Static semantics of message sequence charts

Reniers, M.A.

Published: 01/01/1996

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Static Semantics
of
Message Sequence Charts

by

M.A. Reniers

96/19

ISSN 0926-4515

All rights reserved
editors: prof.dr. R.C. Backhouse
        prof.dr. J.C.M. Baeten

Reports are available at:
http://www.win.tue.nl/win/cs

Computing Science Report 96/19
Eindhoven, September 1996
Static Semantics
of
Message Sequence Charts

M.A. Reniers
Department of Mathematics and Computing Science,
Eindhoven University of Technology.
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands.
Chapter 1

Introduction

The purpose of this document is to formally define the syntax requirements of Message Sequence Charts. The description of the syntax requirements as presented in Recommendation Z.120 [IT93] is open to ambiguous interpretation and should therefore be reconsidered. A new set of syntax requirements is formulated and their formalization is presented. This formalization is based on predicates and functions. The approach taken towards the formalization of the static semantics has been illustrated in [Ren95].

Sequence Charts are a widespread means for the description and specification of selected system runs within distributed systems with asynchronous communication, especially telecommunication systems. Other areas for application of Sequence Charts are as an overview language of a service offered by a number of entities, a requirements statement for SDL specifications, simulation and validation, selection and specification of test cases, formal specification of communication, and interface specification. Within industry Sequence Charts are used mainly as a test case description language. Various kinds of Sequence Charts are used although they differ on minor points only. One can think of the following: Extended Sequence Charts [GR89], Time Sequence Diagrams [ISO91], Arrow Diagrams [CCI88], Information Flow Diagrams [CCHvK90], Synchronous Interworkings [MvWW93, MW93] and Siemens-SCs [Sie92]. A comparison of these languages can be found in [GGR93a]. To enhance tool support, feasibility of Sequence Chart exchange between tools, and harmonization of the use of Sequence Charts within CCITT Study Groups, a standardization of such Sequence Charts was proposed by the CCITT (nowadays called the ITU (International Telecommunication Union)). The recommended version of Sequence Charts is called Message Sequence Charts.

Recommendation Z.120 Message Sequence Chart (MSC) [IT93] contains a description of an abstract syntax, a graphical syntax, and a textual syntax of the language Message Sequence Chart. Besides these syntax descriptions also an informal semantics and an informal description of the syntax requirements are given.

The need for a formal semantics became evident since even experts in the field of Message Sequence Charts could not always agree on the interpretation of specific features. Furthermore validation of computer tools for Message Sequence Charts only makes sense if an exact mean-
A formal semantics will help to harmonize the use of Message Sequence Charts. A formal semantics based on process algebra is proposed for standardization in [MR94b, IT95a]. Other approaches towards the definition of a formal semantics are based on automaton theory [LL92b, LL92a, LL94], Petri net theory [GGR93b], and process algebra [dM93, MvWW93, MW93, MR94a].

Besides the formalization of the dynamic semantics of Message Sequence Charts, also the syntax requirements (static semantics) of Message Sequence Charts need to be formalized. The reasons for doing so are the same as for formalizing the dynamic semantics.

The document is structured as follows. In the next chapter a short introduction to Message Sequence Charts is given. In Chapter 3 some basic notions on relations and multisets are recapitulated. Those will be used frequently in Chapter 4. In Chapter 4 the syntax requirements of Message Sequence Charts are discussed and formalized. Appendix A contains the concrete textual syntax for Message Sequence Charts as it is used in this document. Appendix B contains the definition of a number of functions used in the formalization of Chapter 4.

Acknowledgements We would like to thank Anders Ek (Telelogic), Jens Grabowski (Swiss PTT), Oystein Haugen (Norwegian Computing Center), Sjouke Mauw (Eindhoven University of Technology), and Ekkart Rudolph (Siemens) for their stimulating criticism. Also all other people which participated in the various email discussions and the ITU Interims Meeting in Geneva have been very helpful.
Chapter 2

Message Sequence Charts

2.1 Introduction

This chapter contains an introduction to Message Sequence Charts. First the core language of Message Sequence Charts is introduced. This core language is called Basic Message Sequence Charts. A Basic Message Sequence Chart concentrates on communications and local actions only. These are the features encountered in most languages comparable to Message Sequence Charts as mentioned in the previous chapter. After the introduction of Basic Message Sequence Charts the other primitives incorporated in the language of Message Sequence Charts are introduced. These primitives are conditions, timer handling, process creation and process termination, coregions, and instance decomposition. With respect to instance decomposition, the syntax used in this document deviates slightly from the syntax presented in Recommendation Z.120 [IT93].

2.2 Basic Message Sequence Charts

A Basic Message Sequence Chart is a finite collection of instances. An instance is an abstract entity on which message outputs, message inputs and local actions may be specified. An instance is denoted by a vertical axis. The time along each axis is running from top to bottom. The events specified on an instance are totally ordered in time; no notion of global time is assumed. No two events on an instance are executed at the same time. An instance is labelled with a name, the instance name. This name is placed above the axis representing the instance.

A local action is denoted by a box on the axis with the action text placed in it. A message between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. A message is split into a message output and a message input. A message sent by an instance to the environment is represented by an arrow from the sending instance to the exterior of the Message Sequence Chart. A message received from the environment is represented by an arrow from the exterior of the Message Sequence Chart to the receiving instance. A
message may be labelled with a parameter list. The parameter list is denoted between brackets after the message name.

**Example 1** Consider the messages \( m_1, m_2, m_3 \) and \( m_4 \) in Figure 2.1. Message \( m_0 \) is sent to the environment. The behaviour of the environment is not specified. For instance \( i_2 \) also a local action \( a \) is defined.

The only dependencies between the timing of the instances come from the restriction that a message must be sent before it is consumed. In Figure 2.1 this implies that message \( m_3 \) is received by \( i_4 \) only after it has been sent by \( i_3 \), and, consequently, after the consumption of \( m_2 \) by \( i_3 \). Thus the events concerning \( m_1 \) and \( m_3 \) are ordered in time, while for the events of \( m_4 \) and \( m_3 \) no order is specified apart from the requirement that the output of a message occurs before its input. The execution of a local action is only restricted by the ordering of events on the instance it is defined on. The second Basic Message Sequence Chart in Figure 2.1 defines the same Basic Message Sequence Chart (from a semantic point of view), but in an alternative drawing.
Because of the asynchronous communication, it would even be possible to first send m3, then send and receive m4, and finally receive m3. Another consequence of this mode of communication is that overtaking of messages is allowed, as expressed in Figure 2.2.

Although the application of Message Sequence Charts is mainly focused on the graphical representation, they have a concrete textual syntax. This representation was originally intended for exchanging Message Sequence Charts between computer tools only, but in this document it is used for the discussion and formalization of the syntax requirements.

The textual representation of a Basic Message Sequence Chart is instance oriented. This means that a Basic Message Sequence Chart is defined by specifying the behaviour of all instances. A message output is denoted by “out m1 to i2;” and a message input by “in m1 from i1;”.

The Basic Message Sequence Charts of Figure 2.1 have the following textual representation.

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

In the graphical representation the correspondence between message outputs and message inputs is given by the arrow construction. In the textual representation this correspondence is given by message identifier identification.

The grammar defining the textual syntax of Basic Message Sequence Charts is given in Table 2.1. The nonterminals <at>, <inst name>, <kn>, <min>, <mn>, <msc name>, and <par name> represent identifiers. The symbol <> denotes the empty string. The following identifiers are reserved keywords: action, block, endinstance, endmsc, endtext, env, from, in, inst, instance, msc, out, process, service, system, text, and to. The nonterminal <text> represents an arbitrary text.
Table 2.1: The concrete textual syntax of Basic Message Sequence Charts

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;msc&gt;</code></td>
<td>msc <code>&lt;msc head&gt;</code> <code>&lt;msc body&gt;</code> endmsc;</td>
</tr>
<tr>
<td><code>&lt;msc head&gt;</code></td>
<td><code>&lt;msc name&gt;</code>; [ <code>&lt;msc interface&gt;</code> ];</td>
</tr>
<tr>
<td><code>&lt;msc interface&gt;</code></td>
<td>inst <code>&lt;inst list&gt;</code>;</td>
</tr>
<tr>
<td><code>&lt;inst list&gt;</code></td>
<td><code>&lt;inst name&gt;</code> [:<code>&lt;inst kind&gt;</code>] [, <code>&lt;inst list&gt;</code>]</td>
</tr>
<tr>
<td><code>&lt;inst kind&gt;</code></td>
<td><code>[ </code>&lt;type&gt;<code>]</code>&lt;kn&gt;`</td>
</tr>
<tr>
<td><code>&lt;type&gt;</code></td>
<td>system</td>
</tr>
<tr>
<td><code>&lt;msc body&gt;</code></td>
<td><code>&lt;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;inst def&gt;</code></td>
<td>instance <code>&lt;inst head&gt;</code> <code>&lt;inst body&gt;</code> endinstance;</td>
</tr>
<tr>
<td><code>&lt;inst head&gt;</code></td>
<td><code>&lt;inst name&gt;</code> [:<code>&lt;inst kind&gt;</code>];</td>
</tr>
<tr>
<td><code>&lt;inst body&gt;</code></td>
<td><code>&lt;&gt;</code></td>
</tr>
<tr>
<td><code>&lt;event&gt;</code></td>
<td><code>&lt;out&gt;</code></td>
</tr>
<tr>
<td><code>&lt;out&gt;</code></td>
<td>out <code>&lt;msgid&gt;</code> to <code>&lt;address&gt;</code>;</td>
</tr>
<tr>
<td><code>&lt;msgid&gt;</code></td>
<td><code>&lt;mid&gt;</code> [:<code>&lt;par list&gt;</code>]</td>
</tr>
<tr>
<td><code>&lt;par list&gt;</code></td>
<td><code>&lt;par name&gt;</code> [:<code>&lt;par list&gt;</code>]</td>
</tr>
<tr>
<td><code>&lt;address&gt;</code></td>
<td><code>&lt;inst name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;in&gt;</code></td>
<td>in <code>&lt;msgid&gt;</code> from <code>&lt;address&gt;</code>;</td>
</tr>
<tr>
<td><code>&lt;action&gt;</code></td>
<td>action <code>&lt;at&gt;</code>;</td>
</tr>
<tr>
<td><code>&lt;text def&gt;</code></td>
<td>text <code>&lt;note&gt;</code> endtext;</td>
</tr>
<tr>
<td><code>&lt;note&gt;</code></td>
<td>/* <code>&lt;text&gt;</code> */</td>
</tr>
</tbody>
</table>

The language generated by a nonterminal X in the grammar will be denoted by $\mathcal{L}(X)$.

2.3 Conditions

A condition describes a state referring to a (non-empty) subset of instances specified in the Message Sequence Chart. Conditions are used for documentation purposes in the sense of comments or illustrations. In case of a whole set of Message Sequence Charts conditions determine possible continuations of Message Sequence Charts by means of condition identification.

Example 2 In the graphical representation a condition is represented by a hexagon that is placed on top of the instances it refers to. If a condition crosses an instance axis which is not involved in the condition the instance axis is drawn through the condition. A condition is labelled with a condition name that is placed inside the hexagon.

In Figure 2.3 a Message Sequence Chart with three conditions is given. Condition $C1$ refers to all instances, $C2$ refers to instance $i$ and $C3$ refers to instances $i$ and $k$. 
In the textual representation the condition has to be defined on every instance it refers to using the reserved keyword condition together with the condition name. If the condition refers to several instances then the reserved keyword shared together with the instance list denotes the set of all instances with which the condition is shared. If the condition refers to all instances of the chart, the instance list may be replaced by the reserved keyword all.

In Table 2.2 the rules for the extension with conditions are given. The nonterminal <cn> represents an identifier. The identifiers all, condition, and shared are reserved keywords.

Table 2.2: Extension with conditions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;event&gt;</td>
<td>::= &lt;condition&gt;</td>
</tr>
<tr>
<td>&lt;condition&gt;</td>
<td>::= condition &lt;cn&gt;</td>
</tr>
<tr>
<td></td>
<td>[shared { &lt;shared inst list&gt;</td>
</tr>
<tr>
<td>&lt;shared inst list&gt;</td>
<td>::= &lt;inst name&gt; [,&lt;shared inst list&gt;]</td>
</tr>
</tbody>
</table>

The Message Sequence Chart in Figure 2.3 is in the textual representation given by

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

2.4 Timer handling

In Message Sequence Charts, either the setting of a timer and its subsequent timeout due to timer expiration or the setting of a timer and its subsequent timer reset (time supervision) may be specified. The setting of a timer is denoted by a small rectangle placed adjacent to the instance axis, a timeout is represented by an arrow from the timer set symbol to the axis, and a timer reset is represented by a modified timeout symbol with a dashed arrow. A timer event is labelled by an identifier, the timer name, that is placed aside the small rectangle. The setting of a timer may be labelled with an identifier for the duration, the duration name. The duration name is placed between brackets after the timer name. A timer event is local to the instance it is specified on.

Example 3 In Figure 2.4 on instance $i$ the setting of a timer $T$ with duration $d$ and its subsequent timer reset, and on instance $j$ the setting of a timer $T$ and its subsequent timeout are specified.

![Figure 2.4: Message Sequence Chart with timer handling](image)

In the graphical representation the correspondence between a timer set and a timer reset or timeout is given by the connection of the begin of the timer reset or timeout symbol to the rectangle representing the timer set. In the textual representation the correspondence between timer
set and timer reset or timeout is given by timer identifier identification. The setting of a timer with identifier \( T \) is denoted by "set \( T \);" and the corresponding reset by "reset \( T \);" and timeout by "timeout \( T \);." The grammar in Table 2.2 is extended with the rules in Table 2.3. The nonterminals \(<\text{dn}>\), \(<\text{tin}>\), and \(<\text{tn}>\) represent identifiers. The identifiers \( \text{reset} \), \( \text{set} \), and \( \text{timeout} \) are reserved keywords.

<table>
<thead>
<tr>
<th>Table 2.3: Extension with timer-handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{event}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{set}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{tid}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{reset}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{timeout}&gt;)</td>
</tr>
</tbody>
</table>

The Message Sequence Chart in Figure 2.4 is represented as follows.

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

### 2.5 Process creation and process termination

In the language of Message Sequence Charts a primitive is incorporated for the dynamic creation of an instance by another instance. Such a creation is denoted by a dashed arrow from the creating instance to the top symbol of the created instance. As was the case for communication events, a create event may be labelled with a parameter list.

An instance can terminate by executing a process stop event. Execution of a process stop is allowed only as a last event in the description of an instance. A process stop is denoted by replacing the bottom symbol of the instance by a cross.
Example 4 In Figure 2.5 a Message Sequence Chart with three instances is given. Instance \( i \) creates instance \( j \), instance \( k \) sends a message \( m \) to instance \( j \), and instance \( j \) receives the message \( m \) from instance \( k \) after it is created and then terminates.

![Message Sequence Chart](msc_creation)

Figure 2.5: Message Sequence Chart with process creation and termination

In the textual representation the creation of an instance with name \( j \) is denoted by "create \( j \);" and the termination of an instance by "stop;". The grammar is extended with the rules in Table 2.4. The identifiers create and stop are reserved keywords.

<table>
<thead>
<tr>
<th>Table 2.4: Extension of the grammar with process creation and termination</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{inst body}&gt; ) ::= \text{stop};</td>
</tr>
<tr>
<td>(&lt;\text{event}&gt; ) ::= (&lt;\text{create}&gt;)</td>
</tr>
<tr>
<td>(&lt;\text{create}&gt; ) ::= \text{create} (&lt;\text{inst name}&gt;) [(&lt;\text{par list}&gt;)];</td>
</tr>
</tbody>
</table>

The textual representation of the Message Sequence Chart in Figure 2.5 is as follows.

\[
\text{msc creation;}
\text{instance } i; \\
\text{create } j; \\
\text{endinstance;}
\text{instance } j; \\
\text{in } m \text{ from } k; \\
\text{stop;}
\text{endinstance;}
\]
2.6 Coregions

So far the events specified on an instance were totally ordered in time. To enable the specification of unordered events on an instance the coregion is introduced. A coregion is a dashed part of the instance axis for which the events specified within that part are assumed to be unordered in time. Within a coregion only communication events may be specified.

**Example 5** In Figure 2.6 an instance with a coregion is specified which contains an input of message $m$ and an output of a message $n$. These two events are not ordered in time, but they are executed after the output of message $k$ and before the input of message $l$. On instance $j$ the events are totally ordered in time.

![Figure 2.6: Message Sequence Chart with a coregion](image)

In the textual notation a coregion is denoted by a list of the message events specified within the coregion started with the reserved keyword `concurrent` and ended by the reserved keyword `endconcurrent`. In Table 2.5 the rules for the extension with coregions are given.
Table 2.5: Extension with coregions

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;event&gt;</code></td>
<td>:=</td>
<td><code>&lt;coregion&gt;</code></td>
</tr>
<tr>
<td><code>&lt;coregion&gt;</code></td>
<td>:=</td>
<td>concurrent <code>&lt;coevents</code> endconcurrent;</td>
</tr>
<tr>
<td><code>&lt;coevents&gt;</code></td>
<td>:=</td>
<td><code>&lt;&gt;</code></td>
</tr>
</tbody>
</table>

The textual representation of the Message Sequence Chart in Figure 2.6 is as follows.

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

2.7 Instance decomposition

Since charts can be rather complex, there is a need for the decomposition of one instance by a set of instances defined in another chart. By means of the keyword decomposed placed in the top symbol of the instance a chart with the same name may be attached to that instance. This chart represents a decomposition of the instance without affecting its observable behaviour. The decomposition of an instance into a chart must not affect the ordering of the communication events defined on the decomposed instance. There is no formal mapping between non-communication events specified in the chart and the events specified on the decomposed instance.

**Example 6** In Figure 2.7 a document with two charts is specified. Instance \(d\) of chart \(decinst\) is decomposed by chart \(d\).
In the textual representation these charts are represented by

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

```plaintext
msc d;
    instance j;
       in m from env;
       out o to k;
    endinstance;
    instance k;
       in o from j;
       out n to env;
    endinstance;
endmsc;
```

In the textual representation of a Message Sequence Chart, a decomposed instance is labelled with the reserved keyword decomposed. Because of the decomposition primitive a complete description consists of a collection of charts. Such a collection of charts is called a Message Sequence Chart document. In the remaining part of this Annex the word document is used instead of Message Sequence Chart document. A document may contain more than one chart. In Table 2.6 the rules for the extension with instance decomposition are given. The nonterminals <doc name> and <sdl ref> represent identifiers. The identifiers decomposed, endmscdocument, mscdocument, and related are reserved keywords.
Within the standard of Message Sequence Charts [IT93] there are two types of charts: Message Sequence Charts and Sub Message Sequence Charts. A Sub Message Sequence Chart is distinguished from an ordinary Message Sequence Charts by its use of the reserved keywords submsc and endsubmsc in stead of msc and endmsc. Only Sub Message Sequence Charts may represent a decomposition of an instance in some other chart (either a Message Sequence Chart or a Sub Message Sequence Chart).

Since the difference between Message Sequence Charts and Sub Message Sequence Charts is only in this respect, only one type of chart is considered. This type of chart is allowed to represent a decomposition of an instance. In no way, this restriction implies that the treatment of the syntax requirements does not cover the essentials of the complete standardized language. The only requirement from Recommendation Z.120 [IT93] which no longer applies is that every Sub Message Sequence Chart must be referenced by some decomposed instance in some chart in the Message Sequence Chart document.
Chapter 3

Preliminaries

In this chapter some basic notions on sets, relations and multisets are introduced. These will be used frequently in the formalization of the syntax requirements of Message Sequence Charts in the following chapters.

The union of two sets $S_1$ and $S_2$ is denoted by $S_1 \cup S_2$ and the disjunction of those sets by $S_1 \cap S_2$. Membership test on sets is denoted by $\in$ and set inclusion by $\subseteq$. The symbol \ denotes set difference. The Cartesian product of the sets $S_1$ and $S_2$ is denoted by $S_1 \times S_2$. The powerset of a set $S$ is denoted by $\mathcal{P}(S)$.

A function $f$ which maps elements of the set $A$ onto elements of the set $B$ is denoted by $f : A \rightarrow B$.

A binary relation on a set $A$ is a subset of $A \times A$. In this document only binary relations are considered. Therefore, the adjective binary is left implicit in the remainder. Next, some special kinds of relations are introduced. These are all well known from literature.

Let $R \subseteq A \times A$ be a relation.

1) The relation $R$ is called reflexive if for all $a \in A$: $(a,a) \in R$.

2) The relation $R$ is called symmetric if for all $a,b \in A$: if $(a,b) \in R$, then $(b,a) \in R$.

3) The relation $R$ is called asymmetric if for all $a,b \in A$: if $(a,b) \in R$ and $a \neq b$, then $(b,a) \notin R$.

4) The relation $R$ is called transitive if for all $a,b,c \in A$: if $(a,b) \in R$ and $(b,c) \in R$, then $(a,c) \in R$.

5) The relation $R$ is called strict if for all $a \in A$: $(a,a) \notin R$.

6) The relation $R$ is called an equivalence relation if it is reflexive, symmetric, and transitive.

7) The relation $R$ is called a partial order if it is reflexive, asymmetric, and transitive.

The transitive closure of $R$, notation $R^+$, is the smallest relation that satisfies for all $a,b,c \in A$:
1) if \((a, b) \in R\), then \((a, b) \in R^+\);
2) if \((a, b) \in R^+\) and \((b, c) \in R^+\), then \((a, c) \in R^+\).

The reflexive-transitive closure of \(R\), notation \(R^*\) is the smallest relation that satisfies for all \(a, b \in A\):

1) \((a, a) \in R^*
2) if \((a, b) \in R^+\), then \((a, b) \in R^*\)

In order to denote arbitrary relations letter symbols are used, e.g. \(R\). To denote equivalence relations symbols like \(\sim\) and \(=\) are used in an infix notation style.

If \(\sim \subseteq A \times B\) is an equivalence relation, then \(a \not\sim b\) \((a \in A\) and \(b \in B\)) denotes the negation of \(a \sim b\).

Multisets are a generalization of sets, by allowing elements to have multiple occurrences in a multiset. A multiset is represented by listing its members in arbitrary order between the brackets \([\] and \(]\). For example, the multiset with two occurrences of \(a\) and one occurrence of \(b\) is denoted by \([a, a, b]\), \([a, b, a]\), or \([b, a, a]\). The empty multiset is denoted by \(\square\). The union of two multisets \(M\) and \(N\) is denoted by \(M \cup N\). The membership test is denoted by \(\subseteq\). The difference of two multisets is denoted by \(-\). For a set \(A\), the set of all multisets over \(A\) is denoted by \(M(A)\).

Relations and functions on multisets can be defined inductively using \(\square\) for the base case and \([a] \cup M\) for the composition case.

Let \(A\) and \(B\) be arbitrary sets. Let \(f : A \to B\) and \(M \in M(A)\). The multiset which contains \(f(a)\) for every \(a \in M\), notation \([f(a) \mid a \in M]\), is defined by

\[
[f(a) \mid a \in M] = \square \quad [f(a) \mid a \in [b] \cup M] = [f(b)] \cup [f(a) \mid a \in M]
\]

Let \(A\) and \(B\) be arbitrary sets. Let \(f : A \to M(B)\) and \(M \in M(A)\). The generalized multiset union, notation \(\bigcup_{a \in M} f(a)\), is defined by

\[
\bigcup_{a \in \square} f(a) = \square \\
\bigcup_{a \in [b] \cup M} f(a) = [f(b)] \cup \bigcup_{a \in M} f(a)
\]

Let \(A\) be a set and \(a \in A\) and \(\sim\) an equivalence relation on \(A\). For \(M \in M(A)\), \(\#_\sim^a(M)\) denotes the number of elements from \(M\) that are \(\sim\)-equivalent to \(a\), and is defined by

\[
\#_\sim^a(\square) = 0 \\
\#_\sim^a([b] \cup M) = \begin{cases} 
\#_\sim^a(M) & \text{if } a \not\sim b \\
1 + \#_\sim^a(M) & \text{if } a \sim b
\end{cases}
\]
Example 7 The use of the operator \( \hat{\ } \) is illustrated by the following example. Suppose that a multiset \( M = [0, 0, 1, 2, 2, 2] \) over the natural numbers \( \mathbb{N} \) is given. Denote the normal equivalence relation on \( \mathbb{N} \) by \( = \). The expression \( \#_n \) denotes the number of occurrences of the natural number \( n \) in the multiset \( M \). Then, for \( n \in \mathbb{N} \), the following equalities hold:

\[
\#_0(M) = 2 \quad \#_1(M) = 1 \quad \#_2(M) = 3 \quad \#_{n+3}(M) = 0
\]

The symbol \( \forall \) denotes a universal quantification and the symbol \( \exists \) denotes an existential quantification.
Chapter 4

Requirements for Message Sequence Charts

4.1 Introduction

In this chapter the syntax requirements for well-formedness of Message Sequence Charts are discussed. Each syntax requirement is explained informally and stated in English phrases. Then a formalization of the requirements is obtained by using functions and predicates. Some auxiliary functions needed in formalizing the syntax requirements are defined separately in Appendix B.

4.2 Uniqueness of instances

A chart consists of a finite number of instances. An instance is referenced through its name (for example by message outputs and message inputs). Therefore, it must be possible to determine an instance on basis of its name. Consider for example the chart given in Figure 4.1.

![Example Message Sequence Chart](image)

Figure 4.1: Example Message Sequence Chart
The textual representation of this chart is as follows:

\[
\text{msc example1;}
\text{instance i;}
\text{\hspace{1em}out m(p) to env;}
\text{endinstance;}
\text{instance j;}
\text{\hspace{1em}create i(r);}
\text{endinstance;}
\text{instance i;}
\text{\hspace{1em}out n(q) to env;}
\text{endinstance;}
\text{endmsc;}
\]

The chart contains two instances with name \text{i}. Therefore, it is not clear (in the textual representation) which of the two is created by instance \text{j}. Therefore, the following syntax requirement is formulated.

\text{Within a chart there must not be two or more instances with the same name} \text{ [IT93, section 2.2, page 3].}

In the remainder of this section this requirement will be formalized. First, an equivalence on instance definitions is defined which identifies instances with the same name without considering the kind names in the instance heads.

\textbf{Definition 8} The relation \( \text{\textless \text{inst\_def}\textgreater} \subseteq \mathcal{L}(\text{\textless \text{inst\_def}\textgreater}) \times \mathcal{L}(\text{\textless \text{inst\_def}\textgreater}) \) is for all \( i_1, i_2 \in \mathcal{L}(\text{\textless \text{inst\_def}\textgreater}) \) defined by

\[
i_1 \overset{\text{\textless \text{inst\_def}\textgreater}}{=} i_2 \text{ if and only if } \text{InstName}(i_1) = \text{InstName}(i_2)
\]

The function \text{InstName} associates to an instance definition its instance name. Its definition is given in Appendix B. From the fact that \( = \) is an equivalence on \( \mathcal{L}(\text{\textless \text{name}\textgreater}) \), it follows immediately that \( \text{\textless \text{inst\_def}\textgreater} \) is an equivalence relation on \( \mathcal{L}(\text{\textless \text{inst\_def}\textgreater}) \). The notation \( \mathcal{L} \) has been explained in Section 2.2.

The syntax requirement specifies that there must not be two or more instances with the same name within a chart. This means that there must not be two or more instances which are \( \overset{\text{\textless \text{inst\_def}\textgreater}}{=} \)-equivalent. Using the function \text{AllInsts} (see Appendix B) which associates to a chart the multiset containing all its instance definitions, the formalization amounts to testing whether this multiset contains two or more instances which are \( \overset{\text{\textless \text{inst\_def}\textgreater}}{=} \)-equivalent.

\textbf{Definition 9} The predicate \( \text{UIN} : \mathcal{L}(\text{\textless \text{msc}\textgreater}) \to \mathcal{P} \) is for all \( ch \in \mathcal{L}(\text{\textless \text{msc}\textgreater}) \) defined by

\[
\text{UIN}(ch) = \bigwedge_{i \in \text{AllInsts}(ch)} \#_{i \overset{\text{\textless \text{inst\_def}\textgreater}}{=} i} \text{AllInsts}(ch) \leq 1
\]

20
4.3 Chart Interface Consistency

A chart consists of a chart head and a chart body. In turn, the chart head consists of a chart name and optionally a chart interface. The chart interface is considered an overview of the instances which are specified within the chart body. The following requirement for chart interfaces is stated in [IT95b]:

*The information given in the instance list in the chart interface, if present, must be consistent with the information specified in the instance heads* [IT95b].

Consider the following chart.

```msc
example; inst i: process kn, j: process x;
    instance i: block kn;
    ...
    endinstance;
    instance j;
    ...
    endinstance;
endmsc;
```

The head of instance *i* states that it is of type *block* whereas the chart interface states that instance *i* is of type *process*. The information concerning the type of instance *i* is not consistent. The chart interface also specifies that instance *j* is of type *process* and of kind *x*. However, the instance head does not confirm this information.

In general, the instance heads define the information of the instances and the chart interface is considered an overview thereof. However, it is not required that the chart interface provides for a complete overview of the information given by the instance heads. This is illustrated by the following correct chart.

```msc
example; inst j;
    instance i;
    ...
    endinstance;
    instance j: process x;
    ...
    endinstance;
endmsc;
```

The chart interface contains very limited information. It only signals that there is an instance with name *j*. It does not say anything about other instances which may be present in the chart body.
such as instance \(i\). Although, in the definition of instance \(j\), both the kind name \(x\) and the type process are specified, the chart interface does not give any of those.

Besides the requirement for consistency between chart interface and instance heads, there is also the additional requirement that the chart interface must not contain more than one entry for an instance. Since the following chart contains two entries for instance \(i\), it violates this requirement.

```msc
example; inst i, i:process kn;
    instance i:process kn;
    ...
endinstance;
endmsc;
```

This requirement is also formalized in this section.

*It is not allowed that the instance list in the chart interface contains more than one entry for an instance in the chart body* [Ren94, section 4.3, page 14].

The information the chart interface and the instance heads give about an instance may consist of the instance name and the instance kind. In turn, the instance kind consists of a kind name and optionally a kind denominator. A type `InstInfo` is defined which structures this information in a tuple. In the definition of `InstInfo` the type `KindInfo`, which structures the information of the instance kind in a tuple, is used.

**Definition 10** The types `KindInfo` and `InstInfo` are given by

\[
\begin{align*}
\text{KindInfo} & = \mathcal{L}(\langle \text{kn}\rangle) \times \perp \langle \text{type}\rangle \\
\text{InstInfo} & = \mathcal{L}(\langle \text{inst name}\rangle) \times \text{KindInfo}_\perp
\end{align*}
\]

For a nonterminal \(X\), the notation \(\mathcal{L}_\perp(X)\) is used to indicate the extension of the language \(\mathcal{L}(X)\) with the "fresh" element \(\perp (\perp \not\in \mathcal{L}(X))\). For an arbitrary set \(S\), the notation \(S_\perp\) is used to indicate the extension of the set \(S\) with the "fresh" element \(\perp\). In the definition of `KindInfo` the element \(\perp\) is used to signal the situation in which no kind denominator is present and in the definition of `InstInfo` it is used to signal that no instance kind is present.

First, the information given in the chart interface will be considered. The chart interface consists of the keyword `inst` followed by an instance list. This instance list consists of a number of instance names optionally followed by an instance kind. A function `InfoKind` is defined which structures the kind name and kind denominator of an instance kind in a tuple of the type `KindInfo`. A function `InfoList` is defined which collects the information given in an instance list. Thereafter, using the previously mentioned function, a function `InfoInterface` is defined which associates to a chart the information given in its chart interface.
Definition 11 The function $\text{InfoKind} : \mathcal{L}(\text{<inst list>}) \rightarrow \text{KindInfo}$ is for all $kn \in \mathcal{L}(\text{<kn>})$ and $type \in \mathcal{L}(\text{<type>})$ defined by

\[
\begin{align*}
\text{InfoKind}(kn) &= (kn, \bot) \\
\text{InfoKind}(type kn) &= (kn, \text{type})
\end{align*}
\]

The function $\text{InfoList} : \mathcal{L}(\text{<inst list>}) \rightarrow M(\text{InstInfo})$ is for $\text{name} \in \mathcal{L}(\text{<name>})$, $\text{kind} \in \mathcal{L}(\text{<inst kind>})$ and $\text{list} \in \mathcal{L}(\text{<inst list>})$ defined by

\[
\begin{align*}
\text{InfoList(name)} &= [(\text{name}, \bot)] \\
\text{InfoList(name:kind)} &= [(\text{name}, \text{InfoKind(kind)})] \\
\text{InfoList(name, list)} &= [(\text{name}, \bot)] \cup \text{InfoList(list)} \\
\text{InfoList(name:kind, list)} &= [(\text{name}, \text{InfoKind(kind)})] \cup \text{InfoList(list)}
\end{align*}
\]

The function $\text{InfoInterface} : \mathcal{L}(\text{<msc>}) \rightarrow M(\text{InstInfo})$ is for $\text{name} \in \mathcal{L}(\text{<name>})$, $\text{body} \in \mathcal{L}(\text{<body>})$ and $\text{list} \in \mathcal{L}(\text{<inst list>})$ defined by

\[
\begin{align*}
\text{InfoInterface(msc name; body endmsc;)} &= \emptyset \\
\text{InfoInterface(msc name; list body endmsc;)} &= \text{InfoList(list)}
\end{align*}
\]

In the same way as the information from a chart interface is collected, the information given in an instance head is obtained. The function $\text{InfoInstHead}$ associates to the head of an instance the information provided within that head.

Definition 12 The function $\text{InfoInstHead} : \mathcal{L}(\text{<inst head>}) \rightarrow \text{InstInfo}$ is for all $\text{name} \in \mathcal{L}(\text{<name>})$ and $\text{kind} \in \mathcal{L}(\text{<inst kind>})$ defined by

\[
\begin{align*}
\text{InfoInstHead(name;)} &= (\text{name}, \bot) \\
\text{InfoInstHead(name decomposed;)} &= (\text{name}, \bot) \\
\text{InfoInstHead(name kind;)} &= (\text{name}, \text{kind}) \\
\text{InfoInstHead(name kind decomposed;)} &= (\text{name}, \text{kind})
\end{align*}
\]

Then the information in the heads of the instances of an arbitrary chart $\text{ch}$ is given by the formula $\{\text{InfoInstHead(InstHead(i))} \mid i \in \text{AllInsts(ch)}\}$. The function $\text{InstHead}$ (see Appendix B) associates to an instance its head. Note that it is not necessary to use a multiset to collect the information given by the heads of the instances since the syntax requirement from the previous section guarantees that each element of the multiset occurs only once. Since multisets were needed to collect the information in the chart interface, multisets are used also to collect the information in the instance heads. Thereby, the information from the chart interface and the information from the instance heads can easily be compared.

Definition 13 The function $\text{InfoInstHeads} : \mathcal{L}(\text{<chart>}) \rightarrow M(\text{InstInfo})$ is for all $\text{ch} \in \mathcal{L}(\text{<msc>})$ defined by

\[
\text{InfoInstHeads(ch)} = \{\text{InfoInstHead(InstHead(i))} \mid i \in \text{AllInsts(ch)}\}
\]
Example 14 The collection of information from the instance heads and the chart interface is illustrated by the chart below. For this chart, the information provided by the instance heads consists of the tuples \((i, (x, \bot)), (j, \bot)), \text{and} (k, (x, \text{process}))\). The chart interface contains the tuples \((i, \bot)), (j, \bot)), \text{and} (k, (x, \bot))\).

```plaintext
msc example; inst i, j, k:x;
    instance i:x;
    ...
    endinstance;
    instance j;
    ...
    endinstance;
    instance k:process x;
    ...
    endinstance;
endmsc;
```

Next, the consistency between the information from the instance heads and the information from the chart interface is defined. It is required that the information given by the chart interface is a submultiset of the information given by the instance heads.

**Definition 15** The relation \(\leq^\text{KindInfo}\) \(\subseteq\text{KindInfo}_\bot \times \text{KindInfo}_\bot\) is for all \(k \in \text{KindInfo}_\bot, \text{kn}_1, \text{kn}_2 \in \mathcal{L}(\langle\text{name}\rangle)\) and \(t_1, t_2 \in \mathcal{L}_\bot(\langle\text{type}\rangle)\) defined by

1) \(\bot \leq^\text{KindInfo}_\bot k\)

2) \(k \leq^\text{KindInfo}_\bot \bot\) if and only if \(k = \bot\)

3) \((\text{kn}_1, t_1) \leq^\text{KindInfo}_\bot (\text{kn}_2, t_2)\) if and only if \(\text{kn}_1 = \text{kn}_2 \land (t_1 = \bot \lor t_1 = t_2)\)

The relation \(\leq^\text{InstInfo}\) \(\subseteq\text{InstInfo} \times \text{InstInfo}\) is for all \(i_1, i_2 \in \mathcal{L}(\langle\text{name}\rangle)\) and \(k_1, k_2 \in \text{KindInfo}_\bot\) defined by

\[ (i_1, k_1) \leq^\text{InstInfo} (i_2, k_2) \text{ if and only if } i_1 = i_2 \land k_1 \leq^\text{KindInfo}_\bot k_2 \]

The relation \(\subseteq^\text{ InstInfo} \subseteq \mathcal{M}^\text{InstInfo} \times \mathcal{M}^\text{InstInfo}\) is for all \(X, Y \in \mathcal{M}^\text{InstInfo}\) defined by

\[ X \subseteq^\text{InstInfo} Y \text{ if and only if } \bigwedge_{x \in X} \bigvee_{y \in Y} x \leq^\text{InstInfo} y \]

24
So, the requirement that the chart interface information must be consistent with the information from the instance heads is formalized as follows: $\text{InfoInterface}(ch) \subseteq \text{InfoInstHeads}(ch)$.

**Example 16** Denote the chart example from the previous example by $ch$. The information given by the chart interface is given by the multiset

$$\text{InfoInterface}(ch) = [(i, \bot), (j, \bot), (k, (x, \bot))]$$

and the information from the instance heads by

$$\text{InfoInstHeads}(ch) = [(i, (x, \bot)), (j, \bot), (k, (x, \text{process}))]$$

The chart interface is consistent with the instance heads if for every entry in the chart interface there is an instance head which contains (at least) that information. For example, the chart interface has an entry for instance $k$ with kind name $x$. This information is consistent with the information in the head of instance $k$. It is allowed that the instance head contains more information than the chart interface. Consistency of the chart interface and the instance heads is obtained from the following observations:

1) $(i, \bot) \leq (i, (x, \bot))$ which follows from the definition of $\leq$ and from $\bot \leq (x, \bot)$;

2) $(j, \bot) \leq (j, \bot)$ which follows from the definition of $\leq$ and from $\bot \leq \bot$;

3) $(k, (x, \bot)) \leq (k, (x, \text{process}))$ which follows from the definition of $\leq$ and from $(x, \bot) \leq (x, \text{process})$.

Next, the requirement that the chart interface can have at most one entry for every instance in the chart body is formalized. An approach similar to the approach from the previous section is used. An equivalence relation on the tuples from $\text{InstInfo}$ which relates any two tuples with the same instance name is defined. This equivalence relation is denoted by $\equiv_{\text{InstInfo}}$.

**Definition 17** The relation $\equiv_{\text{InstInfo}} \subseteq \text{InstInfo} \times \text{InstInfo}$ is for all $i_1, i_2 \in \mathcal{L}(<\text{inst name}>)$ and $k_1, k_2 \in \text{KindInfo}$ defined by

$$(i_1, k_1) \equiv_{\text{InstInfo}} (i_2, k_2) \text{ if and only if } i_1 = i_2$$

The syntax requirement is formulated in terms of the number of occurrences of tuples which are equivalent with respect to $\equiv_{\text{InstInfo}}$. Expressing that there can be at most one entry in the chart interface for every instance in the chart body amounts to verifying whether $t_{\text{InstInfo}}(\text{InfoInterface}(ch)) \leq 1$ for every tuple $t \in \text{InfoInterface}(ch)$. The syntax requirement is then formulated as follows.
Definition 18 The predicate \( ChInterface : \mathcal{L}(\text{<msg>}) \rightarrow \mathcal{B} \) is for all \( ch \in \mathcal{L}(\text{<msg>}) \) defined by

\[
ChInterface(ch) = \left( \text{InfoInterface}(ch) \subseteq \text{InfoInstHeads}(ch) \right) \land \\
\left( \text{In"lnfo}(\text{InfoInterface}(ch)) \leq 1 \right)
\]

4.4 Rules for messages

4.4.1 Abstract messages

A message is completely determined by its sender instance, its receiver instance and its message identifier. A message can be either an internal message, i.e. a communication between instances, or an external message, i.e. a communication with the environment. A message can be represented by a triple which consists of the name of the sender instance, the name of the receiver instance and the message identifier. Such a triple representing a message will be called an abstract message. A type \( \text{Msg} \) is defined from which the elements represent abstract messages.

Definition 19 The type \( \text{Msg} \) is defined by

\[
\text{Msg} = \mathcal{L}(\text{<address>}) \times \mathcal{L}(\text{<address>}) \times \mathcal{L}(\text{<msgid>})
\]

The first element of such a triple gives the name of the sending instance, the second element the name of the receiving instance, and the third element the identification of the message. In case of a message output event the sender instance name is the name of the instance the message output is specified on. In case of a message input event the receiver instance name is the name of the instance the message input is specified on. To obtain the address specification and the message identifier from a given communication event the functions \( \text{Address} \) and \( \text{MsgId} \) are used. Their definitions can be found in Appendix B.

Note that the type \( \text{Msg} \) also contains elements which represent messages that are sent from the environment to the environment. For example, the triple \((\text{env},\text{env},m)\) represents a message with message identifier \( m \) that is sent from the environment to the environment. A message that is both sent by and received from the environment can not be specified within a chart. On basis of the instance definition a communication event is specified within, it is possible to determine the abstract message the communication event refers to. The instance definition is needed in order to determine the name of the sender instance in case of a message output and the name of the receiver instance in case of a message input. Note that in fact only the name of the instance needs to be known.
Definition 20 Let $i \in \mathcal{L}(<\text{inst def}>)$, The function $\text{Message}(i) : \mathcal{L}(<\text{out}> | <\text{in}>) \rightarrow \text{Msg}$ is for all $\text{out} \in \mathcal{L}(<\text{out}>)$ and $\text{in} \in \mathcal{L}(<\text{in}>)$ defined by

\[
\text{Message}(i)(\text{out}) = (\text{InstName}(i), \text{Address}(\text{out}), \text{MsgId}(\text{out})) \\
\text{Message}(i)(\text{in}) = (\text{Address}(\text{in}), \text{InstName}(i), \text{MsgId}(\text{in}))
\]

Example 21 Let $i$ denote an instance definition with name $\text{iname}$, i.e. $\text{InstName}(i) = \text{iname}$. The message output $\text{out} m$ to receiver; and the message input $\text{in} m$ from sender; refer to the abstract messages

\[
\text{Message}(i) (\text{out} m \text{ to receiver};) = (\text{iname}, \text{receiver}, m)
\]

and

\[
\text{Message}(i) (\text{in} m \text{ from sender};) = (\text{sender}, \text{iname}, m)
\]

respectively.

The rules for messages which will be discussed in this section express properties such as the unambiguous connection of message outputs and message inputs, the completeness of message specification, and the order in which message sending and message reception must be dealt with. First, a syntax requirement is discussed which expresses that the instances referenced by the communication events must be declared.

4.4.2 Referenced instances must be declared

The instances that are referenced by the communication events are given by the address specification of the communication events of the chart. The address specification of the message outputs to the environment and the message inputs from the environment do not reference an instance. Therefore, only the internal messages need to be considered.

The instance head of an instance definition contains the definition of an instance name. The declared instances within a chart are the instances specified within that chart. The following syntax requirement is formulated.

Only declared instances may be referenced [IT93, section 2.2, page 3].

First, a function $\text{DeclInstNames}$ is defined which determines the names of all instances specified within a chart. These are exactly the names of the declared instances. This follows from the uniqueness rule for instances (see Section 4.2).

Definition 22 The function $\text{DeclInstNames} : \mathcal{L}(<\text{msc}>) \rightarrow \mathcal{P}(\mathcal{L}(<\text{inst name}>))$ is for all $\text{ch} \in \mathcal{L}(<\text{msc}>)$ defined by

\[
\text{DeclInstNames}(\text{ch}) = \{\text{InstName}(i) | i \in \text{AllInsts}(\text{ch})\}
\]
Example 23 The names of the instances which are declared in the chart from Figure 2.1 are given by \( DeclInstNames(ch) = \{i_1, i_2, i_3, i_4\} \).

Next, a function \( InternalAddrSpec \) is defined which associates to a chart the names of all instances referenced by the internal communication events specified on that chart. This definition uses the functions \( Outputs \) and \( Inputs \) (see Appendix B) which associate to a syntactic category such as a chart the multiset containing all occurrences of the message output events and the message input events respectively.

Definition 24 The function \( InternalAddrSpec : \mathcal{L}(<msc>) \rightarrow \mathcal{P}(\mathcal{L}(<inst\ name>)) \) is for all \( ch \in \mathcal{L}(<msc>) \) defined by

\[
InternalAddrSpec(ch) = \{ \text{Address}(com) \mid com \in Outputs(ch) \cup Inputs(ch) \land \text{Address}(com) \neq env \}
\]

Example 25 The names of the instances referenced by the communication events from the chart in Figure 2.1 are given by \( InternalAddrSpec(ch) = \{i_1, i_2, i_3, i_4\} \).

Then the formulation of the syntax requirement is as follows.

Definition 26 The predicate \( RefInsts : \mathcal{L}(<msc>) \rightarrow \mathcal{P} \) is for all \( ch \in \mathcal{L}(<msc>) \) defined by

\[
RefInsts(ch) = (InternalAddrSpec(ch) \subseteq DeclInstNames(ch))
\]

Example 27 From the previous two examples it is known that

\[
DeclInstNames(ch) = \{i_1, i_2, i_3, i_4\}
\]

and

\[
InternalAddrSpec(ch) = \{i_1, i_2, i_3, i_4\}
\]

From this it follows immediately that

\[
InternalAddrSpec(ch) \subseteq DeclInstNames(ch)
\]

Clearly, the chart \( ch \) respects the syntax requirement.

4.4.3 Uniqueness rule for messages

An internal message is split into two events: a message output and a message input. In this section a naming rule for communication events is considered which guarantees that there is at most one way to connect message outputs to message inputs and vice versa. Consider for example the following chart.
Within this chart it is not clear which message output corresponds to which message input. Either one of the two charts shown in Figure 4.2 can be associated to this textual description. To avoid this situation a syntax requirement is formulated which guarantees that there are no two message outputs which refer to the same abstract message and also that there are no two message inputs which refer to the same abstract message.

![Figure 4.2: Charts corresponding with the textual description](image)

Recommendation Z.120 [IT93] states the following syntax requirement.

For messages exchanged between instances the following must hold: to each message output at most one corresponding message input has to be specified and vice versa [IT93, section 4.3, page 12].

Two message output events which are defined on different instances cannot concern the same message since they have different sender instances. Also, two message input events which are defined on different instances cannot concern the same message since they have different receiver instances. Therefore, the syntax requirement is reformulated as follows.

*On an instance there must not be two or more message outputs with the same address specification and the same message identifier. On an instance there must not be two or more message inputs with the same address specification and the same message identifier.*
First, an equivalence relation $\sim_{\text{out}}$ is defined on message output events. Two message output events are output equivalent (or $\sim_{\text{out}}$-equivalent) if and only if both the message identifier and the address specification are identical.

**Definition 28** The relation $\sim_{\text{out}} \subseteq \mathcal{L}(\text{<out>}) \times \mathcal{L}(\text{<out>})$ is for all $\text{out}, \text{out}' \in \mathcal{L}(\text{<out>})$ defined by

\[
\text{out} \sim_{\text{out}} \text{out}' \text{ if and only if } \text{MsgId(out)} = \text{MsgId(out')} \land \text{Address(out)} = \text{Address(out')}
\]

From the definition it is clear that this relation is an equivalence on message outputs. Two message outputs which are specified on the same instance refer to the same abstract message if and only if they are $\sim_{\text{out}}$-equivalent. Two message outputs which are specified on different instances can not refer to the same abstract message, but they can be $\sim_{\text{out}}$-equivalent. With the same approach as was used for the uniqueness of instance names, the syntax requirement is formulated as follows.

**Definition 29** The predicate $\text{UMO} : \mathcal{L}(\text{<inst def>}) \rightarrow \mathcal{B}$ is for all $i \in \mathcal{L}(\text{<inst def>})$ defined by

\[
\text{UMO}(i) = \bigwedge_{\text{out} \in \mathcal{O}(\text{Outputs}(i))} \left[ \#_{\sim_{\text{out}}} (\text{Outputs}(i)) \leq 1 \right]
\]

The function $\mathcal{O}(\text{Outputs}(\text{<inst def>}))$ collects all message output events from an instance definition in a multiset.

For the uniqueness of message inputs the approach is the same as for the uniqueness of message outputs. First an equivalence relation $\sim_{\text{in}}$ on message inputs is defined.

**Definition 30** The relation $\sim_{\text{in}} \subseteq \mathcal{L}(\text{<in>}) \times \mathcal{L}(\text{<in>})$ is for all $\text{in}, \text{in}' \in \mathcal{L}(\text{<in>})$ defined by

\[
\text{in} \sim_{\text{in}} \text{in}' \text{ if and only if } \text{MsgId(in)} = \text{MsgId(in')} \land \text{Address(in)} = \text{Address(in')}
\]

The syntax requirement is formulated as follows.

**Definition 31** The predicate $\text{UMI} : \mathcal{L}(\text{<inst def>}) \rightarrow \mathcal{B}$ is for all $i \in \mathcal{L}(\text{<inst def>})$ defined by

\[
\text{UMI}(i) = \bigwedge_{\text{in} \in \mathcal{O}(\text{Inputs}(i))} \left[ \#_{\sim_{\text{in}}} (\text{Inputs}(i)) \leq 1 \right]
\]

The function $\mathcal{O}(\text{Inputs}(\text{<inst def>}))$ collects all message input events from an instance definition in a multiset.

Combining these two yields the following predicate.

**Definition 32** The predicate $\text{UniqueMessages} : \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{B}$ is for all $\text{ch} \in \mathcal{L}(\text{<msc>})$ defined by

\[
\text{UniqueMessages}(\text{ch}) = \bigwedge_{i \in \text{AllInsts}(\text{ch})} \text{UMO}(i) \land \text{UMI}(i)
\]
4.4.4 Completeness of messages

The syntax requirement concerning the uniqueness of messages from the previous section guarantees that there is at most one way to connect message outputs and message inputs. The syntax requirement introduced in this section guarantees the existence of such a connection, i.e. that there is at least one way to connect message outputs and message inputs. Of course message outputs to the environment and message inputs from the environment need not be considered. Together the syntax requirements express that there is exactly one way to connect message outputs and message inputs. Consider the following chart.

```
msc example;
  instance i;
    out m1 to env;
    out m2 to j;
  endinstance;
  instance j;
    in m3 from env;
    in m4 from i;
  endinstance;
endmsc;
```

Within this chart there are four communication events. Two of these specify a communication with the environment. The other two specify a communication between instances. The syntax requirement for uniqueness of messages is satisfied by this chart. Consider the message m2 sent by instance i to instance j. For this message only the message output is specified. There is no corresponding message input. The following syntax requirement is formulated.

*For messages exchanged between instances the following must hold: to each message output a corresponding message input has to be specified and vice versa [IT93, section 4.3, page 12].*

This requirement can be combined easily with the requirement from the previous section. Together they express that there is a 1-1-correspondence between message outputs and message inputs. Note that this requirement only applies to messages which are exchanged between instances. Messages sent to and received from the environment need not be considered since they do not have a corresponding message input and message output respectively.

Next, predicates CorOut and CorIn are defined which determine whether there is a corresponding message input for each message output, and vice versa. The uniqueness rule for messages (see Section 4.4.3) ascertains that such a correspondence, if it exists, is unique.

**Definition 33** The predicate CorOut : \( L(<\text{msc}>) \rightarrow \mathcal{B} \) is for all \( ch \in L(<\text{msc}>) \) defined by

\[
\text{CorOut}(ch) = \bigwedge_{i \in \text{AllInst}(ch)} \bigvee_{j \in \text{AllInst}(ch)} \bigwedge_{o \in \text{Outputs}(i)} \bigwedge_{i \in \text{Inputs}(j)} \text{Message}(i)(out) = \text{Message}(j)(in)
\]
The predicate $\text{CorIn} : \mathcal{L}(\text{<msg>}) \rightarrow \mathcal{I}$ is for all $ch \in \mathcal{L}(\text{<msg>})$ defined by

$$\text{CorIn}(ch) = \bigwedge_{i \in \text{AllInsts}(ch)} \bigvee_{j \in \text{AllInsts}(ch)} \bigwedge_{in \in \text{Inpu}(i)} \bigvee_{out \in \text{Outpu}(j)} \text{Message}(i)(\text{in}) = \text{Message}(j)(\text{out})$$

Combining these two predicates the formalization of the syntax requirement is as follows.

**Definition 34** The predicate $\text{Correspondence} : \mathcal{L}(\text{<msg>}) \rightarrow \mathcal{I}$ is for all $ch \in \mathcal{L}(\text{<msg>})$ defined by

$$\text{Correspondence}(ch) = \text{CorOut}(ch) \land \text{CorIn}(ch)$$

### 4.5 Rules for conditions

#### 4.5.1 Abstract conditions

A condition describes either a global system state (global condition) referring to all instances contained in the chart or a state referring to a (non-empty) subset of all instances (nonglobal condition). 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.

A condition is completely determined by its name and the set of the names of the instances the condition references. Therefore, a condition with name $cn$ which refers to all instances with a name from the set $I$ is represented by the tuple $(cn,I)$. Such a tuple will be called an abstract condition.

**Definition 35** The type $\text{Condition}$ is defined by

$$\text{Condition} = \mathcal{L}(\text{<cn>}) \times \mathcal{P}(\mathcal{L}(\text{<inst name>}))$$

Given the chart and the instance a condition event is defined on, it is possible to determine the abstract condition the condition event refers to. Therefore, functions $\text{CondName}$ (see Appendix B) and $\text{CondRefInsts}$ are defined which determine the name of a condition event and the set of the names of the instances the condition refers to. By taking the enclosing instance definition and chart of a condition event into consideration it is possible to determine these instance names. This is necessary in order to determine the names of the instances of a condition which is shared by all instances, since those are not specified explicitly by the condition event. In the definition of the function $\text{CondRefInsts}$ the function $\text{SharedInstSet}$, which associates to a shared instance list the set containing all instance names listed, is used.
Definition 36 The function $\text{SharedInstSet} : \mathcal{L}(<\text{shared inst list}>) \to \mathcal{P}(\mathcal{L}(<\text{inst name}>))$ is for all $\text{instname} \in \mathcal{L}(<\text{inst name}>)$ and $\text{list} \in \mathcal{L}(<\text{shared inst list}>)$ defined by

$$\text{SharedInstSet}($$instname$$) = \{instname\}$$

$$\text{SharedInstSet}($$instname$$, list) = \{instname\} \cup \text{SharedInstSet}($$list$$)$$

Let $ch \in \mathcal{L}(<\text{msg}>)$ and $i \in \mathcal{L}(<\text{inst def}>)$ be given. The function $\text{CondReflnsts}(ch,i) : \mathcal{L}(<\text{condition}>) \to \mathcal{P}(\mathcal{L}(<\text{inst name}>))$ is for all $\text{cname} \in \mathcal{L}(<\text{cn}>)$ and all $\text{list} \in \mathcal{L}(<\text{shared inst list}>)$ defined by

$$\text{CondReflnsts}(ch,i)(condition\ cname;) = \{\text{InstName}(i)\}$$

$$\text{CondReflnsts}(ch,i)(condition\ cname\ shared\ all;) = \text{DeclInstNames}(ch)$$

$$\text{CondReflnsts}(ch,i)(condition\ cname\ shared\ list;) = \{\text{InstName}(i)\} \cup \text{SharedInstSet}($$list$$)$$

Next, a function $\text{Cond}$ is defined which, given a chart and an instance definition, associates to a condition event the abstract condition it refers to.

Definition 37 Let $ch \in \mathcal{L}(<\text{msg}>)$ and $i \in \mathcal{L}(<\text{inst def}>)$. The function $\text{Cond}(ch,i) : \mathcal{L}(<\text{condition}>) \to \text{Condition}$ is for all $c \in \mathcal{L}(<\text{condition}>)$ defined by

$$\text{Cond}(ch,i)(c) = (\text{CondName}(c), \text{CondReflnsts}(ch,i)(c))$$

Example 38 Let $ch$ denote the textual description of the chart shown in Figure 2.3 and $I$ denote the definition of instance $i$ therein. The condition events defined in instance definition $I$ refer to the following abstract conditions:

1) $\text{Cond}(ch,I)(\text{condition } C2;) = (C2, \{i\});$

2) $\text{Cond}(ch,I)(\text{condition } C1\ \text{shared all;};) = (C1, \{i,j,k\});$

3) $\text{Cond}(ch,I)(\text{condition } C3\ \text{shared } k;) = (C3, \{i,k\}).$

In the remainder of this section two syntax requirements for conditions will be considered. These are the reference to declared instances, and the completeness of condition specification.

4.5.2 Referenced instances must be declared

If a condition event contains a shared instance list, it references the instances listed therein. The syntax requirement from Section 4.4.2 is repeated for conditions.

Only declared instances may be referenced [IT93, section 2.2, page 3].
A function \textit{SpecifiedInsts} is defined which associates to a condition event the names of all instances listed in its shared instance list. If the condition event does not have a shared instance list, the empty set is taken as a value.

\textbf{Definition 39} The function \textit{SpecifiedInsts} : $\mathcal{L}(\text{<condition>}) \rightarrow \mathcal{P}(\mathcal{L}(\text{<inst name>}))$ is for all $cn \in \mathcal{L}(\text{<cn>})$ and $list \in \mathcal{L}(\text{<shared inst list>})$ defined by

\[
\begin{align*}
\text{SpecifiedInsts}(\text{condition } cn; ) &= \emptyset \\
\text{SpecifiedInsts}(\text{condition } cn \text{ shared all;} ) &= \emptyset \\
\text{SpecifiedInsts}(\text{condition } cn \text{ shared list;} ) &= \text{SharedInstSet(list)}
\end{align*}
\]

The instances to which the conditions of a chart \textit{ch} refer are then given by the set

\[
\{\text{SpecifiedInsts}(c) \mid c \in \text{CondEvents}(ch)\}
\]

The function \textit{CondEvents} associates to a chart the collection of all its condition events (see Appendix B). Then the formulation of the syntax requirement is as follows.

\textbf{Definition 40} The predicate \textit{RejInsts'} : $\mathcal{L}(\text{<msc>}) \rightarrow \mathcal{B}$ is for all $ch \in \mathcal{L}(\text{<msc>})$ defined by

\[
\text{RejInsts'}(ch) = \left(\{\text{SpecifiedInsts}(c) \mid c \in \text{CondEvents}(ch)\} \subseteq \text{DeclInstNames}(ch)\right)
\]

\subsection*{4.5.3 Completeness of condition definitions}

In this section a syntax requirement for the completeness of condition definitions is discussed and formalized. Consider the following example.

\begin{verbatim}
msc example;
  instance i;
    action a;
    condition C shared j;
  endinstance;
  instance j;
    action b;
  endinstance;
endmsc;
\end{verbatim}

On instance \textit{i} a condition event is specified which refers to the abstract condition (\textit{C}, \{ \textit{i,j} \}). From the textual description it is also clear that the global system state described by this condition occurs somewhere after the execution of a local action \textit{a}. On instance \textit{j} it is not specified whether that system state occurs before or after the execution of action \textit{b}, although it is specified by instance \textit{i} that the system state deals with instance \textit{j} as well. This means that either one of the two charts from
Figure 4.3 could be associated to the textual description. In order to disallow such an ambiguity the following syntax requirement is formulated in Recommendation Z.120.

![Figure 4.3: Two charts resulting from incomplete condition definitions](image)

To each instance name 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 \( b \) [IT93, section 4.4, page 13].

Unfortunately, this formulation does not cover the complete set of problem cases. This can be illustrated by the following chart.

```
msc example;
  instance i;
    condition C shared all;
  endinstance;
  instance j;
    action a;
  endinstance;
endmsc;
```

This condition event is not covered by the syntax requirement as formulated above, since that does not consider conditions which are shared by all instances of a chart by means of the reserved keywords `shared all`. However, for the same reasons this chart should be disallowed. This leads to the following reformulation of the requirement.
On every instance referenced by a condition event there has to be a corresponding condition event on that instance (no reference).

For each instance that is referenced by a condition there has to be a condition event specified on that instance. For example, if a chart specifies a reference to a condition \((C,I)\), then all instances of the set \(I\) must have a condition event referring to this condition. Since a condition can be specified more than once within a chart this is not specific enough. Consider for example the following chart.

\[
\text{msc nextexample;}
\text{instance i;}
\text{condition C shared j;}
\text{action a;}
\text{condition C shared j;}
\text{endinstance;}
\text{instance j;}
\text{condition C shared i;}
\text{endinstance;}
\text{endmsc;}
\]

On instance \(i\) two condition events which reference the condition \((C, \{i,j\})\) are specified, whereas instance \(j\) has only one condition event which references this condition. This chart respects the above requirement. However, in the graphical representation it is not allowed to draw such a chart. The number of occurrences of the references to a condition should be taken into account. The syntax requirement for the completeness of condition specification is therefore rephrased as follows.

The number of references of an instance to a condition must be equal for all instances the condition refers to (no reference).

By the syntax requirement the number of occurrences of the conditions two instances have in common must be equal. This is expressed by the predicate \(\text{Compl}\).

**Definition 41** The predicate \(\text{Compl} : \mathcal{L}(\langle \text{msc} \rangle) \rightarrow \mathbb{B}\) is for all \(ch \in \mathcal{L}(\langle \text{msc} \rangle)\) defined by

\[
\text{Compl}(ch) = \bigwedge_{i,j \in \text{AllInsts}(ch)} \bigwedge_{c \in \text{CondEvents}(i)} \bigwedge_{\text{InstName}(j) \in \text{CondRefInsts}(ch,i)(c)} \#_{\text{Cond}(ch,i)(c)}([\text{Cond}(ch,i)(c') | c' \in \text{CondEvents}(i)]) = \#_{\text{Cond}(ch,j)(c)}([\text{Cond}(ch,j)(c'') | c'' \in \text{CondEvents}(j)])
\]

**Example 42** The requirement is illustrated for chart \text{nextexample}. The abstract conditions the instances have in common are given by the set \\{\{(C, \{i,j\})\}\}. However, this abstract condition occurs twice on instance \(i\), whereas it only occurs once on instance \(j\). The syntax requirement is violated.
4.6 Rules for process creation

A create event specifies the name of the instance that is created, and optionally, a parameter list. In Appendix B a function \textit{CreateName} is defined which associates to a create event the name of the instance that is created. For the syntactic categories \texttt{<inst body>}, \texttt{<inst def>}, and \texttt{<msc>} the function \textit{CreateEvents} (see Appendix B) collects the create events specified on these syntactic objects. This function will be used in the formalization of the first syntax requirement. In this section syntax requirements for the reference to instances, the uniqueness of creation, and the type of creating and created instances are discussed and formalized.

4.6.1 Reference rule for instances

Each create event references exactly one instance, the instance to be created. As was the case for messages and conditions, the create events may only reference instances which are declared.

\textit{To each create there must be a corresponding instance with the same name} [IT93, section 4.7, page 18].

Using the functions \textit{CreateName} and \textit{CreateEvents} from Appendix B the instances referenced by the create events of a chart \texttt{ch} are given by \{CreateName(\textit{cr}) \mid \textit{cr} \in \textit{CreateEvents}(\texttt{ch})\}. Using the function \textit{DeclInstNames} from Section 4.4.2 the formulation of the syntax requirement for referencing declared instances is as follows.

\textbf{Definition 43} The predicate \textit{RefInsts}$''$ : \(\mathcal{L}(\texttt{<msc>}) \rightarrow \mathcal{B}\) is for all \texttt{ch} \in \mathcal{L}(\texttt{<msc>}) defined by

\[
\textit{RefInsts}$''$(\texttt{ch}) = (\{\text{CreateName}(\textit{cr}) \mid \textit{cr} \in \text{CreateEvents}(\texttt{ch})\} \subseteq \text{DeclInstNames}(\texttt{ch}))
\]

4.6.2 Uniqueness of process creation

In this section a syntax requirement is formulated for the uniqueness of created instances. This requirement states that an instance can be created only once. This requirement can be illustrated with the following charts.

\begin{verbatim}
msc example;
instance i;
create j(p);
action a;
create j(q);
endinstance;
instance j;
action b;
msc example2;
instance k;
create l(p);
endinstance;
instance l;
action a;
endinstance;
instance m;
\end{verbatim}

37
Instance $i$ creates instance $j$ twice. In the second example instance $l$ is also created twice, once by instance $k$ and once by instance $m$. Recommendation Z.120 states the uniqueness rule for process creation as follows.

\textit{An instance can be created only once, i.e. within one chart two or more creates with the same name must not appear [IT93, section 4.7, page 18].}

First, an equivalence $\equiv_{<\text{create}>}$ is defined which identifies two create events if and only if they refer to the same instance.

**Definition 44** The relation $\equiv_{<\text{create}>} \subseteq \mathcal{L}(<\text{create}>) \times \mathcal{L}(<\text{create}>)$ is for all $cr_1, cr_2 \in \mathcal{L}(<\text{create}>)$ defined by

$$cr_1 \equiv_{<\text{create}>} cr_2 \text{ if and only if } \text{CreateName}(cr_1) = \text{CreateName}(cr_2)$$

The formalization of the syntax requirement is as follows.

**Definition 45** The predicate $\text{UniqueCreation} : \mathcal{L}(<\text{msc}>) \rightarrow \mathcal{B}$ is for all $ch \in \mathcal{L}(<\text{msc}>)$ defined by

$$\text{UniqueCreation}(ch) = \bigwedge_{cr \in \text{CreateEvents}(ch)} \mu_{\equiv_{<\text{msc}>}}(\text{CreateEvents}(ch)) \leq 1$$

### 4.6.3 The type of creating and created instances

With Message Sequence Charts instances of type block, service, or system may neither be created nor create another instance. For instances without type information there are no such limitations. Instances with type process may create another instance and may be created.

\textit{An instance on which a create event is specified must be of type process, if its type is specified. The instance name has to refer to an instance with type process, if its type is specified [IT93, section 4.7, pages 17-18].}

This requirement expresses that both the creating instance, i.e. the instance on which the create event is specified, and the created instance, i.e. the instance the create event refers to, must be — if a type is present — of type process. A predicate $\text{typeok}$ is defined which determines whether an instance head has no type at all or if it is of type process.
Definition 46 The predicate $\text{typeok} : \mathcal{L}(\text{<inst head>}) \rightarrow \mathcal{B}$ is for $\text{iname} \in \mathcal{L}(\text{<inst name>})$, $\text{type} \in \mathcal{L}(\text{<type>})$ and $\text{kn} \in \mathcal{L}(\text{<kind name>})$ defined by

\[
\begin{align*}
\text{typeok}(\text{iname};) & = \text{true} \\
\text{typeok}(\text{iname decomposed};) & = \text{true} \\
\text{typeok}(\text{iname kn};) & = \text{true} \\
\text{typeok}(\text{iname kn decomposed};) & = \text{true} \\
\text{typeok}(\text{iname type kn};) & = (\text{type} = \text{process}) \\
\text{typeok}(\text{iname type kn decomposed};) & = (\text{type} = \text{process}) \\
\text{typeok}(\text{iname kn};) & = \text{true} \\
\text{typeok}(\text{iname type kn};) & = (\text{type} = \text{process}) \\
\text{typeok}(\text{iname type kn decomposed};) & = (\text{type} = \text{process})
\end{align*}
\]

This predicate is used in formalizing the syntax requirement concerning the type of created and creating instances.

Definition 47 The predicate $\text{TypelnstsOK} : \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{B}$ is for all $\text{ch} \in \mathcal{L}(\text{<msc>})$ defined by

\[
\text{TypelnstsOK}(\text{ch}) = \bigwedge_{i \in \text{Alllnsts}(\text{ch})} (\text{CreateEvents}(i) \neq \square \Rightarrow \text{typeok}(\text{InstHead}(i)))
\]

where $\text{IsCreated}(\text{ch},i)$ is an abbreviation for $\bigvee_{\text{cr} \in \text{CreateEvents}(\text{ch})} \text{CreateName}(\text{cr}) = \text{InstName}(i)$.

The first conjunct in the above predicate expresses that an instance which has a create event in its body must be checked for correct typing and the second conjunct expresses the type checking for created instances. Instances which are not created and which do not create an instance do not have to be type checked.

### 4.7 Rule for Process Termination

As was the case for process creation, also for process termination there is a requirement on the type of instances which may be terminated. Only instances of type process may be terminated. Recommendation Z.120 states for the typing of terminating instances the following requirement.

*The stop at the end of an instance definition is allowed only for instances of type process [IT93, section 4.8, page 18].*

A predicate $\text{Process}$ is defined which determines whether the type of an instance head is process.
Definition 48 The predicate Process: \( \mathcal{L}(\text{<inst head>}) \rightarrow \mathcal{B} \) is for iname \( \in \mathcal{L}(\text{<inst name>}) \), type \( \in \mathcal{L}(\text{<type>}) \) and kn \( \in \mathcal{L}(\text{<kind name>}) \) defined by

\[
\begin{align*}
\text{Process(iname,)} &= \text{false} \\
\text{Process(iname decomposed,)} &= \text{false} \\
\text{Process(iname kn,)} &= \text{false} \\
\text{Process(iname kn decomposed,)} &= \text{false} \\
\text{Process(iname type kn,)} &= (\text{type = process}) \\
\text{Process(iname type kn decomposed,)} &= (\text{type = process}) \\
\text{Process(iname:kn,)} &= \text{false} \\
\text{Process(iname:kn decomposed,)} &= \text{false} \\
\text{Process(iname:type kn,)} &= (\text{type = process}) \\
\text{Process(iname:type kn decomposed,)} &= (\text{type = process})
\end{align*}
\]

Next, a predicate Stopped is defined which determines whether a stop event is specified within an instance definition.

Definition 49 The predicate Stopped: \( \mathcal{L}(\text{<inst body>}) \rightarrow \mathcal{B} \) is for all \( e \in \mathcal{L}(\text{<event>}) \) and \( ib \in \mathcal{L}(\text{<inst body>}) \) defined by

\[
\begin{align*}
\text{Stopped(<>)} &= \text{false} \\
\text{Stopped(stop)} &= \text{true} \\
\text{Stopped(e ib)} &= \text{Stopped(ib)}
\end{align*}
\]

and the predicate Stopped: \( \mathcal{L}(\text{<inst def>}) \rightarrow \mathcal{B} \) is for all \( i \in \mathcal{L}(\text{<inst def>}) \) defined by

\[
\text{Stopped(i)} = \text{Stopped(InstBody(i))}
\]

Clearly, all instances with a stop event have to be checked for their type.

Definition 50 The predicate TypeInstsOK': \( \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{B} \) is for all \( ch \in \mathcal{L}(\text{<msc>}) \) defined by

\[
\text{TypeInstsOK'}(ch) = \bigwedge_{i \in \text{AllInsts}(ch), \text{Stopped(i)}} \text{Process}(i)
\]

4.8 Causal Order Consistency

A chart specifies a partial ordering on the set of events being contained. This partial ordering restricted to communication events and process creation events is described in a minimal form by the causal dependency graph. The causal dependency graph is an extension of the connectivity graph [IT93]. It is extended with nodes representing process creation events and with arrows from
a create node to the first node stemming from the created instance. The nodes of the causal dependency graph represent the message output, message input and process creation events. If a node represents a message output event it is labelled with an exclamation mark (!). If a node represents a message input event it is labeled with a question mark (?). Besides these labels a node is also labelled by the triple that identifies the abstract message that the communication event references. A process creation event is represented by a pair of instance names: the creating instance and the created instance. The arrows between these nodes represent the partial ordering of the events as specified by the chart.

In this section the rules on the causal ordering specified by the chart are discussed. First, those requirements will be discussed and explained informally. The reason for postponing the formal treatment of the requirements is that all of those are defined in terms of the relation induced by the causal dependency graph.

It is required that a message is sent before it is consumed. Also with Message Sequence Charts this convention is followed. This means that it is not allowed that the partial ordering of the events specified by the chart states that a message input occurs (in time) before its corresponding message output. Consider for example the charts from Figure 4.4.

It is clear that chart ex1 specifies that the input of message $m$ is executed before the output of message $m$. For chart ex2, the observation that the output of message $n$ is preceded by the input of the same message is somewhat more difficult. The syntax requirement is formulated as follows.

It is not allowed that a message output is causally depending on its corresponding message input, directly or via other messages [IT93, section 4.3, page 12].

In Figure 4.5 the causal dependency graphs of the example charts are given. In both cases it is clear that it contains a loop.
The requirement as stated in [IT93] is not completely satisfactory. It states the dependency of message inputs on message outputs only in terms of messages. But it can also be the case that a message output is causally depending on its corresponding message input via the creation of an instance. Consider for example the chart given in Figure 4.6. In this chart instance $i$ receives a message $m$ from instance $j$ and then it creates instance $j$. However, instance $j$ must be created first before it can send message $m$ to instance $i$. As a result, instance $i$ cannot receive message $m$ from instance $j$ since instance $j$ has not been created yet and instance $j$ cannot be created by instance $i$ since it must first receive message $m$.

Therefore, the requirement is adapted by stating that a message input may not causally depend on its corresponding message output via zero or more events. The only events which can cause such a causal dependency are communication events and create events.

*It is not allowed that a message output is causally depending on its corresponding message input, directly or via other events (no reference).*

42
The above requirement is not strong enough since the creation of an instance is considered a special type of communication. In this context, the instance on which the create event is specified is considered the sender of the message. The instance specified by the create event is, implicitly, considered the receiver of the message. Moreover, the reception of the create message is the first action executed by the created instance. Therefore, it must not be the case that the chart specifies that another event of the created instance precedes the execution of the process creation event.

Consider the chart given in Figure 4.6. After the reception of a message \( m \) from instance \( j \), instance \( i \) creates instance \( j \). Instance \( j \) sends a message \( m \) to instance \( i \) after it has been created by instance \( i \). The instances are waiting for each other. For the example chart the causal dependency graph is given in Figure 4.7. To avoid this kind of deadlock specification, the following syntax requirement is added.

*The creation of an instance must not depend causally on any event from that instance (no reference).*

Next, the requirements are formalized. For the labels of the nodes of the causal dependency graph the following types are introduced.

**Definition 51** The following types are defined for the various labels of the nodes of the dependency graph.

\[
\begin{align*}
\text{MsgLabel} &= \{!, ?\} \times \text{Msg} \\
\text{CreateLabel} &= \mathcal{L}(<\text{inst name}> \rangle) \times \mathcal{L}(<\text{inst name}> \rangle) \\
\text{NodeLabel} &= \text{MsgLabel} \cup \text{CreateLabel}
\end{align*}
\]

Next, a function \( \text{MsgEvent} \) is defined which, given an instance definition, associates to a communication event a message label. This function will be used in determining the label of the node which represents a communication event. Given an instance definition, the function \( \text{CrEvent} \) associates to a create event the label of the node representing it in the causal dependency graph.
Definition 52 Let $i \in \mathcal{L}(\text{<inst def>})$. The function $\text{MsgEvent} : \mathcal{L}(\text{<out> | <in>}) \to \mathcal{M}sg\text{Label}$ is for all $\text{out} \in \mathcal{L}(\text{<out>})$ and $\text{in} \in \mathcal{L}(\text{<in>})$ defined by

$$\text{MsgEvent}(i)(\text{out}) = (!, \text{Message}(i)(\text{out}))$$
$$\text{MsgEvent}(i)(\text{in}) = (? , \text{Message}(i)(\text{in}))$$

Let $i \in \mathcal{L}(\text{<inst def>})$. The function $\text{CrEvent}(i) : \mathcal{L}(\text{<create>}) \to \text{CreateLabel}$ is for all $\text{cr} \in \mathcal{L}(\text{<create>})$ defined by

$$\text{CrEvent}(i)(\text{cr}) = (\text{InstName}(i), \text{CreateName}(\text{cr}))$$

Example 53 The functions $\text{MsgEvent}$ and $\text{CrEvent}$ are illustrated for the chart example from Figure 4.6. Denote the textual description of the instance definitions with names $i$ and $j$ respectively. Then the label of the node associated to the output of message $m$ by instance $j$ is given by $\text{MsgEvent}(i_2)(\text{out m to } i_1) = (!, (j, i, m))$. The label of the node representing the corresponding input of message $m$ by instance $i$ is given by $\text{MsgEvent}(i_1)(\text{in m from } j) = (? , (j, i, m))$. The label of the node representing the creation of instance $j$ by $i$ is given by $\text{CrEvent}(i_1)(\text{create } j) = (i, j)$.

First, the ordering on events specified by an instance is computed. This is done by scanning the instance body and relating those events which are specified immediately adjoining. A function $\text{Nodes}$ is defined which, given an instance definition, associates to an event the label of the node representing this event in the causal dependency graph, and which associates to a coregion the labels of the nodes that represent the communication events specified within the coregion.

Definition 54 Let $i \in \mathcal{L}(\text{<inst def>})$. The function $\text{Nodes}(i) : \mathcal{L}(\text{<out> | <in> | <coregion> | <create>}) \to \mathcal{I}P(\text{NodeLabel})$ is for all $\text{out} \in \mathcal{L}(\text{<out>})$, $\text{in} \in \mathcal{L}(\text{<in>})$, $\text{co} \in \mathcal{L}(\text{<coregion>})$ and $\text{cr} \in \mathcal{L}(\text{<create>})$ defined by

$$\text{Nodes}(i)(\text{out}) = \{\text{MsgEvent}(i)(\text{out})\}$$
$$\text{Nodes}(i)(\text{in}) = \{\text{MsgEvent}(i)(\text{in})\}$$
$$\text{Nodes}(i)(\text{co}) = \{\text{MsgEvent}(i)(\text{com}) \mid \text{com} \in \text{Outputs}(\text{co}) \cup \text{Inputs}(\text{co})\}$$
$$\text{Nodes}(i)(\text{cr}) = \{\text{CrEvent}(i)(\text{cr})\}$$

The function $\text{ICDG}$, which stands for instance causal dependency graph, associates to an instance definition a set of pairs of node labels. Such a pair represents an arrow in the instance dependency graph. Note that this does not give the complete causal dependency graph; it only gives the partial ordering specified by an instance in isolation. The first argument of the function $\text{ICDG}(i)$ acts as a "memory". It records the events immediately preceding the event under consideration. This is needed to be able to compute the arrows of the instance dependency graph. In the case that the immediately preceding event is a coregion, all communication events of the coregion are memorized.

Definition 55 Let $i \in \mathcal{L}(\text{<inst def>})$ be given. The function $\text{ICDG}(i) : \mathcal{I}P(\text{NodeLabel}) \times \mathcal{L}(\text{<inst body>}) \to \mathcal{I}P(\text{NodeLabel} \times \text{NodeLabel})$ is for all $\mathcal{L} \subseteq \text{NodeLabel}$, $e \in \mathcal{L}(\text{<event>})$
and \( ib \in \mathcal{L}(\text{<inst body>}) \) defined by 

\[
\begin{align*}
ICDG(i)(L, \langle \rangle) & = \emptyset \\
ICDG(i)(L, \text{stop};) & = \emptyset \\
ICDG(i)(L, e \ ib) & =
\begin{cases}
L \times \text{Nodes}(i)(e) \cup ICDG(i)(\text{Nodes}(i)(e), \ib) & \text{if } e \in \mathcal{L}(\langle \text{out} \rangle | \langle \text{in} \rangle) \\
ICDG(i)(L, \ib) & \text{otherwise}
\end{cases}
\end{align*}
\]

The function \( ICDG : \mathcal{L}(\text{<inst def>}) \rightarrow IP(\text{NodeLabel} \times \text{NodeLabel}) \) is for \( i \in \mathcal{L}(\text{<inst def>}) \) defined by 

\[
ICDG(i) = ICDG(i)(\emptyset, \text{InstBody}(i))
\]

**Example 56** The function \( ICDG \) is illustrated also for the chart from Figure 4.6 of the previous example. The causal dependency graphs for the instance definitions in isolation are computed:

\[
ICDG(i_1) = ICDG(i_1)(\emptyset, \text{in} m \text{ from } j; \ 	ext{create } j)
\]

\[
= (\emptyset \times \text{Nodes}(i_1)(\text{in} m \text{ from } j)) \\
\cup ICDG(i_1)(\text{Nodes}(i_1)(\text{in} m \text{ from } j), \text{create } j)
\]

\[
= \emptyset \cup ICDG(i_1)(\text{Nodes}(i_1)(\text{in} m \text{ from } j), \text{create } j)
\]

\[
= ICDG(i_1)(\text{Nodes}(i_1)(\text{in} m \text{ from } j), \text{create } j)
\]

\[
= ICDG(i_1)(\{(?, (j, i, m))\}, \text{create } j)
\]

\[
= \{(?, (j, i, m)) \times \text{Nodes}(i_1)(\text{create } j)\}
\]

\[
\cup ICDG(i_1)(\text{Nodes}(i_1)(\text{create } j), \langle \rangle)
\]

\[
= \{(?, (j, i, m)) \times \{(i,j)\}\} \cup \emptyset
\]

\[
= \{(?, (j, i, m)), (i,j)\}
\]

A similar computation for instance definition \( i_2 \) results in \( ICDG(i_2) = \emptyset \).

In words the above computations are interpreted as follows. Instance \( i \) specifies that the input of message \( m \) must be executed before the creation of instance \( j \). Instance \( j \) does not give any causal dependencies between events specified on that instance since there is only one event.

Next, the set of pairs of labels is interpreted as a relation on labels. Besides the ordering on events specified explicitly by the instances, there is also the ordering between corresponding message outputs and message inputs and the ordering between a create event and the first events from the created instance. The relation \( \longrightarrow \) which will be defined shortly, specifies both the ordering specified by the instances of the chart (as expressed by \( ICDG \)) and the implicit orderings via communication and process creation.

To determine the nodes which represent the first events of a created instance the function \( \text{FirstEvents} \) is used.
Definition 57 Let \( i \in \mathcal{L}(\texttt{inst def}). \) The function \( \text{FirstEvents}(i) : \mathcal{L}(\texttt{inst body}) \rightarrow \mathcal{IP}(\texttt{NodeLabel}) \) is for all \( e \in \mathcal{L}(\texttt{event}) \) and \( ib \in \mathcal{L}(\texttt{inst body}) \) defined as follows.

\[
\text{FirstEvents}(i)(<>) = \emptyset \\
\text{FirstEvents}(i)(\text{stop}_i) = \emptyset \\
\text{FirstEvents}(i)(e \text{ ib}) = \begin{cases} 
\text{Nodes}(i)(e) & \text{if } e \in \mathcal{L}(<\text{out}> | <\text{in}>), \\
\forall e \in \mathcal{L}(<\text{coregion}> | <\text{create}>), \\
\text{FirstEvents}(i)(ib) & \text{otherwise}
\end{cases}
\]

The function \( \text{FirstEvents} : \mathcal{L}(\texttt{inst def}) \rightarrow \mathcal{IP}(\texttt{NodeLabel}) \) is for all \( i \in \mathcal{L}(\texttt{inst def}) \) defined as follows.

\[
\text{FirstEvents}(i) = \text{FirstEvents}(i)(\text{InstBody}(i))
\]

Definition 58 Let \( ch \in \mathcal{L}(\texttt{msc}). \) The relation \( \xrightarrow{\text{ch}} \subseteq \texttt{NodeLabel} \times \texttt{NodeLabel} \) is for all \( t_1, t_2 \in \texttt{NodeLabel} \) defined as follows: \( t_1 \xrightarrow{\text{ch}} t_2 \) if and only if either one of the following conditions holds

1) \( (t_1, t_2) \in \text{ICDG}(i) \) for some \( i \in \text{AllInsts}(ch); \)

2) \( t_1 \equiv \text{MsgEvent}(i)(\text{out}) \) and \( t_2 \equiv \text{MsgEvent}(j)(\text{in}) \) for some \( i \in \text{AllInsts}(ch), \text{out} \in \text{Outputs}(i), \) and, for some \( j \in \text{AllInsts}(ch), \text{in} \in \text{Inputs}(j) \) such that \( \text{Message}(i)(\text{out}) = \text{Message}(j)(\text{in}); \)

3) \( t_1 \equiv \text{CrEvent}(i)(cr) \) for some \( i \in \text{AllInsts}(ch) \) and \( cr \in \text{CreateEvents}(i) \) such that, for some \( j \in \text{AllInsts}(ch), \text{CreateName}(cr) = \text{InstName}(j) \) and \( t_2 \in \text{FirstEvents}(j). \)

The first clause in the above definition describes the ordering of events specified explicitly by the instances, the second clause relates corresponding message output and message input events and the third clause relates a create event to the first events of the created instance.

Example 59 In the previous example the instance dependency graphs for the instance definitions of the chart shown in Figure 4.6 were computed:

\[
\text{ICDG}(i_1) = \{((?,(j,i,m)),(i,j))\} \\
\text{ICDG}(i_2) = \emptyset
\]

Denote the complete chart of Figure 4.6 by \( ch. \) In this example these are combined into a causal dependency graph for the complete chart as follows. Denote the complete chart by \( ch. \)

1) From the first clause edges are obtained which represent the causal ordering specified by the instances in isolation: \((?,(j,i,m)) \xrightarrow{\text{ch}} (i,j); \)

2) From the second clause edges are obtained which relate a node representing a message output event to a node representing the corresponding message input event: \((!, (j,i,m)) \xrightarrow{\text{ch}} (?, (j,i,m)); \)
3) From the third clause edges are obtained which relate a node representing a create event to
the nodes representing the events of the created instance: \((i,j) \xrightarrow{ch} (!, (j,i,m))\).

The graphical representation of this causal dependency graph is already given in Figure 4.7.

In terms of the causal dependency graph the syntax requirement is formulated as: the causal de­
pendency graph does not contain loops or, alternatively, there must not be a path from a node to
itself. Next, this is formulated in terms of the relation \(\xrightarrow{ch}\), using the notions of transitive closure
and strictness of relations (see Chapter 3).

**Definition 60** The predicate \(\text{Acyclic} : \mathcal{L}(<\text{msc}>) \rightarrow \mathbb{B}\) is for all \(ch \in \mathcal{L}(<\text{msc}>)\) defined by

\[
\text{Acyclic}(ch) = \xrightarrow{ch}^+ \text{ is strict}
\]

### 4.9 Consistent Condition Order

The drawing rules of Recommendation Z.120 ([IT93, Drawing Rules: section 2.5, page 5]) do not
allow for two conditions to overlap or to cross. In this section this drawing rule for the graphical
representation of charts is considered. This requirement is formulated for the textual representa­
tion of charts as follows.

*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* [IT94, Extensions: section 5, page 8].

From the textual representation of a chart, a *condition graph* will be constructed. The nodes of
the condition graph are the abstract conditions from Section 4.5 extended with the occurrence
number. The relation between these nodes expresses the orderings as they are enforced by the
instance definitions. The requirement above then amounts to the following: the condition graph
of a chart may not contain loops.

First, for every instance in isolation the condition graph will be constructed. The nodes of this
graph are labelled by a tuple that consists of the abstract condition corresponding with the con­
dition event and the occurrence number of the reference to the abstract condition on the instance
under consideration. The arrows of the condition graph represent the relation between the con­
ditions as specified by the instance. Thereafter, the condition graphs of the instances of the chart
are merged into one condition graph by identifying nodes with the same label.

Consider the following chart.

```plaintext
msc example;
instance i;
```
condition Cl shared k;
condition C2 shared j;
endinstance;
instance j;
    condition C2 shared i;
    condition C3 shared k;
endinstance;
instance k;
    condition C3 shared j;
    condition C1 shared i;
endinstance;
endmsc;

The graphical representation and the condition graph of this chart are given in Figure 4.8.

Figure 4.8: Chart example and its condition graph

The condition graph is obtained in essentially the same way as the causal dependency graph was obtained in the previous section. First a type \textit{CondLabel} is defined which is used for the nodes representing conditions. Then the condition graph for a single instance and for a complete chart are defined.

\textbf{Definition 61} The type \textit{CondLabel} is defined by

\[ \text{CondLabel} = \text{Condition} \times \text{IN} \]
In order to define the condition graph of one instance the function \(\text{Nodes}(ch,i,ib)\) is defined which associates to a condition event \(c\) a tuple consisting of its abstract condition \((\text{Cond}(ch,i)(c))\) and its occurrence number. This number is needed to distinguish the different occurrences of the same abstract condition. The occurrence number of a condition event is determined from the number of occurrences of that abstract condition after the condition event under consideration. Thereto, the remaining part of the instance body after the occurrence of condition event \(c\) is needed. The choice for associating a set of such nodes to a single condition is purely technical. It prevents the definition of an auxiliary function which acts as an initialization case.

**Definition 62** Let \(ch \in \mathcal{L}(<\text{msc}>)\), \(i \in \mathcal{L}(<\text{inst def}>)\) and \(ib \in \mathcal{L}(<\text{inst body}>)\). The function \(\text{Node}(ch,i,ib) : \mathcal{L}(<\text{condition}>) \rightarrow \mathcal{P}(<\text{CondLabel}>)\) is for all \(c \in \mathcal{L}(<\text{condition}>)\) defined by

\[
\text{Nodes}(ch,i,ib)(c) = \{(\text{Cond}(ch,i)(c),
\begin{cases}
\text{Cond}(ch,i)(c') & \text{if } c' \in \text{CondEvents}(i) - \text{CondEvents}(ib) \\
& \text{otherwise}
\end{cases}
\}
\]

Using the function \(\text{Nodes}\), the condition graph for an instance is defined straightforwardly as follows.

**Definition 63** Let \(ch \in \mathcal{L}(<\text{msc}>)\) and \(i \in \mathcal{L}(<\text{inst def}>)\) be given. The function \(\text{CG}(ch,i) : \mathcal{P}(<\text{CondLabel}>) \times \mathcal{L}(<\text{inst body}>) \rightarrow \mathcal{P}(<\text{CondLabel} \times \text{CondLabel}>)\) is for all \(L \subseteq \text{CondLabel}\), \(e \in \mathcal{L}(<\text{event}>)\) and \(ib \in \mathcal{L}(<\text{inst body}>)\) defined by

\[
\text{CG}(ch,i)(L,<> ) = \emptyset \\
\text{CG}(ch,i)(L,\text{stop} ;) = \emptyset \\
\text{CG}(ch,i)(L,e \; ib) = \begin{cases}
L \times \text{Nodes}(ch,i,ib)(e) & \text{if } e \in \mathcal{L}(<\text{condition}>) \\
\text{CG}(ch,i)(\text{Nodes}(ch,i,ib)(e),ib) & \text{if } e \notin \mathcal{L}(<\text{condition}>)
\end{cases}
\]

The function \(\text{CG} : \mathcal{L}(<\text{msc}>) \times \mathcal{L}(<\text{inst def}>) \rightarrow \mathcal{P}(<\text{CondLabel} \times \text{CondLabel}>)\) is for all \(ch \in \mathcal{L}(<\text{msc}>)\) and \(i \in \mathcal{L}(<\text{inst def}>)\) defined by

\[
\text{CG}(ch,i) = \text{CG}(ch,i)(\emptyset,\text{InstBody}(i))
\]

Then, for a chart \(ch\), the condition graph is obtained by merging the individual instance condition graphs by identifying nodes with the same name. This condition graph is expressed as a relation.
Definition 64 Let $ch \in \mathcal{L}(<\text{msc}>)$). The relation $\frac{ch}{\Rightarrow} \subseteq \text{CondLabel} \times \text{CondLabel}$ is for all $t_1, t_2 \in \text{CondLabel}$ defined by

$$t_1 \xrightarrow{ch} t_2 \text{ if and only if } \bigvee_{i \in \text{AllInsts}(ch)} (t_1, t_2) \in CG(ch, i)$$

As is the previous section, the requirement can be stated as follows: the relation which represents the condition graph is strict.

Definition 65 The predicate $\text{NoOverlap} : \mathcal{L}(<\text{msc}>) \rightarrow \mathcal{B}$ is for all $ch \in \mathcal{L}(<\text{msc}>)$ defined by

$$\text{NoOverlap}(ch) = \xrightarrow{ch}^+ \text{ is strict}$$

4.10 The relation between conditions, messages and process creates

Conditions define possible compositions and decompositions of charts. The syntax requirement discussed and formalized in this section is closely related to the composition and decomposition of charts. Consider the chart shown in Figure 4.9.

![Figure 4.9: Chart example cannot be decomposed using condition C](image)

Although a condition is attached to all instances of the chart, this chart cannot be decomposed. The reason for this is that in that case the communication of message $m$ from instance $i$ to instance $j$ must be split into the output of message $m$ in one chart and the input of message $m$ in the other chart. These "charts" are shown in Figure 4.10.
Obviously these charts violate the requirement relating to the completeness of messages from Section 4.4.4. As a matter of fact this chart can never be decomposed along condition \( C \). Also this chart can not have resulted from composing two charts using condition \( C \). The conclusion should therefore be that this condition does not have any function at all. Therefore, the following requirement is stated in Recommendation Z.120.

*If instance a and instance b share the same condition then for each message exchanged between these instances, the message output and message input must be placed both before or both after the condition [IT93, section 4.4, page 13].*

Since create events are considered special communication events this can be repeated for the crossing of a condition by a create arrow. This results in the following additional requirement.

*If instance a and instance b share the same condition then instance a may not create instance b after the shared condition (no reference).*

Note that these requirements are not sufficient for ensuring that any chart can be decomposed. To formalize this syntax requirement, the causal dependency graph for the instance definitions from Section 4.8 is extended with nodes representing the abstract conditions the condition events refer to. The syntax requirement can then be phrased as follows. Both a message output event and its corresponding message input event must be specified either before or after the condition which refers both to the sender and receiver instance of the communication. Formulated alternatively, it may not be the case that on the one hand the message output event is specified before the condition and on the other hand the message input event is specified after the condition, and vice versa. For create events the requirement can be formulated as follows: it may not be the case that a create event is specified after a condition that references the created instance.
Example 66 On instance $i$ in Figure 4.9 an output of the abstract message $(i,j,m)$ is specified before the only reference to the abstract condition $(C, \{i,j\})$. On instance $j$ an input of the abstract message $(i,j,m)$ is specified after the only reference to the abstract condition $(C, \{i,j\})$. So, the syntax requirement formulated above is violated.

From the textual representation of a chart a *condition/message dependency graph* will be constructed for every instance in isolation. The nodes of the condition/message dependency graph represent the message output and message input events, the create events and the condition events of the instance. The relation between these nodes represents the partial ordering as it is enforced by the instance definition.

The formalization of the condition/message dependency graph follows the same lines as the formalization of the causal dependency graph in Section 4.8 and the formalization of the condition graph in Section 4.9.

For the labels of the nodes that result from a communication event the type $MsgLabel$ from Section 4.8 is used. For the labels of the nodes that result from a condition event the type $CondLabel$ from Section 4.9 is used. In Section 4.5.3 it is already explained that the occurrence number of a condition is needed.

Definition 67 The type $NodeLabel$ is defined by

$$\text{NodeLabel} = \text{MsgLabel} \cup \text{CreateLabel} \cup \text{CondLabel}$$

In Section 4.8 a function $Nodes$ was used to determine the labels of the nodes representing communication and create events. In Section 4.9 a function $Nodes$ was used to determine the labels of the nodes representing conditions. These functions will be combined in order to determine the labels of the nodes of the graph.

Definition 68 Let $ch \in \mathcal{L}(<msc>)$, $i \in \mathcal{L}(<inst\ def>)$ and $ib \in \mathcal{L}(<inst\ body>)$. The function $Nodes(ch,i,ib) : \mathcal{L}(<condition> | <out> | <in> | <coregion> | <create>) \rightarrow IP(NodeLabel)$ is for all $c \in \mathcal{L}(<condition>)$ and $e \in \mathcal{L}(<out> | <in> | <coregion> | <create>)$ defined by

$$Nodes(ch,i,ib)(c) = Nodes'(ch,i,ib)(c)$$
$$Nodes(ch,i,ib)(e) = Nodes(i)(e)$$

where the function $Nodes'(ch,i,ib)$ is identical to the function $Nodes(ch,i,ib) : \mathcal{L}(<condition>) \rightarrow IP(CondLabel)$ as defined in Section 4.9.

Definition 69 Let $ch \in \mathcal{L}(<msc>)$ and $i \in \mathcal{L}(<inst\ def>)$. The function $CMDG(ch,i) : IP(NodeLabel) \times \mathcal{L}(<inst\ body>) \rightarrow IP(NodeLabel \times NodeLabel)$ is for all $S \subseteq NodeLabel$. 52
\(e \in \mathcal{L}(\text{<event>})\) and \(ib \in \mathcal{L}(\text{<inst body>})\) defined by

\[
C/MDG(ch,i)(S,<>) = \emptyset
\]
\[
C/MDG(ch,i)(S,\text{stop}) = \emptyset
\]
\[
C/MDG(ch,i)(S,ib) = \begin{cases}
S \times \text{Nodes}(ch,ib)(e) & \text{if } e \in \mathcal{L}(\text{<condition>}) \\
\cup & \forall e \in \mathcal{L}(\text{<out> | <in>}) \\
C/MDG(ch,i)(\text{Nodes}(ch,ib)(e),ib) & \forall e \in \mathcal{L}(\text{<coregion>}) \\
C/MDG(ch,i)(S,ib) & \text{otherwise}
\end{cases}
\]

The function \(C/MDG : \mathcal{L}(\text{<msg>}) \times \mathcal{L}(\text{<inst def>}) \rightarrow \mathcal{P}(\text{NodeLabel} \times \text{NodeLabel})\) is for \(ch \in \mathcal{L}(\text{<msg>})\) and \(i \in \mathcal{L}(\text{<inst def>})\) defined by

\[
C/MDG(ch,i) = C/MDG(ch,i)(\emptyset,\text{InstBody}(i))
\]

**Definition 70** Let \(i \in \mathcal{L}(\text{<inst def>}).\) The relation \(\Rightarrow \subseteq \text{NodeLabel} \times \text{NodeLabel}\) is for all \(l_1, l_2 \in \text{NodeLabel}\) defined by

\[
l_1 \Rightarrow l_2 \quad \text{if and only if} \quad (l_1, l_2) \in C/MDG(ch,i)
\]

Next, the condition/message dependency graphs for the instances are used to formally define the predicates \(\text{NoCrossingMsg}\) and \(\text{NoCrossingCreate}.\) An abstract message \((i,j,m)\) is crossing the \(n\)-th occurrence of the abstract condition \((cn,l)\) if the node representing the message output event \(!, (i,j,m))\) precedes the node representing the \(n\)-th occurrence of the abstract condition \(((cn,l),n)\) in the condition/message dependency graph for instance \(i\), while the node representing the message input event \(? , (i,j,m)\) follows the node \(((cn,l),n)\) in the condition/message dependency graph for instance \(j\), or vice versa. A process creation \((i,j)\) is crossing the \(n\)-th occurrence of the abstract condition \((cn,l)\) if the node representing the create event follows the node representing the \(n\)-th occurrence of the abstract condition \(((cn,l),n)\) in the condition/message dependency graph for instance \(i\).

**Definition 71** The predicate \(\text{NoCrossingMsg} : \mathcal{L}(\text{<msg>}) \rightarrow \mathcal{B}\) is for all \(ch \in \mathcal{L}(\text{<msg>})\) defined by

\[
\text{NoCrossingMsg}(ch) = \bigwedge_{i,j \in \text{AllIns}(ch), \text{Out}(i), \text{Input}(j), \text{Message}(i)(\text{out})=\text{Message}(j)(\text{in}), c \in \text{CondEvents}(i), \text{InstName}(j) \in \text{CondReflIns}(ch,i)(c), n \in \mathbb{N}}
\]

\[
\neg ( \text{MsgEvent}(i)(\text{out}) \Rightarrow (\text{Cond}(ch,i)(c),n) \land \text{MsgEvent}(j)(\text{in})) \land (\text{Cond}(ch,i)(c),n) \Rightarrow \text{MsgEvent}(j)(\text{in}) \land \text{MsgEvent}(j)(\text{in}) \Rightarrow (\text{Cond}(ch,i)(c),n) \land (\text{Cond}(ch,i)(c),n) \Rightarrow \text{MsgEvent}(i)(\text{out}))
\]

53
The predicate \( \text{NoCrossingCreate} : \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{I} \) is for all \( ch \in \mathcal{L}(\text{<msc>}) \) defined by

\[
\text{NoCrossingCreate}(ch) = \bigwedge_{i,j \in \text{AllInsts}(ch)} \neg((\text{Cond}(ch,i)(c), n) \Rightarrow \text{CrEvent}(i)(cr))
\]

The predicate \( \text{NoCrossing} : \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{I} \) is for all \( ch \in \mathcal{L}(\text{<msc>}) \) defined by

\[
\text{NoCrossing}(ch) = \text{NoCrossingMsg}(ch) \land \text{NoCrossingCreate}(ch)
\]

4.11 Rules for instance decomposition

In this section the syntax requirements for instance decomposition will be considered.

4.11.1 Uniqueness of chart names

A document consists of a finite number of charts. A chart is referenced through a chart name. Therefore, it is required that there are no two charts within a document with the same chart name.

Within a document there must not be two or more charts with the same name [IT93, section 2.2, page 3].

The formalization of this requirement follows the lines of the formalization of the previously discussed uniqueness rules. First an equivalence \( \equiv_{\text{chart}} \) on charts is defined which identifies charts with the same name.

\textbf{Definition 72} The relation \( \equiv_{\text{chart}} \subseteq \mathcal{L}(\text{<msc>}) \times \mathcal{L}(\text{<msc>}) \) is for all \( ch_1, ch_2 \in \mathcal{L}(\text{<msc>}) \) defined by

\[
ch_1 \equiv_{\text{chart}} ch_2 \text{ if and only if } \text{ChName}(ch_1) = \text{ChName}(ch_2)
\]

The function \( \text{ChName} \) (see Appendix B) associates to a chart its name.

Using the function \( \text{Charts} \) (see Appendix B), which collects all charts of a document in a multiset, the formalization of the syntax requirement is as follows.

\textbf{Definition 73} The predicate \( \text{UCN} : \mathcal{L}(\text{<msc doc>}) \rightarrow \mathcal{I} \) is for all \( doc \in \mathcal{L}(\text{<msc doc>}) \) defined by

\[
\text{UCN}(doc) = \bigwedge_{ch \in \text{Charts}(doc)} \#_{\text{chart}}^ch(\text{Charts}(doc)) \leq 1
\]
4.11.2 Reference rule for charts

Charts are only referenced in the instance head of a decomposed instance. If there is a decomposed instance with name $d$ within some chart, there also has to be a chart with chart name $d$ within the same document. The formulation of the syntax requirement for references to declared charts is formulated as follows.

Within the charts of a document only references to charts specified within that document may be specified [IT93, section 2.2, page 3].

As a first step towards the formalization of this requirement a function $\text{DeclaredChartNames}$ is defined which associates to a document the names of the charts defined therein.

**Definition 74** The function $\text{DeclaredChartNames}: \mathcal{L}(\langle \text{msc doc} \rangle) \to \mathcal{P}(\mathcal{L}(\langle \text{msc name} \rangle))$ is for all $doc \in \mathcal{L}(\langle \text{msc doc} \rangle)$ defined by

\[
\text{DeclaredChartNames}(doc) = \{\text{ChName}(ch) \mid ch \in \text{Charts}(doc)\}
\]

Next, a function $\text{DecInstNames}$ is defined which associates to a document the names of all instances that are decomposed. The predicate $\text{IsDecomposed}$ determines whether an instance is decomposed. The function $\text{DecInsts}$ associates to a chart the definitions of all its decomposed instances.

**Definition 75** The predicate $\text{IsDecomposed}: \mathcal{L}(\langle \text{inst def} \rangle) \to \mathcal{B}$ is for all $\text{instname} \in \mathcal{L}(\langle \text{inst name} \rangle), \text{kind} \in \mathcal{L}(\langle \text{inst kind} \rangle)$ and $ib \in \mathcal{L}(\langle \text{inst body} \rangle)$ defined by

\[
\text{IsDecomposed}(\text{instance instname; ib endinstance;}) = \text{false}
\]
\[
\text{IsDecomposed}(\text{instance instname kind; ib endinstance;}) = \text{false}
\]
\[
\text{IsDecomposed}(\text{instance instname:kind; ib endinstance;}) = \text{false}
\]
\[
\text{IsDecomposed}(\text{instance instname decomposed; ib endinstance;}) = \text{true}
\]
\[
\text{IsDecomposed}(\text{instance instname kind decomposed; ib endinstance;}) = \text{true}
\]
\[
\text{IsDecomposed}(\text{instance instname:kind decomposed; ib endinstance;}) = \text{true}
\]

The function $\text{DecInsts}: \mathcal{L}(\langle \text{msc} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{inst def} \rangle))$ is for all $ch \in \mathcal{L}(\langle \text{msc} \rangle)$ defined by

\[
\text{DecInsts}(ch) = \{i \in \text{AllInsts}(ch) \mid \text{IsDecomposed}(i)\}
\]

The function $\text{DecInstNames}: \mathcal{L}(\langle \text{msc doc} \rangle) \to \mathcal{P}(\mathcal{L}(\langle \text{msc name} \rangle))$ is for $doc \in \mathcal{L}(\langle \text{msc doc} \rangle)$ defined by

\[
\text{DecInstNames}(doc) = \bigcup \{\text{InstName}(i) \mid i \in \text{DecInsts}(ch)\}
\]

The formalization of the syntax requirement is as follows.

**Definition 76** The predicate $\text{ReferencedCharts}: \mathcal{L}(\langle \text{msc doc} \rangle) \to \mathcal{B}$ is for $doc \in \mathcal{L}(\langle \text{msc doc} \rangle)$ defined by

\[
\text{ReferencedCharts}(doc) = (\text{DecInstNames}(doc) \subseteq \text{DeclaredChartNames}(doc))
\]
4.11.3 Restriction on Recursive Refinement

By introducing the possibility to decompose an instance by a chart, the charts specified in a document are no longer independent of each other. This dependency is illustrated by the charts of the document shown in Figure 4.11.

Consider the charts example and d. Chart example specifies a non-decomposed instance i and a decomposed instance d. This decomposed instance d refers to chart d. As a result chart example depends on chart d.

A chart may not be depending on itself, directly or through a number of decompositions [IT94, Extensions: section 6, page 8].

A chart dependency graph is constructed from the textual representation of a document. The nodes of the chart dependency graph are labelled by the charts from the document and the arrows between the nodes of the graph represent the dependency relation induced by decomposition. First, a relation $\sim_{\text{doc}}$ is defined which relates a chart $ch_1$ and a chart $ch_2$ if chart $ch_1$ has a decomposed instance with the name of chart $ch_2$. Note that it would be sufficient to label the nodes of the chart dependency graph by the names of the charts instead of the complete definitions of the chart. However, it is easier to use the complete definitions.

**Definition 77** Let $doc \in \mathcal{L}(<\text{msc doc}>)$. The relation $\sim_{\text{doc}} \subseteq \text{Charts}(doc) \times \text{Charts}(doc)$ is for all $ch_1, ch_2 \in \text{Charts}(doc)$ defined by

$$ch_1 \sim_{\text{doc}} ch_2 \text{ if and only if } \forall i \in \text{DeclInst}(ch_1) \text{ InstName}(i) = \text{ChName}(ch_2)$$
Example 78 The chart dependency graph is illustrated for the above document. Denote this document by \( doc \) and denote the charts with names \( example \) and \( d \) by \( ch_1 \) and \( ch_2 \) respectively. Then the chart dependency graph of \( doc \) is represented by \( ch_1 \xrightarrow{doc} ch_2 \).

The requirement that a chart must not depend on itself via decomposition can now be expressed as a property of the chart dependency graph: the chart dependency graph must not contain loops. Formulated in terms of the relation underlying the chart dependency graph the formalization is as follows.

Definition 79 The predicate \( Relation: L(\langle msc \text{ doc} \rangle) \to \mathcal{B} \) is defined by

\[
Relation(doc) = \xrightarrow{doc^+} \text{ is strict}
\]

4.11.4 Additional uniqueness rules

The relation between the charts of a document imposed by the instance decompositions implies an additional uniqueness rule for the instances of related charts. Also for messages a new uniqueness rule is needed. In this section these uniqueness rules are considered. First, the additional uniqueness rule for instance names is discussed and formalized. Then, the additional uniqueness rule for messages is treated.

Consider the document shown in Figure 4.12.

![Figure 4.12: Example Message Sequence Chart document](image)

Both the charts in the document respect the uniqueness rule for instances from Section 4.2. But in this situation the charts \( ex \) and \( d \) are related via the decomposition of instance \( d \) of chart \( ex \). Thereby, the specified system consists of the nondecomposed instances of chart \( ex \), i.e. instance \( i \), and the nondecomposed instances of chart \( d \), i.e. instance \( i \). Although uniqueness of instances is respected by each chart in separation, the combination contains two instances \( i \). The following additional uniqueness rule is formulated.
It is not allowed that two charts, from the same document, which are related via decomposition have any instances with the same name (no reference).

First, a relation \( \text{doc} \) will be defined which determines whether two charts are related via decomposition. Informally, this can be expressed by stating that two charts are related via decompositions if and only if there is a chart in the document which depends on both of them. Translated to the chart dependency graph this amounts to: two charts are related via decompositions if and only if they have a common ancestor in the chart dependency graph.

**Definition 80** Let \( \text{doc} \in \mathcal{L}(<\text{mec doc}>). \) The relation \( \text{doc} \subseteq \text{Charts}(\text{doc}) \times \text{Charts}(\text{doc}) \) is for all \( \text{ch}_1, \text{ch}_2 \in \text{Charts}(\text{doc}) \) defined by

\[
\exists_{\text{doc}} \text{ ch}_1 \leftrightarrow \text{ch}_2 \text{ if and only if } \bigvee_{\text{ch} \in \text{Charts}(\text{doc})} (\text{ch} \leftrightarrow \text{ch}_1 \land \text{ch} \leftrightarrow \text{ch}_2)
\]

The symbol \( \ast \) denotes the reflexive-transitive closure of a relation (see Chapter 3).

**Example 81** Denote the document from above by \( \text{doc} \) and denote the charts with names \( \text{example} \) and \( \text{d} \) by \( \text{ch}_1 \) and \( \text{ch}_2 \) respectively. The chart dependency graph of \( \text{doc} \) is represented by \( \text{ch}_1 \leftrightarrow \text{ch}_2 \). From this chart dependency graph it is obtained that the charts \( \text{ch}_1 \) and \( \text{ch}_2 \) are related: \( \text{ch}_1 \leftrightarrow \text{ch}_2 \). Also according to the definition of \( \leftrightarrow \) the charts are related to themselves: \( \text{ch}_1 \leftrightarrow \text{ch}_1 \) and \( \text{ch}_2 \leftrightarrow \text{ch}_2 \).

Using the function \( \text{DeclInstNames} \), the above requirement is formalized as follows.

**Definition 82** The predicate \( \text{UniqueInstances} : \mathcal{L}(<\text{mec doc}>) \rightarrow \mathbb{B} \) is for \( \text{doc} \in \mathcal{L}(<\text{mec doc}>) \) defined by

\[
\text{UniqueInstances}(\text{doc}) = \bigwedge_{\text{ch}_1, \text{ch}_2 \in \text{Charts}(\text{doc}) \land \text{ch}_1 \leftrightarrow \text{ch}_2} \text{DeclInstNames}(\text{ch}_1) \cap \text{DeclInstNames}(\text{ch}_2) = \emptyset
\]

**Example 83** Verifying the uniqueness of instance names for the document from the previous example amounts to verifying whether the names of the declared instances of the charts which are related \( \text{ch}_1 \leftrightarrow \text{ch}_2 \) and which are not identical \( \text{ch}_1 \neq \text{ch}_2 \) are disjoint. The names of the instances of \( \text{ch}_1 \) and \( \text{ch}_2 \) are given by \( \text{DeclInstNames}(\text{ch}_1) = \{i,d\} \) and \( \text{DeclInstNames}(\text{ch}_2) = \{i\} \). It is clear that these sets are not disjoint. Therefore, the syntax requirement on the uniqueness of instances is violated.

In the context of decomposition there is a need for a reformulation of the uniqueness of messages. Consider, for example, the charts shown in Figure 4.13.
Figure 4.13: Example Message Sequence Chart document

Within chart example there are messages \((d,i,m)\) and \((d,j,m)\). Clearly the syntax requirement for the uniqueness of messages is not violated in this chart. In chart \(d\) there are messages \((k,env,m)\) and \((l,env,m)\). Also for this chart the syntax requirement for the uniqueness of messages is not violated. But now it is impossible to determine which message output on the decomposed instance is connected to which message output to the environment of the chart. For this reason the following syntax requirement is formulated.

*On a decomposed instance there must not be two or more message output events with the same message identifier. On a decomposed instance there must not be two or more message input events with the same message identifier (no reference).*

The formalization of this rule is almost analogous to the formalization of the syntax requirement for the uniqueness of messages in Section 4.4.3. Only the equivalence relations \(<\text{out}>\) and \(<\text{in}>\) need to be adjusted for decomposed instances. All other predicates remain almost the same.

**Definition 84** The relation \(\equiv_{\text{dec}}\subseteq L(<\text{out}>) \times L(<\text{out}>)\) is for all \(out, out' \in L(<\text{out}>)\) defined by

\[
out \equiv_{\text{dec}} out' \text{ if and only if } MsgId(out) = MsgId(out')
\]

The predicate \(UMO_{\text{dec}} : L(<\text{inst def}>) \to \mathcal{B}\) is for all \(i \in L(<\text{inst def}>)\) defined by

\[
UMO_{\text{dec}}(i) = \bigwedge_{out \subseteq Outputs(i)} \forall_{\equiv_{\text{dec}}}(Outputs(i)) \leq 1
\]

The relation \(\equiv_{\text{dec}}\subseteq L(<\text{in}>) \times L(<\text{in}>)\) is for all \(in, in' \in L(<\text{in}>)\) defined by

\[
in \equiv_{\text{dec}} in' \text{ if and only if } MsgId(in) = MsgId(in')
\]
The predicate $UM_{i\text{dec}} : \mathcal{L}(\text{<inst def>}) \rightarrow \mathcal{B}$ is for all $i \in \mathcal{L}(\text{<inst def>})$ defined by

$$UM_{i\text{dec}}(i) = \bigwedge_{in \in Inputs(i)} \frac{\text{in}}{\text{dec}}(\text{Inputs}(i)) \leq 1$$

The predicate $Unique\text{Messages}_{i\text{dec}} : \mathcal{L}(\text{<msc doc>}) \rightarrow \mathcal{B}$ is for all $doc \in \mathcal{L}(\text{<msc doc>})$ defined by

$$Unique\text{Messages}_{i\text{dec}}(doc) = \bigwedge_{ch \in \text{Charts}(doc)} \bigwedge_{i \in \text{Decins}(ch)} UM_{i\text{dec}}(i)$$

4.11.5 Interface relation rule

In an early stage of design only the specification of the interaction with the other functional blocks is given. Later in the development stage there may well be a need for a more concise description of that functional block. As a result there may be a need for a partition of the functional block into several functional blocks. Meanwhile the interaction with the original functional blocks remains the same. This yields the following syntax requirement.

There must be a unique correspondence between the external message outputs of a decomposed instance and the message outputs of the corresponding chart which are sent to the exterior. An analogous correspondence must hold for incoming messages [IT93, section 4.2, page 9, section 5.2, pages 20-21].

With respect to a given instance, an external message output of that instance is a message output which is sent by the given instance but not also received by that instance. Analogously, an environmental message input is a message input which is not sent by its receiving instance. An environmental message output is a message output that is sent to the environment. An environmental message input is a message input that is received from the environment. The functions Ext\text{Outputs} and Env\text{Outputs} determine the set of external message outputs of an instance and the set of environmental outputs of a chart respectively. The functions Ext\text{Inputs} and Env\text{Inputs} determine analogous sets for message inputs.

**Definition 85** The function $Env\text{Outputs} : \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{P}(\mathcal{L}(\text{<out>}))$ is for all $ch \in \mathcal{L}(\text{<msc>})$ defined by

$$Env\text{Outputs}(ch) = \{out | out \in \text{Outputs}(ch) \land \text{Address}(out) = \text{env}\}$$

The function $Env\text{Inputs} : \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{P}(\mathcal{L}(\text{<in>}))$ is for all $ch \in \mathcal{L}(\text{<msc>})$ defined by

$$Env\text{Inputs}(ch) = \{in | in \in \text{Inputs}(ch) \land \text{Address}(in) = \text{env}\}$$
The function $\text{ExtOutputs} : \mathcal{L}(\text{<inst def>}) \rightarrow IP(\mathcal{L}(\text{<out>}))$ is for all $i \in \mathcal{L}(\text{<inst def>})$ defined by

$$\text{ExtOutputs}(i) = \{\text{out} \mid \text{out} \in \text{Outputs}(i) \land \text{Address(out)} \neq \text{InstName}(i)\}$$

The function $\text{ExtInputs} : \mathcal{L}(\text{<inst def>}) \rightarrow IP(\mathcal{L}(\text{<in>}))$ is for all $i \in \mathcal{L}(\text{<inst def>})$ defined by

$$\text{ExtInputs}(i) = \{\text{in} \mid \text{in} \in \text{Inputs}(i) \land \text{Address(in)} \neq \text{InstName}(i)\}$$

By the syntax requirement concerning the uniqueness of messages, it is only necessary to verify whether there is a correspondence at all. This can be formulated in terms of the equivalences $\sim_{\text{dec}}^{\text{out}}$ and $\sim_{\text{dec}}^{\text{in}}$ from Section 4.11.4.

**Definition 86** The predicate $\text{Cor} : \mathcal{L}(\text{<inst def>}) \times \mathcal{L}(\text{<msc>}) \rightarrow \mathcal{B}$ is for all $\text{dec} \in \mathcal{L}(\text{<inst def>})$ and $\text{ch} \in \mathcal{L}(\text{<msc>})$ defined by

$$\text{Cor}(\text{dec}, \text{ch}) = \left( \bigwedge_{\text{out} \in \text{ExtOutputs}(\text{dec})} \bigvee \text{out} \sim_{\text{dec}}^{\text{out}} \text{out}' \right) \land \left( \bigwedge_{\text{in} \in \text{ExtInputs}(\text{dec})} \bigvee \text{in} \sim_{\text{dec}}^{\text{in}} \text{in}' \right) \land \left( \bigwedge_{\text{out} \in \text{EnvOutputs}(\text{ch})} \bigvee \text{out} \sim_{\text{dec}}^{\text{out}} \text{out}' \right) \land \left( \bigwedge_{\text{in} \in \text{EnvInputs}(\text{ch})} \bigvee \text{in} \sim_{\text{dec}}^{\text{in}} \text{in}' \right)$$

Because of the uniqueness rule for messages such a correspondence, if it exists, is unique. Next, the rule for decomposition is considered.

*The ordering of the external communication events of the decomposed instance must be preserved in the corresponding chart [IT93, section 5.2, page 21].*

This means that if two communication events $\text{com}_1$ and $\text{com}_2$ from a decomposed instance $\text{d}$ of chart $\text{ch}_1$ are ordered causally, their corresponding environmental communication events $\text{com'}_1$ and $\text{com'}_2$ in the chart $\text{ch}_2$ with name $\text{d}$ must also be ordered causally. In order to formalize this requirement the causal dependency graphs of both the decomposed instance and the corresponding chart are used.

61
**Definition 87** The predicate $PreservesOrd: \mathcal{L}(<\text{inst def}>) \times \mathcal{L}(<\text{mst}>) \rightarrow \mathcal{B}$ is for all $d \in \mathcal{L}(<\text{inst def}>)$ and $ch \in \mathcal{L}(<\text{mst}>)$ defined by

$$PreservesOrd(d, ch) = \bigwedge_{c_1 \in \text{ExtComms}(d)} \bigvee_{c_2 \in \text{ExtComms}(d)} \bigwedge_{c_1 \xrightarrow{d} c_2} \bigvee_{c_1' \in \text{EnvComms}(ch)} \bigwedge_{c_2' \in \text{EnvComms}(ch)} \bigwedge_{\text{MsgId}(c_1) = \text{MsgId}(c_1')} \bigwedge_{\text{MsgId}(c_2) = \text{MsgId}(c_2')}$$

where the relation $d \rightarrow \subseteq \text{MsgLabel} \times \text{MsgLabel}$ is for all $l_1, l_2 \in \text{MsgLabel}$ defined by

$$l_1 \xrightarrow{d} l_2 \text{ if and only if } (l_1, l_2) \in \text{ICDG}(d)$$

and where the following abbreviations are used

$$\text{ExtComms}(d) = \text{ExtOutputs}(d) \cup \text{ExtInputs}(d)$$
$$\text{EnvComms}(ch) = \text{EnvOutputs}(ch) \cup \text{EnvInputs}(ch)$$

The relation $\xrightarrow{ch}$ is defined in Section 4.8. It gives the ordering specified by the instances of the chart and the implicit orderings via communication and process creation. The symbol $+$ denotes the transitive closure of a relation (see Chapter 3).

The formalization of the syntax requirement is as follows.

**Definition 88** The predicate $Interface: \mathcal{L}(<\text{mac doc}>) \rightarrow \mathcal{B}$ is for all $doc \in \mathcal{L}(<\text{mac doc}>)$ defined by

$$Interface(doc) = \bigwedge_{ch \in \text{Charts}(doc)} \bigwedge_{ch' \in \text{Charts}(doc)} \bigwedge_{d \in \text{Declts}(ch)} \bigwedge_{\text{ChName}(ch') = \text{InstName}(d)} \text{Cor}(d, ch') \land PreservesOrd(d, ch')$$
Bibliography


Appendix A

Concrete textual syntax for Message Sequence Charts

In Table A.1 an overview of the textual syntax for Message Sequence Charts, as introduced piecewise in Chapter 2, is presented. The textual syntax for Message Sequence Charts used is different from the textual syntax as given by Recommendation Z.120 [IT93]. The most important differences are the use of shorter names for nonterminals, and the removal of the operator *. The reason for the removal of the operator * is that this operator does not support the use of inductively defined functions on the languages generated by nonterminals. Also, as was mentioned before, the nonterminal for Sub Message Sequence Charts (<submsc>) is left out.

In Table A.1, the following nonterminals represent identifiers:

<table>
<thead>
<tr>
<th>&lt;at&gt;</th>
<th>&lt;cn&gt;</th>
<th>&lt;dn&gt;</th>
<th>&lt;inst name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;kn&gt;</td>
<td>&lt;min&gt;</td>
<td>&lt;mn&gt;</td>
<td>&lt;msc name&gt;</td>
</tr>
<tr>
<td>&lt;par name&gt;</td>
<td>&lt;text&gt;</td>
<td>&lt;tin&gt;</td>
<td>&lt;tn&gt;</td>
</tr>
</tbody>
</table>

The symbol <> denotes the empty string. The following identifiers are the reserved keywords:

- action
- condition
- endinstance
- env
- instance
- process
- set
- text

<table>
<thead>
<tr>
<th>action</th>
<th>all</th>
<th>block</th>
<th>concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>create</td>
<td>decomposed</td>
<td>endconcurrent</td>
</tr>
<tr>
<td>endinstance</td>
<td>endmsc</td>
<td>endmscdocument</td>
<td>endtext</td>
</tr>
<tr>
<td>env</td>
<td>from</td>
<td>in</td>
<td>inst</td>
</tr>
<tr>
<td>instance</td>
<td>msc</td>
<td>mscdocument</td>
<td>out</td>
</tr>
<tr>
<td>process</td>
<td>related</td>
<td>reset</td>
<td>service</td>
</tr>
<tr>
<td>set</td>
<td>shared</td>
<td>stop</td>
<td>system</td>
</tr>
<tr>
<td>text</td>
<td>timeout</td>
<td>to</td>
<td></td>
</tr>
</tbody>
</table>

65
Table A.1: Alternative grammar for Message Sequence Charts

```
<msc doc> ::= mscedocument <doc head> <doc body>
endmscedocument;
<doc head> ::= <doc name> [ related to <sdl ref> ];
<doc body> ::= <> | <msc> <doc body>
<msc> ::= msc <msc head> <msc body> endmsc;
<msc head> ::= <msc name>; [<msc interface>]
<msc interface> ::= inst <inst list>;
<inst list> ::= <inst name> | [: <inst kind>] [, <inst list>]
<inst kind> ::= [ <type> ] <kn>
<type> ::= system | block | process | service
<msc body> ::= <> | <inst def> <msc body>
| <text def> <msc body>
<inst def> ::= instance <inst head> <inst body> endinstance;
<inst head> ::= <inst name> [: ] <inst kind> [decomposed];
<inst body> ::= <> | stop; | <event> <inst body>
<event> ::= <out> | <in> | <action> | <condition>
| <set> | <reset> | <timeout>
| <create> | <coregion>
<out> ::= out <msgid> to <address>;
<msgid> ::= <mid> [ { <par list> } ]
<mid> ::= <mm> [, , <min>]
<par list> ::= <par name> [, , <par list>]
<address> ::= <inst name> | env
<in> ::= in <msgid> from <address>;
'action> ::= action <at>;
<condition> ::= condition <cn>
[shared { <shared inst list> | all } ];
<shared inst list> ::= <inst name> [, , <shared inst list>]
<set> ::= set <tid> [ { <dn> } ];
<tid> ::= <tn> [, , <tin>]
<reset> ::= reset <tid>;
<timeout> ::= timeout <tid>;
<create> ::= create <inst name> [ { <par list> } ];
<coregion> ::= concurrent <coevents> endconcurrent;
<coevents> ::= <> | <out> <coevents> | <in> <coevents>
<text def> ::= text <note> endtext;
<note> ::= /* <text> */
```
Appendix B

Auxiliary functions and predicates

In this chapter the definitions of the following functions and predicates are given: Address, AllInsts, Charts, ChBody, ChName, CondEvents, CondName, CreateEvents, CreateName, DocBody, Inputs, InstBody, InstHead, InstName, MsgId, Outputs.

The definitions are presented in alphabetic order.

**Definition 89** The function \( \text{Address} : \mathcal{L}(< \text{in} > | < \text{out} >) \rightarrow \mathcal{L}(< \text{address} >) \) is for all \( \text{msgid} \in \mathcal{L}(< \text{msgid} >) \) and \( \text{address} \in \mathcal{L}(< \text{address} >) \) defined by

\[
\text{Address}(\text{out}\ \text{msgid} \text{ to } \text{address};) = \text{address} \\
\text{Address}(\text{in}\ \text{msgid} \text{ from } \text{address};) = \text{address}
\]

The function \( \text{AllInsts} \) determines the multiset of all instances which are specified within a chart body and chart respectively.

**Definition 90** The function \( \text{AllInsts} : \mathcal{L}(< \text{msc body} >) \rightarrow \mathcal{M}(\mathcal{L}(< \text{inst def} >)) \) is for all \( i \in \mathcal{L}(< \text{inst def} >), \text{textdef} \in \mathcal{L}(< \text{text def} >) \) and \( \text{mscbody} \in \mathcal{L}(< \text{msc body} >) \) defined by

\[
\text{AllInsts}(<> ) = \emptyset \\
\text{AllInsts}(i \ \text{mscbody}) = [i] \cup \text{AllInsts}(\text{mscbody}) \\
\text{AllInsts}(\text{textdef} \ \text{mscbody}) = \text{AllInsts}(\text{mscbody})
\]

The function \( \text{AllInsts} : \mathcal{L}(< \text{msc} >) \rightarrow \mathcal{M}(\mathcal{L}(< \text{inst def} >)) \) is for all \( \text{ch} \in \mathcal{L}(< \text{msc} >) \) defined by

\[
\text{AllInsts}(\text{ch}) = \text{AllInsts}(\text{ChBody}(\text{ch}))
\]

**Definition 91** The function \( \text{Charts} : \mathcal{L}(< \text{doc body} >) \rightarrow \mathcal{M}(\mathcal{L}(< \text{msc} >)) \) is for all \( \text{ch} \in \mathcal{L}(< \text{msc} >) \) and \( \text{db} \in \mathcal{L}(< \text{doc body} >) \) defined by

\[
\text{Charts}(<> ) = \emptyset \\
\text{Charts}(\text{ch} \ \text{db}) = [\text{ch}] \cup \text{Charts}(\text{db})
\]
The function $Charts : \mathcal{L}(\langle msc \, doc \rangle) \rightarrow \mathcal{M}(\mathcal{L}(\langle msc \rangle))$ is for all $doc \in \mathcal{L}(\langle msc \, doc \rangle)$ defined by

$$Charts(doc) = Charts(DocBody(doc))$$

**Definition 92** The function $ChBody : \mathcal{L}(\langle msc \rangle) \rightarrow \mathcal{L}(\langle msc \, name \rangle)$ is for all $msc\_head \in \mathcal{L}(\langle msc \, head \rangle)$ and $msc\_body \in \mathcal{L}(\langle msc \, body \rangle)$ defined by

$$ChBody(msc \, msch\_head, msc\_body \, endmsc;) = msc\_body$$

**Definition 93** The function $ChName : \mathcal{L}(\langle msc \rangle) \rightarrow \mathcal{L}(\langle msc \, name \rangle)$ is for all $msc\_name \in \mathcal{L}(\langle msc \, name \rangle)$, $msc\_interface \in \mathcal{L}(\langle msc \, interface \rangle)$ and $msc\_body \in \mathcal{L}(\langle msc \, body \rangle)$ defined by

$$ChName(msc \, msc\_name, msc\_body \, endmsc;) = msc\_name$$

**Definition 94** The function $CondEvents \in \mathcal{L}(\langle inst \, body \rangle) \rightarrow \mathcal{M}(\mathcal{L}(\langle condition \rangle))$ is for all $e \in \mathcal{L}(\langle condition \rangle)$ and $ib \in \mathcal{L}(\langle inst \, body \rangle)$ defined by

$$CondEvents(\langle \rangle) = \emptyset$$
$$CondEvents(stop;) = \emptyset$$
$$CondEvents(e \, ib) = CondEvents(ib) \; \text{if} \; e \notin \mathcal{L}(\langle condition \rangle)$$
$$CondEvents(e \, ib) = [e] \cup CondEvents(ib) \; \text{if} \; e \in \mathcal{L}(\langle condition \rangle)$$

The function $CondEvents \in \mathcal{L}(\langle inst \, def \rangle) \rightarrow \mathcal{M}(\mathcal{L}(\langle condition \rangle))$ is for all $ide \in \mathcal{L}(\langle inst \, def \rangle)$ defined by

$$CondEvents(ide) = CondEvents(InstBody(ide))$$

The function $CondEvents \in \mathcal{L}(\langle msc \rangle) \rightarrow \mathcal{M}(\mathcal{L}(\langle condition \rangle))$ is for all $ch \in \mathcal{L}(\langle msc \rangle)$ defined by

$$CondEvents(ch) = \bigcup_{i \in \text{AllInsts}(ch)} CondEvents(i)$$

**Definition 95** The function $CondName : \mathcal{L}(\langle condition \rangle) \rightarrow \mathcal{L}(\langle cn \rangle)$ is for all $cn \in \mathcal{L}(\langle cn \rangle)$ and $list \in \mathcal{L}(\langle \text{shared inst list} \rangle)$ defined by

$$CondName(condition \, cn;) = cn$$
$$CondName(condition \, cn \; \text{shared} \; all;) = cn$$
$$CondName(condition \, cn \; \text{shared} \; list;) = cn$$

**Definition 96** The function $CreateEvents : \mathcal{L}(\langle inst \, body \rangle) \rightarrow \mathcal{M}(\mathcal{L}(\langle create \rangle))$ is for all $e \in \mathcal{L}(\langle event \rangle)$ and $ib \in \mathcal{L}(\langle inst \, body \rangle)$ defined by
CreateEvents(<>) = \emptyset
CreateEvents(stop;) = \emptyset
CreateEvents(e \text{ ib}) = \begin{cases} 
\text{CreateEvents(ib)} & \text{if } e \notin \mathcal{L}(<\text{create}>) \\
[e] \cup \text{CreateEvents(ib)} & \text{if } e \in \mathcal{L}(<\text{create}>) 
\end{cases}

The function CreateEvents : \mathcal{L}(<\text{inst def}>) \to \IM(\mathcal{L}(<\text{create}>)) is for \( i \in \mathcal{L}(<\text{inst def}>) \) defined by

\[ \text{CreateEvents}(i) = \text{CreateEvents}(\text{InstBody}(i)) \]

The function CreateEvents : \mathcal{L}(<\text{msc}>) \to \IM(\mathcal{L}(<\text{create}>)) is for all \( ch \in \mathcal{L}(<\text{msc}>) \) defined by

\[ \text{CreateEvents}(ch) = \bigcup_{i \in \text{AllInsts}(ch)} \text{CreateEvents}(i) \]

**Definition 97** The function CreateName \( \in \mathcal{L}(<\text{create}>) \to \mathcal{L}(<\text{inst name}>) \) is for all \( \text{instname} \in \mathcal{L}(<\text{inst name}>) \) and \( p \in \mathcal{L}(<\text{par list}>) \) defined by

\[ \text{CreateName}(\text{create instname};) = \text{instname} \]
\[ \text{CreateName}(\text{create instname}(p);) = \text{instname} \]

**Definition 98** The function DocBody : \mathcal{L}(<\text{msc doc}>) \to \mathcal{L}(<\text{doc body}>) is for all \( \text{dochead} \in \mathcal{L}(<\text{doc head}>) \) and \( db \in \mathcal{L}(<\text{doc body}>) \) defined by

\[ \text{DocBody}(\text{msc document dochead db endmsc document;};) = db \]

**Definition 99** The function Inputs : \mathcal{L}(<\text{coevents}>) \to \IM(\mathcal{L}(<\text{in}>) \) is for \( \text{out} \in \mathcal{L}(<\text{out}>) \), \( \text{in} \in \mathcal{L}(<\text{in}>) \) and \( \text{coevents} \in \mathcal{L}(<\text{coevents}>) \) defined by

\[ \text{Inputs}(<>) = \emptyset \]
\[ \text{Inputs}(\text{out coevents}) = \text{Inputs}(\text{coevents}) \]
\[ \text{Inputs}(\text{in coevents}) = [\text{in}] \cup \text{Inputs}(\text{coevents}) \]

The function Inputs : \mathcal{L}(<\text{coregion}>) \to \IM(\mathcal{L}(<\text{in}>) \) is for all \( \text{coevents} \in \mathcal{L}(<\text{coevents}>) \) defined by

\[ \text{Inputs}(\text{concurrent coevents endconcurrent;};) = \text{Inputs}(\text{coevents}) \]

The function Inputs : \mathcal{L}(<\text{event}>) \to \IM(\mathcal{L}(<\text{in}>) \) is for all \( e \in \mathcal{L}(<\text{event}>) \) defined by

\[ \text{Inputs}(e) = \begin{cases} 
\emptyset & \text{if } e \notin \mathcal{L}(<\text{in}>) \cup <\text{coregion}> \\
[e] & \text{if } e \in \mathcal{L}(<\text{in}>) \\
\text{Inputs}'(e) & \text{if } e \in \mathcal{L}(<\text{coregion}>) 
\end{cases} \]
The function $\text{Inputs}'$ is identical to the function $\text{Inputs} : \mathcal{L}(<\text{coregion}>) \to \mathcal{M}(\mathcal{L}(<\text{in}>)$, and is therefore omitted.

The function $\text{Inputs} : \mathcal{L}(<\text{inst body}>) \to \mathcal{M}(\mathcal{L}(<\text{in}>)$ is for all $e \in \mathcal{L}(<\text{event}>)$ and $ib \in \mathcal{L}(<\text{inst body}>)$ defined by

\[
\text{Inputs}(<>) = \Box \\
\text{Inputs}(\text{stop};) = \Box \\
\text{Inputs}(e \text{ } ib) = \text{Inputs}(e) \cup \text{Inputs}(ib)
\]

The function $\text{Inputs} : \mathcal{L}(<\text{inst def}>) \to \mathcal{M}(\mathcal{L}(<\text{in}>)$ is for all $i \in \mathcal{L}(<\text{inst def}>)$ defined by

\[
\text{Inputs}(i) = \text{Inputs}(\text{InstBody}(i))
\]

The function $\text{Inputs} : \mathcal{L}(<\text{msc}>) \to \mathcal{M}(\mathcal{L}(<\text{in}>)$ is for all $ch \in \mathcal{L}(<\text{msc}>)$ defined by

\[
\text{Inputs}(ch) = \bigcup_{i \in \text{AllInsts}(ch)} \text{Inputs}(i)
\]

**Definition 100** The function $\text{InstBody} : \mathcal{L}(<\text{inst def}>) \to \mathcal{L}(<\text{inst body}>)$ is for $\text{insthead} \in \mathcal{L}(<\text{inst head}>)$ and $ib \in \mathcal{L}(<\text{inst body}>)$ defined by

\[
\text{InstBody}(\text{instance insthead } ib \text{ endinstance};) = ib
\]

**Definition 101** The function $\text{InstHead} : \mathcal{L}(<\text{inst def}>) \to \mathcal{L}(<\text{inst head}>)$ is for $\text{insthead} \in \mathcal{L}(<\text{inst head}>)$ and $ib \in \mathcal{L}(<\text{inst body}>)$ defined by

\[
\text{InstHead}(\text{instance insthead } ib \text{ endinstance};) = \text{insthead}
\]

**Definition 102** The function $\text{InstName} : \mathcal{L}(<\text{inst def}>) \to \mathcal{L}(<\text{inst name}>)$ is for $\text{instname} \in \mathcal{L}(<\text{inst name}>)$, $\text{kind} \in \mathcal{L}(<\text{inst kind}>)$ and $ib \in \mathcal{L}(<\text{inst body}>)$ defined by

\[
\begin{align*}
\text{InstName}(\text{instance instname; } ib \text{ endinstance};) &= \text{instname} \\
\text{InstName}(\text{instance instname kind; } ib \text{ endinstance};) &= \text{instname} \\
\text{InstName}(\text{instance instname decomposed; } ib \text{ endinstance};) &= \text{instname} \\
\text{InstName}(\text{instance instname kind decomposed; } ib \text{ endinstance};) &= \text{instname}
\end{align*}
\]

**Definition 103** The function $\text{MsgId} : \mathcal{L}(<\text{in} > | <\text{out}>) \to \mathcal{L}(<\text{msgid}>)$ is for all $\text{msgid} \in \mathcal{L}(<\text{msgid}>)$ and $\text{address} \in \mathcal{L}(<\text{address}>)$ defined by

\[
\begin{align*}
\text{MsgId}(\text{out } \text{msgid to address;}) &= \text{msgid} \\
\text{MsgId}(\text{in } \text{msgid from address;}) &= \text{msgid}
\end{align*}
\]
**Definition 104** The function $\text{Outputs} : \mathcal{L}(\langle \text{coevents} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$ is for all $\text{out} \in \mathcal{L}(\langle \text{out} \rangle), \text{in} \in \mathcal{L}(\langle \text{in} \rangle)$ and $\text{coevents} \in \mathcal{L}(\langle \text{coevents} \rangle)$ defined by

$$
\begin{align*}
\text{Outputs}(\langle \rangle) &= \Box \\
\text{Outputs}(\text{out} \ \text{coevents}) &= [\text{out}] \cup \text{Outputs}(\text{coevents}) \\
\text{Outputs}(\text{in} \ \text{coevents}) &= \text{Outputs}(\text{coevents})
\end{align*}
$$

The function $\text{Outputs} : \mathcal{L}(\langle \text{coregion} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$ is for all $\text{co} \in \mathcal{L}(\langle \text{coevents} \rangle)$ defined by

$$
\text{Outputs}(\text{concurrent} \ \text{co} \ \text{endconcurrent};) = \text{Outputs}(\text{co})
$$

The function $\text{Outputs} : \mathcal{L}(\langle \text{event} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$ is for all $\text{e} \in \mathcal{L}(\langle \text{event} \rangle)$ defined by

$$
\text{Outputs}(\text{e}) = \begin{cases} 
\Box & \text{if } \text{e} \notin \mathcal{L}(\langle \text{out} \rangle \mid \langle \text{coregion} \rangle) \\
[\text{e}] & \text{if } \text{e} \in \mathcal{L}(\langle \text{out} \rangle) \\
\text{Outputs'}(\text{e}) & \text{if } \text{e} \in \mathcal{L}(\langle \text{coregion} \rangle)
\end{cases}
$$

The auxiliary function $\text{Outputs'}$ is identical to the function $\text{Outputs} : \mathcal{L}(\langle \text{coregion} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$, and is therefore omitted.

The function $\text{Outputs} : \mathcal{L}(\langle \text{inst body} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$ is for all $\text{e} \in \mathcal{L}(\langle \text{event} \rangle)$ and $\text{ib} \in \mathcal{L}(\langle \text{inst body} \rangle)$ defined by

$$
\begin{align*}
\text{Outputs}(\langle \rangle) &= \Box \\
\text{Outputs}(\text{stop};) &= \Box \\
\text{Outputs}(\text{e} \ \text{ib}) &= \text{Outputs}(\text{e}) \cup \text{Outputs}(\text{ib})
\end{align*}
$$

The function $\text{Outputs} : \mathcal{L}(\langle \text{inst def} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$ is for all $\text{i} \in \mathcal{L}(\langle \text{inst def} \rangle)$ defined by

$$
\text{Outputs}(\text{i}) = \text{Outputs}(\text{InstBody}(\text{i}))
$$

The function $\text{Outputs} : \mathcal{L}(\langle \text{msc} \rangle) \to \mathcal{M}(\mathcal{L}(\langle \text{out} \rangle))$ is for all $\text{ch} \in \mathcal{L}(\langle \text{msc} \rangle)$ defined by

$$
\text{Outputs}(\text{ch}) = \bigcup_{i \in \text{AllInsts}(\text{ch})} \text{Outputs}(\text{i})
$$
Computing Science Reports

In this series appeared:

93/01  R. van Geldrop
93/02  T. Verhoef
93/03  T. Verhoef
93/04  E.H.L. Aarts J.H.M. Korst P.J. Zwietering
93/05  J.C.M. Baeten C. Verhoef
93/06  J.P. Veltkamp
93/07  P.D. Moerland
93/08  J. Verhoosel
93/09  K.M. van Hee
93/10  K.M. van Hee
93/11  K.M. van Hee
93/12  K.M. van Hee
93/13  K.M. van Hee
93/14  J.C.M. Baeten J.A. Bergstra
93/15  J.C.M. Baeten J.A. Bergstra R.N. Bol
93/16  H. Schepers J. Hoeman
93/17  D. Alstein P. van der Stok
93/18  C. Verhoef
93/19  G-J. Houben
93/20  F.S. de Boer
93/21  M. Codish D. Dams G. Filé M. Bruynooghe
93/22  E. Poll
93/23  E. de Kogel
93/24  E. Poll and Paula Severi
93/25  H. Schepers and R. Gerth
93/26  W.M.P. van der Aalst
93/27  T. Klok and D. Kratsch
93/28  F. Kamareddine and R. Nederpelt
93/29  R. Post and P. De Bruijn
93/30  J. Deogun T. Klok D. Kratsch H. Müller

Department of Mathematics and Computing Science
Eindhoven University of Technology

Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.

A continuous version of the Prisoner's Dilemma, p. 17.

Quicksort for linked lists, p. 8.

Deterministic and randomized local search, p. 78.

A congruence theorem for structured operational semantics with predicates, p. 18.

On the unavoidability of metastable behaviour, p. 29.

Exercises in Multiprogramming, p. 97.

A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.


Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.


A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27.

Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.

A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.

The Design of an Online Help Facility for ExSpect, p. 21.


Freeness Analysis for Logic Programs - And Correctness, p. 24.

A Typechecker for Bijective Pure Type Systems, p. 28.

Relational Algebra and Equational Proofs, p. 23.

Pure Type Systems with Definitions, p. 38.


Multi-dimensional Petri nets, p. 25.

Finding all minimal separators of a graph, p. 11.

A Semantics for a finite \( \lambda \)-calculus with de Bruijn indices, p. 49.

GOLD, a Graph Oriented Language for Databases, p. 42.

On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31 W. Körver  Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.


93/33 L. Leyens and J. Moonen  ILIAS, a sequential language for parallel matrix computations, p. 20.

93/34 J.C.M. Baeten and J.A. Bergstra  Real Time Process Algebra with Infinitesimals, p.39.

93/35 W. Ferrer and P. Seven  Abstract Reduction and Topology, p. 28.

93/36 J.C.M. Baeten and J.A. Bergstra  Non Interleaving Process Algebra, p. 17.


93/38 C. Verhoef  A general conservative extension theorem in process algebra, p. 17.


93/41 A. Bijlsma  Temporal operators viewed as predicate transformers, p. 11.

93/42 P.M.P. Rambags  Automatic Verification of Regular Protocols in P/T Nets, p. 23.

93/43 B.W. Watson  A taxonomy of finite automata construction algorithms, p. 87.

93/44 B.W. Watson  A taxonomy of finite automata minimization algorithms, p. 23.


93/48 R. Gerth  Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

94/01 P. America M. van der Kammen R.P. Nederpelt O.S. van Rooyen K.H. de Swart  The object-oriented paradigm, p. 28.

94/02 F. Kamareddine R.P. Nederpelt  Canonical typing and $\Pi$-conversion, p. 51.


94/04 J.C.M. Baeten J.A. Bergstra  Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.


94/06 T. Basten T. Kunz J. Blok M. Coffin D. Taylor  Time and the Order of Abstract Events in Distributed Computations, p. 29.


94/08 O.S. van Rooyen K.H. de Swart  A Hierarchical Diagrammatic Representation of Class Structure, p. 22.

94/09 J.C.M. Baeten J.A. Bergstra  Process Algebra with Partial Choice, p. 16.
| 94/10 | T. Verheoff | The testing Paradigm Applied to Network Structure, p. 31. |
| 94/11 | J. Peleska  |
|       | C. Huizing  |
| 94/12 | T. Kloks    |
|       | D. Kratsch  |
| 94/13 | R. Seflie   | A New Method for Integrity Constraint checking in Deductive Databases, p. 34. |
| 94/14 | W. Peremans |
| 94/15 | R.J.M. Vaessens |
|       | E.H.L. Aarts |
|       | J.K. Lenstra | Ups and Downs of Type Theory, p. 9. |
| 94/16 | R.C. Backhouse |
|       | H. Doornbos  | Job Shop Scheduling by Local Search, p. 21. |
| 94/17 | S. Mauw     |
|       | M.A. Reniers | Mathematical Induction Made Calculational, p. 36. |
| 94/18 | F. Kamareddine |
|       | R. Nederpelt | An Algebraic Semantics of Basic Message Sequence Charts, p. 9. |
| 94/19 | B.W. Watson |
|       | D. Dams     | Refining Reduction in the Lambda Calculus, p. 15. |
| 94/20 | R. Bloo     |
|       | F. Kamareddine |
|       | R. Nederpelt | The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46. |
| 94/21 | B.W. Watson |
|       | R.R. Hoogerwoord | Beyond  \( \beta \)-Reduction in Church's \( \lambda \rightarrow \), p. 22. |
| 94/22 | B.W. Watson |
|       | S. Mauw and M.A. Reniers | An algebraic semantics of Message Sequence Charts, p. 43. |
| 94/23 | C.W.A.M. van Overveld |
|       | M. Verhoeven | Abstract Interpretation of Reactive Systems: Abstractions Preserving VCTL*, 3CTL* and CTL*, p. 28. |
| 94/24 | J. Hooman   |
|       | J.A. Bergstra |
|       | G. Stéphanescu | K\(\omega\)-free and W\(\omega\)-free graphs, p. 10. |
| 94/25 | J.C.M. Baeten |
|       | J.A. Bergstra |
|       | G. Stéphanescu | On the foundations of functional programming: a programmer's point of view, p. 54. |
| 94/26 | C.W.A.M. van Overveld |
|       | M. Verhoeven | Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20. |
| 94/27 | J. Hooman   |
|       | J.A. Bergstra |
|       | G. Stéphanescu | Correctness of Real Time Systems by Construction, p. 22. |
| 94/28 | J.C.M. Baeten |
|       | J.A. Bergstra |
|       | G. Stéphanescu | Process Algebra with Feedback, p. 22. |
| 94/29 | B.W. Watson |
|       | R.E. Watson  | A Boyer-Moore type algorithm for regular expression pattern matching, p. 22. |
| 94/31 | T. Laan      | A formalization of the Ramified Type Theory, p.40. |
| 94/32 | R. Bloo      |
|       | F. Kamareddine |
|       | R. Nederpelt | The Barendregt Cube with Definitions and Generalised Reduction, p. 37. |
| 94/33 | J.C.M. Baeten |
|       | S. Mauw     |
|       | S. Mauw     | Delayed choice: an operator for joining Message Sequence Charts, p. 15. |
| 94/34 | F. Kamareddine |
|       | R. Nederpelt | Canonical typing and II-conversion in the Barendregt Cube, p. 19. |
| 94/35 | T. Basten   |
|       | B. Boi      | Simulating and Analyzing Railway Interlockings in ExSpect, p. 30. |
| 94/36 | A. Bijlsma  |
|       | C.S. Scholien | Point-free substitution, p. 10. |
On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160.

On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160.

On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160.
Deductive Database Systems and integrity constraint checking, p. 36.
Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
De proceedings: ACP'95, p.
On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
Specification of tools for Message Sequence Charts, p. 36.
Discrete Time Process Algebra with Abstraction, p. 15.
On Normalisation, p. 33.
Axiomatizing Early and Late Input by Variable Elimination, p. 44.
Petri net based scheduling, p. 20.
Synchronous Sequence Charts In Action, p. 36.
A Class of Petri nets for modeling and analyzing business processes, p. 24.
A Conservative Look at term Deduction Systems with Variable Binding, p. 29.
Practical Symbolic Model Checking of the full m-calculus using Compositional Abstractions, p. 17.
Handboek simulatie, p. 51.
Context-Free Graph Grammars and Concatenation of Graphs, p. 35.
Record concatenation with intersection types, p. 46.
An algebraic semantics for hierarchical P/T Nets, p. 32.
Process Algebra with Autonomous Actions, p. 12.
Multi-User Publishing in the Web: DroSS, A Document Repository Service Station, p. 12
Example specifications in phi-SDL.
A Process-Algebraic Approach to Life-Cycle Inheritance
Inheritance = Encapsulation + Abstraction, p. 15.
Life-Cycle Inheritance
A Petri-Net-Based Approach, p. 18.
Structural Petri Net Equivalence, p. 16.
AUTOMATH and Pure Type Systems, p. 30.
A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
Prisorean Tense Logics in Modal Pure Type Systems, p. 61.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>96/14</td>
<td>S.H.J. Bos and M.A. Reniers</td>
<td>The $I^2$ C-bus in Discrete-Time Process Algebra</td>
<td>25</td>
</tr>
<tr>
<td>96/15</td>
<td>M.A. Reniers and J.J. Vereijken</td>
<td>Completeness in Discrete-Time Process Algebra</td>
<td>139</td>
</tr>
<tr>
<td>96/16</td>
<td>P. Hoogendijk and O. de Moor</td>
<td>What is a data type?</td>
<td>29</td>
</tr>
<tr>
<td>96/17</td>
<td>E. Boiten and P. Hoogendijk</td>
<td>Nested collections and polytypism</td>
<td>11</td>
</tr>
<tr>
<td>96/18</td>
<td>P.D.V. van der Stok</td>
<td>Real-Time Distributed Concurrency Control Algorithms with mixed time constraints</td>
<td>71</td>
</tr>
</tbody>
</table>