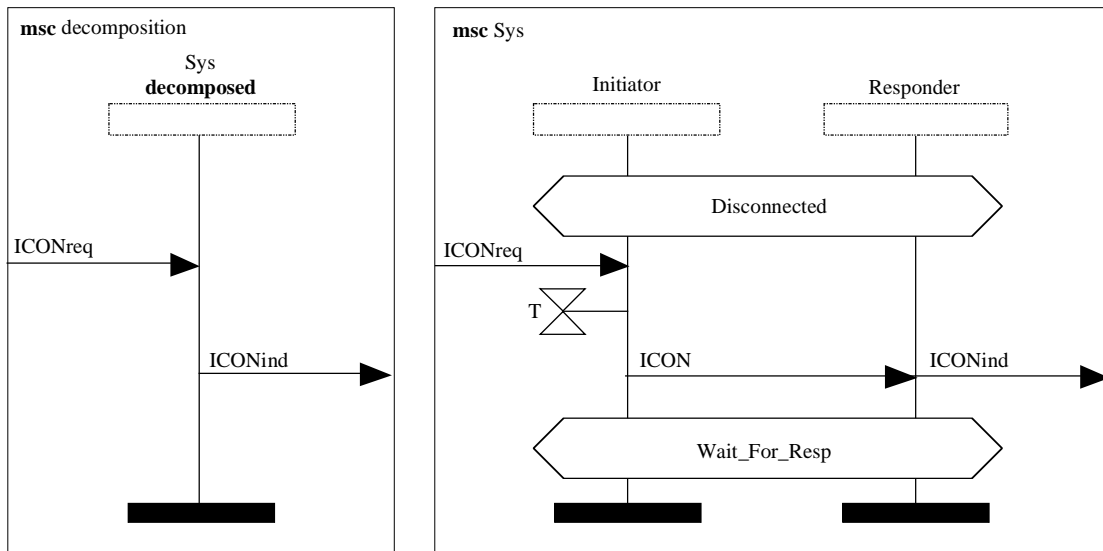
T1009640-96/d46

```

msc TSD; inst Initiator, Responder;
instance Initiator;
  in ICONreq from env before Label1;
  out IDISind to env;
endinstance;
instance Responder;
  Label1 out ICONind to env;
  in ICONresp from env;
endinstance;
endmsc;
  
```

6.13 Instance decomposition

This example contains the refining MSC 'Sys'. This MSC is attached to the instance 'Sys' thus representing a decomposition of this instance.



T1009650-96/d47

```

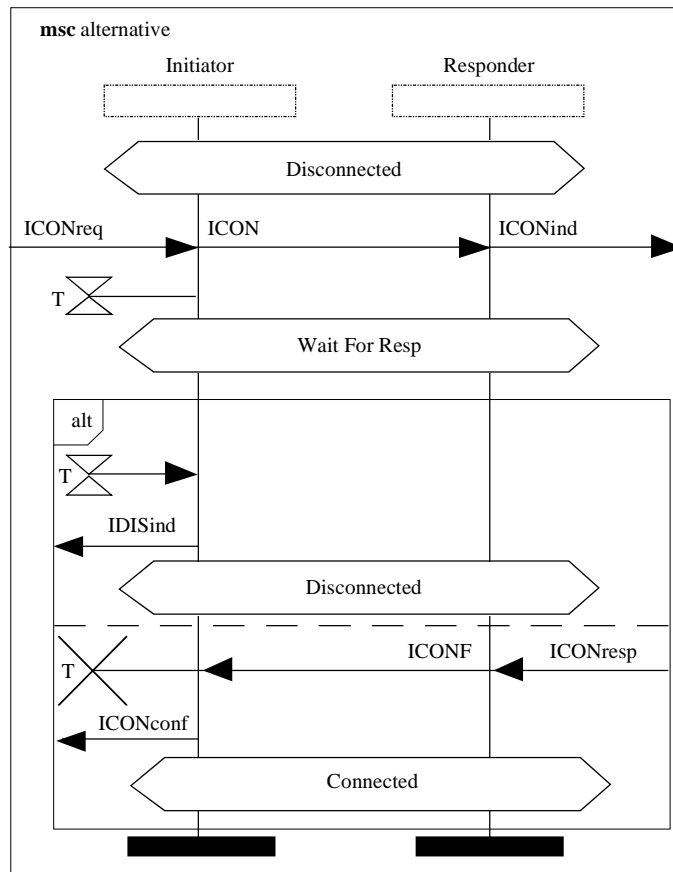
mhc decomposition; inst Sys;
  instance Sys decomposed;
    in ICONreq from env;
    out ICONind to env;
  endinstance;
endmhc;

mhc Sys; inst Initiator, Responder;
  instance Initiator;
    condition Disconnected shared all;
    in ICONreq from env;
    set T;
    out ICON to Responder;
    condition Wait_For_Resp shared all;
  endinstance;
  instance Responder;
    condition Disconnected shared all;
    in ICON from Initiator;
    out ICONind to env;
    condition Wait_For_Resp shared all;
  endinstance;
endmhc;

```

6.14 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.



T1009660-96/d48

```

msc alternative; inst Initiator, Responder;
Initiator:      instance;
Responder:     instance;
all:         condition Disconnected;
Initiator:     in ICONreq from env;
               out ICON to Responder;
               set 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;
               reset T;
               out ICONconf to env;

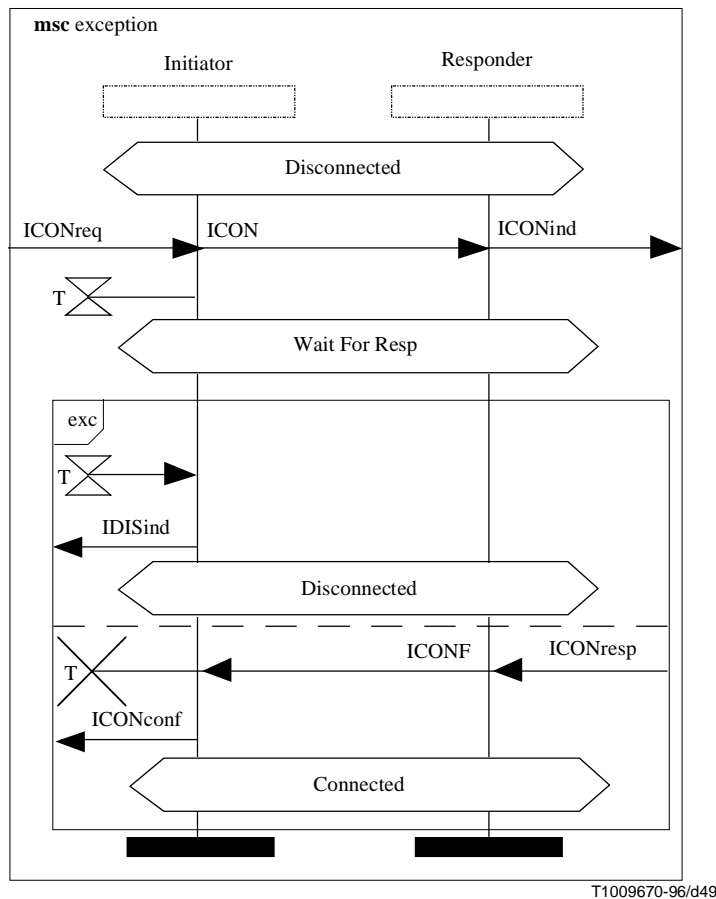
all:         condition Connected;
               alt end;

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':



```

msc exception; inst Initiator, Responder;
Initiator:      instance;
Responder:      instance;
all:           condition Disconnected;
Initiator:      in ICONreq from env;
                out ICON to Responder;
                set 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;
                reset T;
                out ICONconf to env;

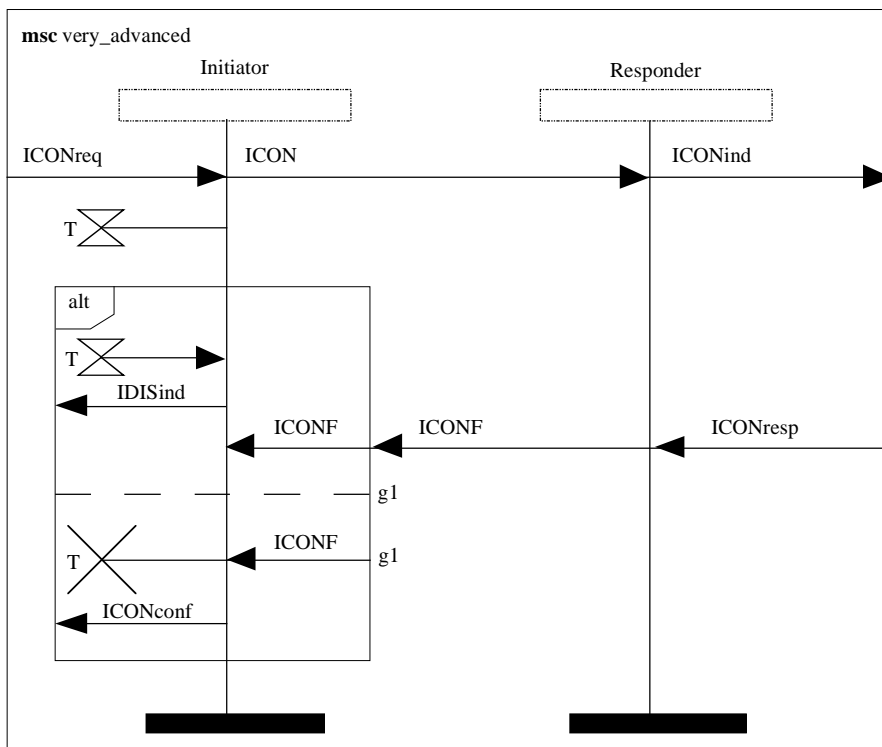
all:           condition Connected;
Initiator:      endinstance;
Responder:      endinstance;
endmsc;

```

6.15 Inline Expression with gates

This example describes the scenario of 6.14 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':

- the 'Responder' is not answering in time: the message input 'ICONF' is discarded after time-out and disconnection of 'Initiator';
- the 'Responder' is answering in time which leads to a successful connection.



T1009680-96/d50

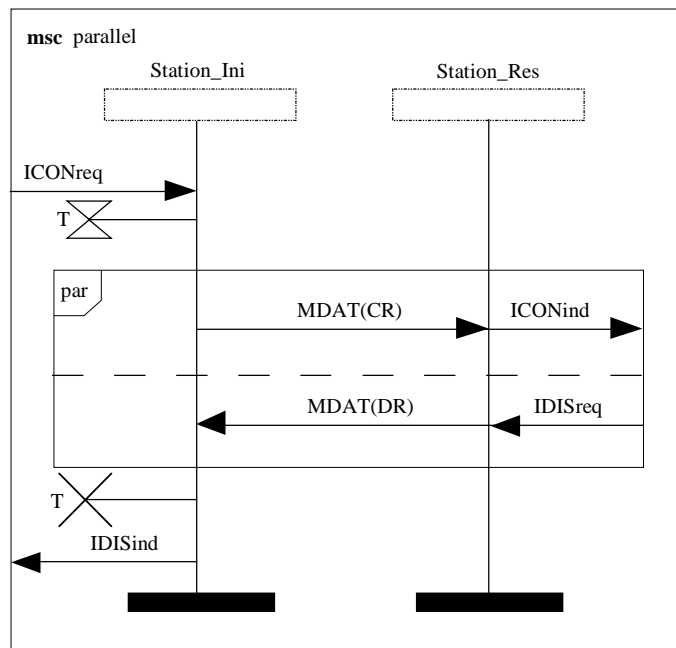
```

msc very_advanced; inst Initiator, Responder; gate out ICONreq to Initiator;
gate in ICONind from Responder; gate out ICONresp to Responder;
    Initiator:      instance;
    Responder:     instance;
    Initiator:     in ICONreq from env;
                  out ICON to Responder;
                  set 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 in IDISind from Initiator;
                  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;
                  gate in ICONconf from Initiator;
                  in ICONF from env via g1;
                  reset T;
                  out ICONconf to env;
    alt end;
    Initiator:    endinstance;
    Responder:    endinstance;
endmsc;

```

6.16 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)'.



T1009690-96/d51

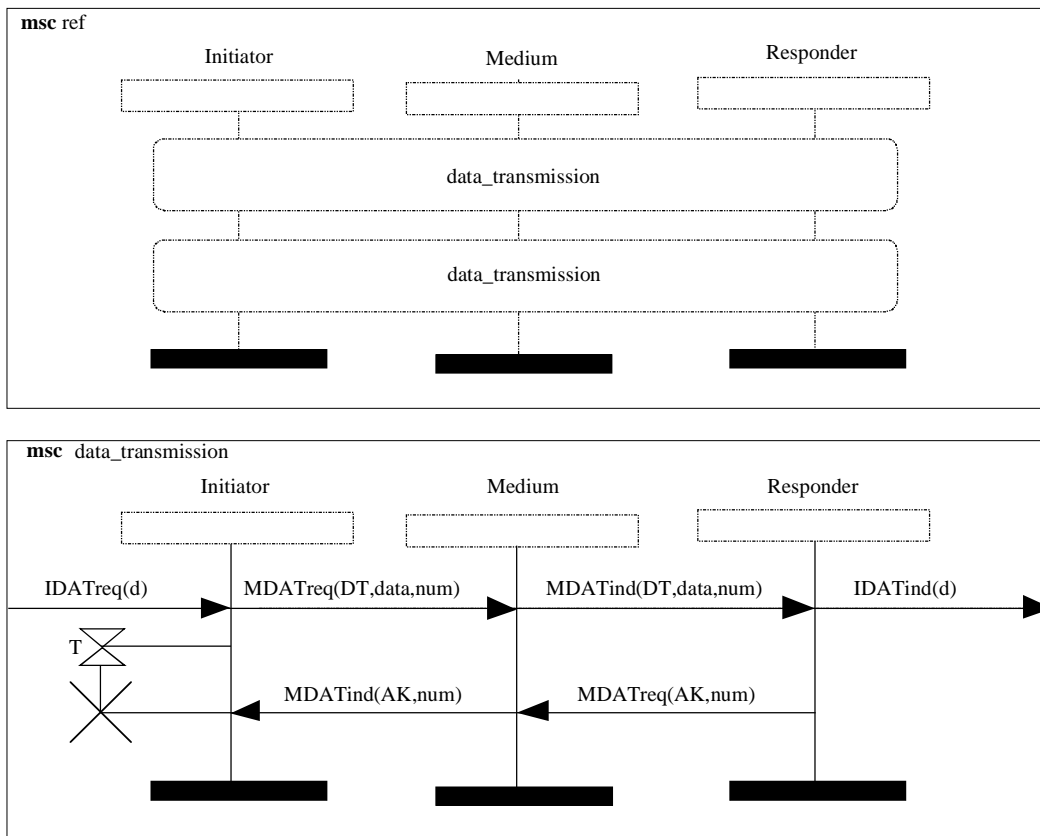

```

msc parallel; inst Station_Ini, Station_Res;
  Station_Ini:      instance;
                   in ICONreq from env;
                   set T;
  Station_Res:      instance;
  all:              par begin;
    Station_Ini:    out MDAT(CR) to Station_Res;
    Station_Res:    in MDAT(CR) from Station_Ini;
                   out ICONind to env;
                   par;
    Station_Res:    in IDISreq from env;
                   out MDAT(DR) to Station_Ini;
    Station_Ini:    in MDAT(DR) from Station_Res;
                   par end;
  Station_Res:      endinstance;
  Station_Ini:      reset T;
                   out IDISind to env;
                   endinstance;
endmsc;

```

6.17 MSC reference

Within this example the MSC references 'data_transmission' are employed to denote two successful data transmissions referring to the same MSC definition.



T1009700-96/d52

```

msc ref; inst Initiator, Medium, Responder;
  Initiator:      instance;
  Medium:         instance;
  Responder:      instance;
  all:            reference data_transmission;
                 reference data_transmission;
  Initiator:      endinstance;
  Medium:         endinstance;
  Responder:      endinstance;
endmsc;

```

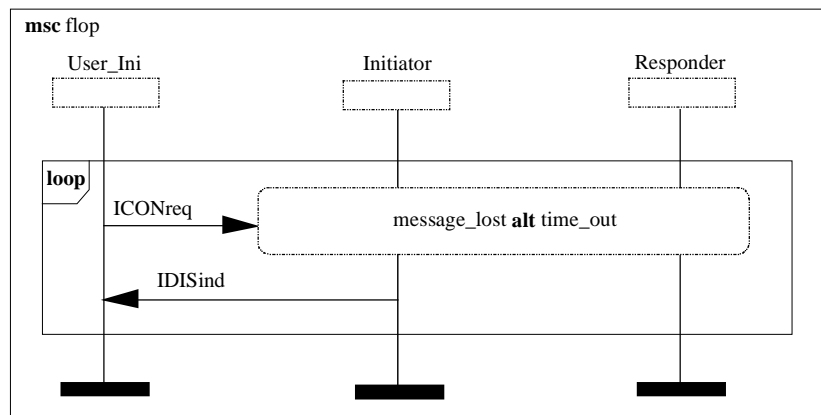
```

msc data_transmission; inst Initiator, Medium, Responder;
    Initiator:      instance;
    Medium:         instance;
    Responder:     instance;
    Initiator:     in IDATreq(d) from env;
                  out MDATreq(DT,data,num) to Medium;
                  set 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;
                  reset T;
    Initiator:    endinstance;
    Medium:       endinstance;
    Responder:    endinstance;
endmsc;

```

6.18 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 6.21.



T1009710-96/d53

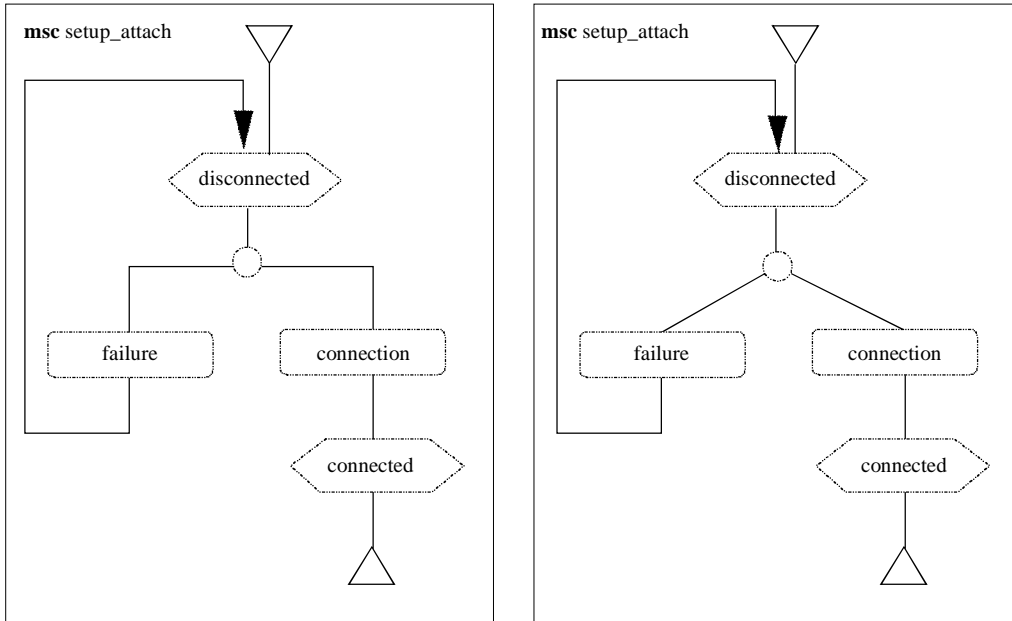
```

msc flop; inst User_Ini, Initiator, Responder;
    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;
        Initiator:     out IDISind to User_Ini;
        User_Ini:      in IDISind from Initiator;
    loop end;
    User_Ini:      endinstance;
    Initiator:     endinstance;
    Responder:    endinstance;
endmsc;

```

6.19 High-level MSC with free loop

MSC 'setup_attach' in this example shows the modelling of the connection set-up by means of a free loop.



T1009720-96/d54

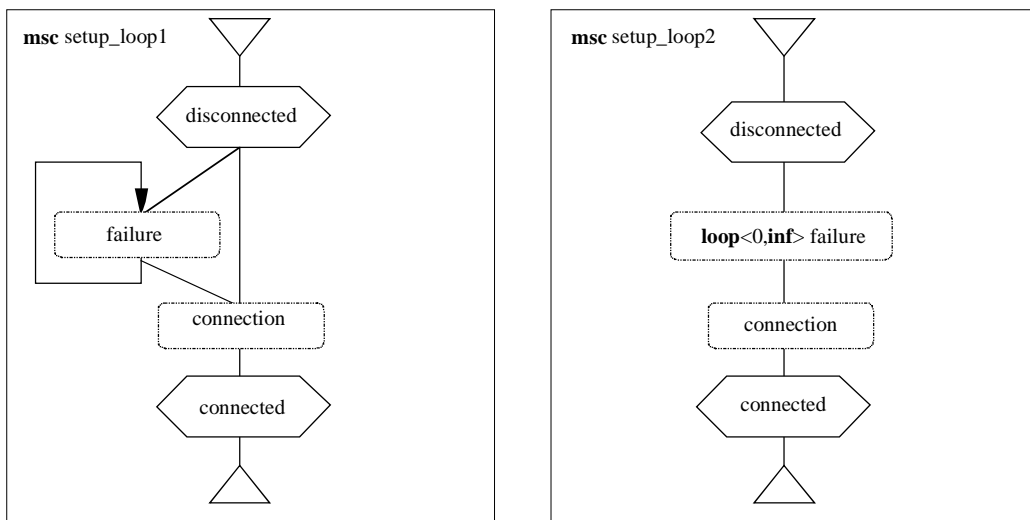
```

mhc setup_attach;
expr L1;
  L1: condition disconnected seq (L2);
  L2: connect seq (L3 alt L4);
  L3: failure seq (L1);
  L4: connection seq (L5);
  L5: condition connected seq (L6);
  L6: end;
endmhc;

```

6.20 High-level MSC with loop

This example shows the modelling of the connection set-up by means of a loop attached to the MSC reference 'failure'.



T1009730-96/d55

```

mhc setup_loop1;
expr L1;
  L1: condition disconnected seq (L2 alt L3);
  L2: failure seq (L2 alt L3);
  L3: connection seq (L4);

```

```

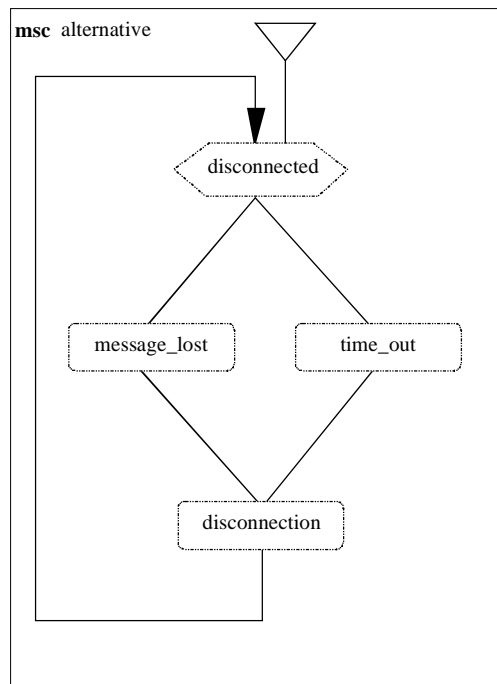
L4: condition connected seq (L5);
L5: end;
endmsc;

msc setup_loop2;
expr L1;
    L1: condition disconnected seq (L2);
    L2: (loop <0,inf> failure) seq (L3);
    L3: connection seq (L4);
    L4: condition connected seq (L5);
    L5: end;
endmsc;

```

6.21 High-level MSC with alternative composition

MSC 'alternative' in this example shows an alternative construct with correct parenthesising where the branching has a corresponding join construct.

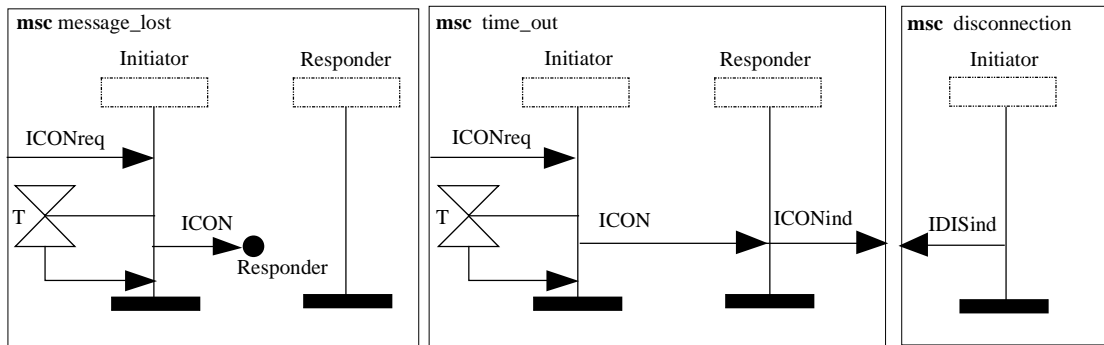


T1009740-96/d56

```

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

```



T1009750-96/d57

```

msc message_lost; inst Initiator, Responder;
    instance Initiator;
        in ICONreq from env;
        set T;
        out ICON to lost Responder;
        timeout T;
    endinstance;
    instance Responder;
        in ICON Initiator;
    endinstance;
endmsc;

```

```

msc time_out; inst Initiator, Responder;
    instance Initiator;
        in ICONreq from env;
        set T;
        out ICON to Responder;
        timeout T;
    endinstance;
    instance Responder;
        in ICON from Initiator;
        out ICONind to env;
    endinstance;
endmsc;

```

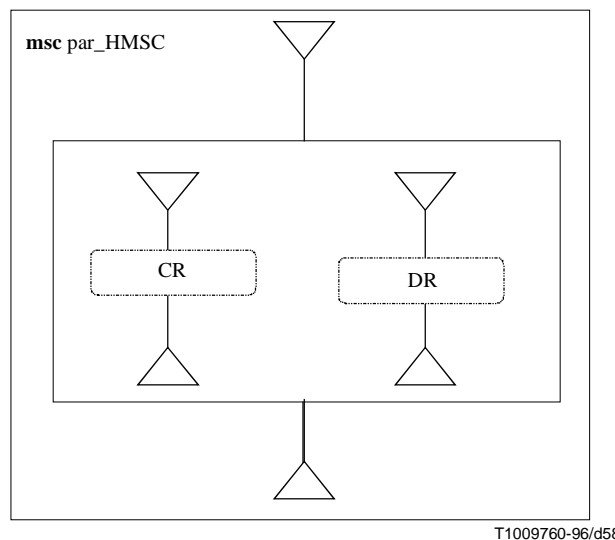
```

msc disconnection; inst Initiator, Responder;
    instance Initiator;
        out IDISind to env;
    endinstance;
endmsc;

```

6.22 High-level MSC with parallel composition

In this example the connection request from the 'Initiator' is parallelly merged with the disconnection request from the 'Responder' by means of the parallel HMSC expression.



T1009760-96/d58

mse par_HMSC

expr L1;

L1: **expr L2;**

L2: CR seq (L3);

L3: **end;**

endexpr;

par

expr L4;

L4: DR seq (L5);

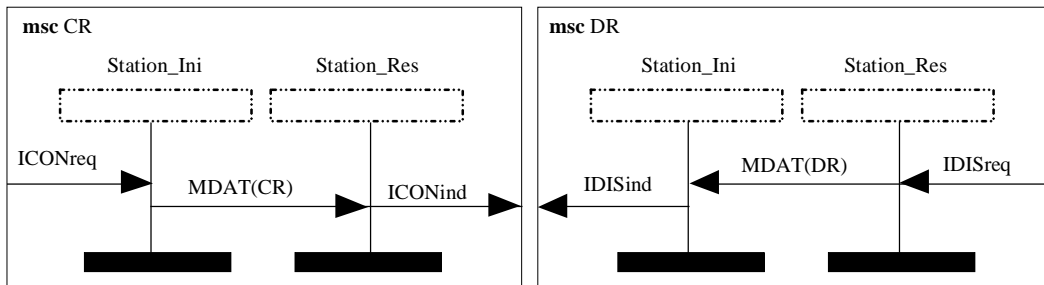
L5: **end;**

endexpr;

seq (L6);

L6: **end;**

endmsc;



T1009770-96/d59

mse CR; inst Station_Ini, 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;

endmsc;

mse DR; inst Station_Ini, Station_Res;

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;

Annex A

Index

The entries are the <keyword>s and the non-terminals from *Concrete textual grammar* and *Concrete graphical grammar*. **Bolded** page numbers refer to definitions of non-terminals.

<action area>, 14, **30**, 32
<action character string>, 30
<action symbol>, 10, 25, **30**
<action text>, 30
<action>, 13, 16, **30**
<actual gate area>, **24**, 38
<actual in gate area>, 19, **24**
<actual in gate>, **22**, 37
<actual order in gate area>, **24**, 25
<actual order in gate>, 13, 22, 37
<actual order out gate area>, **24**, 25
<actual order out gate>, **22**, 37
<actual out gate area>, 19, 24
<actual out gate>, **22**, 37
<address>, 19
<alphanumeric>, 7
<alt area>, 35
<alt expr>, 34
<alt op area>, 39, 41
<apostrophe>, 7
<character string>, 2, 6, **7**, 9, 10
<coevent area>, 32
<coevent layer>, 32
<coevent>, 31
<comment area>, **10**
<comment symbol>, 10
<comment>, 9
<composite special>, 6, **8**
<concurrent area>, 14, **32**
<condition area>, 15, **27**
<condition identification>, 26
<condition name list>, 26, 27, 39, 40
<condition name>, 26, 41, 42, 43
<condition symbol>, 10, 27, 40, 42, 43
<condition>, 13, **26**, 27
<connection point symbol>, 10
<connector layer>, 6, 14
<coregion symbol>, 10, **32**
<coregion symbol1>, 32
<coregion symbol2>, 32
<coregion>, 13, **31**
<create area>, 14, 25, **30**, 32
<create>, 13, **30**
<createline symbol>, 10, 16, 30, 31
<decimal digit>, 7
<decomposition>, 13, 16, 17, **33**
<def gate area>, 14, **23**
<def in gate area>, 19, 23
<def in gate>, 13, 18, **22**
<def order in gate area>, 23, **24**, 25
<def order in gate>, 13, **22**
<def order out gate area>, 23, 25
<def order out gate>, 13, 22
<def out gate area>, 19, 23
<def out gate>, 13, 18, **22**
<document head area>, **12**
<document head>, 12
<duration name>, 28, 29
<end>, 2, **9**, 10, 12, 13, 31, 33, 34, 37, 39
<event area>, 14
<event definition>, 13, 16
<event layer>, 6, 14
<event name list>, 13
<event name>, 13, 22
<exc area>, 35
<exc expr>, 34
<exc inline expression symbol>, **35**, 36
<found message area>, 2, 20
<found message symbol>, 10, 20, 23, 24
<frame symbol>, 12, 14, 40
<full stop>, 7
<gate def layer>, 6, 14
<gate identification>, 22, 23, 24
<gate name>, 13, 18, 20, 22, 24
<general order area>, 19, 20, 23, 24, 28, 30
<general order symbol>, 10, 23, 24

<general order symbol1>, 24, 25
 <general order symbol2>, 24, 25
 <hmsc condition area>, 40
 <hmsc end area>, 39, 40
 <hmsc end symbol>, 10, 41, 42
 <hmsc line symbol>, 40, 41
 <hmsc line symbol1>, 40
 <hmsc line symbol2>, 40
 <hmsc reference area>, 40
 <hmsc start symbol>, 10, 39, 41, 42, 43
 <identifier>, 2, 12, 13
 <incomplete message area>, 2, 14, 20, 25, 32
 <incomplete message event>, 13, 18
 <incomplete message input>, 18
 <incomplete message output>, 18
 <inf natural>, 34
 <inline expr identification>, 18, 33, 34
 <inline expr name>, 34
 <inline expr>, 13, 34
 <inline expression area>, 15, 35, 36
 <inline expression symbol>, 10, 23, 35
 <inline gate area>, 19, 22, 35
 <inline gate interface>, 33, 34, 36
 <inline gate>, 34
 <inline in gate area>, 22, 23
 <inline in gate>, 22, 34
 <inline order gate area>, 23, 25, 35
 <inline order in gate area>, 23
 <inline order in gate>, 22, 34
 <inline order out gate area>, 23
 <inline order out gate>, 13, 22, 34
 <inline out gate area>, 22, 23
 <inline out gate>, 22, 34
 <input address>, 18, 22
 <input dest>, 22
 <instance area>, 14, 16
 <instance axis symbol>, 19, 27, 28, 29, 30, 32, 35, 38
 <instance axis symbol2>, 32
 <instance body area>, 16, 17
 <instance end statement>, 13, 14, 16
 <instance end symbol>, 10, 17
 <instance event area>, 14
 <instance event list>, 13, 33, 34
 <instance event>, 13, 16
 <instance head area>, 11, 16, 30
 <instance head statement>, 13, 14, 16
 <instance head symbol>, 10, 16, 17, 30
 <instance heading>, 16, 17
 <instance kind>, 12, 13, 16, 17
 <instance layer>, 6, 14
 <instance list>, 2, 12, 14
 <instance name list>, 13, 16
 <instance name>, 12, 13, 16, 17, 18, 20, 26, 30, 37
 <keyword>, 1, 6, 8, 65
 <kind denominator>, 13
 <label name>, 39
 <left curly bracket>, 7
 <left square bracket>, 7
 <letter>, 7, 9
 <lexical unit>, 6, 9
 <loop area>, 35
 <loop boundary>, 33, 34, 35, 37
 <loop expr>, 34
 <lost message area>, 2, 20
 <lost message symbol>, 10, 20, 23, 24
 <message area>, 14, 19
 <message end area>, 19, 20
 <message event area>, 14, 19, 25, 32
 <message event>, 13, 18
 <message in area>, 19
 <message in symbol>, 10, 19
 <message input>, 18, 19, 27, 33
 <message instance name>, 18, 19
 <message name>, 18, 19, 37
 <message out area>, 19
 <message out symbol>, 10, 19
 <message output>, 18, 19, 27, 33
 <message sequence chart>, 2, 12
 <message start area>, 19, 20
 <message symbol>, 19, 23, 24
 <msc body area>, 6, 14
 <msc body>, 12, 13, 14, 34
 <msc diagram>, 2, 12, 14
 <msc document body>, 2, 11, 12
 <msc document head>, 11, 12
 <msc document name>, 12
 <msc document>, 11
 <msc expression>, 12, 39
 <msc gate def>, 13
 <msc gate interface>, 12, 13, 16

<msc head>, 12
 <msc heading>, 11, 14
 <msc inst interface>, 2, 12, 16
 <msc interface>, 12, 14, 16
 <msc name>, 2, 12, 14, 37, 38, 39
 <msc ref exc expr>, **37**
 <msc ref expr>, 37, 38, 39
 <msc ref loop expr>, 37
 <msc ref opt expr>, 37
 <msc ref par expr>, 37
 <msc ref seq expr>, 37
 <msc reference area>, 15, **38**
 <msc reference identification>, 18, 37, 38
 <msc reference name>, 37, 38
 <msc reference symbol>, 10, 24, 38, 41
 <msc reference>, 13, **37**, 42
 <msc statement>, 13
 <msc symbol>, 14, 23
 <mscexpr area>, 14, **39**, 40
 <msg gate>, 13
 <msg identification>, 18, 19, 20, 22
 <multi instance event list>, 13
 <multi instance event>, 13, 16
 <name>, 2, **8**, 9, 12, 13
 <national>, 7, 9
 <natural name>, 34
 <node area>, 41
 <node expression area>, 39, **40**, 41
 <non-orderable event>, 13
 <note>, 9
 <note>, 6, **8**, 9
 <old instance head statement>, 13, 14
 <operand area>, **35**
 <opt area>, 35
 <opt expr>, 34
 <order dest>, 22
 <order gate>, 13
 <orderable event>, 13, 31
 <ordered event area>, 24, **25**
 <other character>, 7
 <output address>, 18, 22
 <output dest>, 22
 <overline>, 7
 <par area>, 35
 <par expr area>, 40, 42, 43
 <par expr>, **34**
 <par expression>, 39
 <par frame symbol>, 10, 40
 <parameter list>, 18, 19, 30
 <parameter name>, 18
 <parameter substitution>, 37
 <qualifier>, 2, 12
 <ref gate>, 37
 <reference gate interface>, **37**, 38
 <reference identification>, 18, 22
 <replace instance>, 37
 <replace message>, 37
 <replace msc>, 37
 <reset symbol>, 10, **29**
 <reset symbol1>, 28, 29
 <reset symbol2>, 28, 29
 <reset>, 28
 <right curly bracket>, 7
 <right square bracket>, 7
 <sdl document identifier>, 12
 <sdl reference>, 12
 <separator area>, **35**
 <separator symbol>, 10
 <set symbol>, 10, 28, 29
 <set symbol1>, **28**
 <set symbol2>, **28**
 <set>, 28
 <set-reset symbol>, **28**, 29
 <shared alt expr>, 33, 34
 <shared condition>, 13, 16, **26**
 <shared event area>, 14, **15**
 <shared exc expr>, 33, 34
 <shared inline expr>, 13, **33**
 <shared instance list>, 26
 <shared loop expr>, 33
 <shared msc reference>, 13, **37**
 <shared opt expr>, 33
 <shared par expr>, 33, 34
 <shared>, **26**, 33, 34, 37
 <space>, 7, 9
 <special>, 6, 7
 <start area>, **39**
 <start>, 39
 <stop symbol>, 10, **31**
 <stop>, 13, 14, **31**

<substitution list>, 37, 38
<substitution>, 37
<substructure reference>, 33
<text area>, 6, **10**, 14
<text definition>, **10**, 13
<text layer>, 6, 14
<text symbol>, 10
<text>, **7**, 8, 10, 12
<timeout area>, 28, **29**
<timeout area1>, 29
<timeout area2>, **29**
<timeout symbol>, 10, 29
<timeout symbol1>, **29**
<timeout symbol2>, **29**
<timeout symbol3>, 28
<timeout>, 28
<timer area>, 14, 25, **28**, 32
<timer instance name>, 28
<timer name>, 28, 29
<timer reset area>, **28**
<timer reset area1>, 28
<timer reset area2>, 28, 29
<timer set area>, 28
<timer set area1>, 28
<timer set area2>, 28
<timer statement>, 13, **28**
<underline>, **7**, 8, 9
<upward arrow head>, 7
<vertical line>, 7
<void symbol>, 19, 23, 24
<word>, 6, **7**, 8

—A—

action, 8, 30
all, 8, 16, 26
alt, 8, 34, 35, 36, 37, 39
as, 8

—B—

before, 8, 22
begin, 8, 33, 34
block, 8, 33
by, 8, 37

—C—

comment, 8, 9
concurrent, 8, 31
condition, 8, 25, 26, 39
connect, 8, 39
create, 8

—D—

decomposed, 8, 33

—E—

empty, 8, 37, 38, 39
end, 8, 34, 39
endconcurrent, 8, 31
endexpr, 8, 39
endinstance, 8, 16
endmsc, 8, 12
env, 8, 18, 22, 33
exc, 8, 35, 36, 37
expr, 8, 39
external, 8, 22

—F—

found, 8, 18, 22
from, 8, 18, 22

—G—

gate, 8

—I—

in, 8, 18, 22
inf, 8, 34, 36
inline, 8, 18
inst, 8, 12, 37
instance, 8, 16

—L—

loop, 8, 33, 34, 35, 36, 37
lost, 8, 18, 22

—M—

msc, 8, 12, 14, 37
mscdocument, 8, 12
msg, 8, 37

—O—

opt, 8, 35, 36, 37
order, 8
out, 8, 18, 22

—P—

par, 8, 34, 36, 37, 42
process, 8, 13, 33

—R—

reference, 8, 18, 37
related, 8
related to, 12
reset, 8, 28

—S—

seq, 8, 36, 37, 39
service, 8, 13, 33
set, 8, 28
shared, 8, 16, 25, 26
shared all, 25, 27
stop, 8, 31
subst, 8, 36, 37, 38
system, 8, 13, 33

—T—

text, 9, 10
timeout, 9, 28
to, 9, 18, 22

—V—

via, 9, 18, 22

ITU-T RECOMMENDATIONS SERIES

- Series A Organization of the work of the ITU-T
- Series B Means of expression: definitions, symbols, classification
- Series C General telecommunication statistics
- Series D General tariff principles
- Series E Overall network operation, telephone service, service operation and human factors
- Series F Non-telephone telecommunication services
- Series G Transmission systems and media, digital systems and networks
- Series H Audiovisual and multimedia systems
- Series I Integrated services digital network
- Series J 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 Maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
- Series N Maintenance: international sound programme and television transmission circuits
- Series O Specifications of measuring equipment
- Series P Telephone transmission quality, telephone installations, local line networks
- Series Q Switching and signalling
- Series R Telegraph transmission
- Series S Telegraph services terminal equipment
- Series T Terminals for telematic services
- Series U Telegraph switching
- Series V Data communication over the telephone network
- Series X Data networks and open system communication
- Series Z Programming languages**