Syntax Requirements of Message Sequence Charts

M.A. Reniers [a]

[a]Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands

A set of syntax requirements of MSC is discussed and formalized. The treatment is restricted to Basic MSCs without conditions. Syntax requirements of communication events in Basic MSCs are discussed and formalized. The formalization is syntax-directed and based on functions and predicates.

1. INTRODUCTION

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 [1] 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. To enhance tool support, feasibility of Sequence Chart exchange between tools, and harmonization of the use of Sequence Charts within ITU Study Groups, a standardization of such Sequence Charts was proposed by the ITU (the former CCITT). The recommended version of Sequence Charts is called *Message Sequence Charts (MSC)*.

Recommendation Z.120 Messages sequence chart (MSC) [2] contains a description of an abstract syntax, a graphical syntax, and a textual syntax of the language MSC. Besides these syntax descriptions also an informal semantics and an informal description of the syntax requirements are given. The formal semantics of MSC is standardized in Annex B to Recommendation Z.120 [3].

The purpose of this paper is to discuss the syntax requirements of MSC. The description of the syntax requirements as presented in [2] is open to ambiguous interpretation and should therefore be formalized. We restrict the treatment to the core language of MSC without conditions, called *Basic MSC*. The reason for this restriction is, besides the limited space available, that Basic MSCs are easily understood and so are their syntax requirements. For the definition of a formal semantics of Basic MSC we refer to [4]. For a more elaborate treatment of the formalization of syntax requirements of MSC we refer to [6]. The syntax requirements we treat in this paper all concern the communication events of MSCs. We formalize a reference rule, a uniqueness rule, a completeness rule, and a causal dependency rule for communication events. As a starting point we take

the informal descriptions of these requirements from Recommendation Z.120. The formal syntax requirements are stated in terms of predicates and functions on the textual syntax of Basic MSC. Advantages of the use of predicates and functions are that they are universally known and that they have applications in almost every area of computing science. In general a translation of the formalization based on functions and predicate logic to description methods as ASF/SDF, Z, VDM, PSF, etc. is straightforward. An implementation of the syntax requirements and formal semantics of Basic MSC is given in [5]. A disadvantage is the number of predicates and functions needed in formalizing even the simplest syntax requirements. However, most of the auxiliary functions and predicates are defined in order to obtain information from the textual syntax. When formalizing syntax requirements for the complete language MSC, most of these are reused.

The paper is structured as follows. In Section 2 a short introduction to MSC is given. In Section 3 we recapitulate some basic notions on relations and multisets. Those will be used frequently in the formalization of the syntax requirements. In Section 4 the syntax requirements of MSC are discussed and formalized.

Acknowledgements

I would like to thank Anders Ek, Øystein Haugen, and Ekkart Rudolph for their contributions to the discussion on syntax requirements in Geneva. Without these discussions this work would be far less interesting, not only to me but also to the formalization efforts of Study Group 10 of the ITU. Special thanks go to Sjouke Mauw for his efforts to have me write 'readable' papers and efforts to keep up with my changes to the paper. The anonymous reviewers are acknowledged for their comments and criticism.

2. BASIC MESSAGE SEQUENCE CHARTS

A Basic MSC is a finite collection of instances. An instance is an abstract entity on which message outputs and message inputs 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 message between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. A message is divided 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 MSC. A message received from the environment is represented by an arrow from the exterior of the MSC to the receiving instance.

Example 1 Consider the messages $m1$, $m2$, $m3$ and $m4$ in Figure 1. Message $m0$ is sent to the environment. The behaviour of the environment is not specified.

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 1 this implies that message $m3$ is received by $i4$ only after it has been sent by $i3$, and, consequently, after the consumption of $m2$ by $i3$. Thus the events concerning $m1$ and $m3$ are ordered in time, while for the events of $m4$ and $m3$ no order is specified apart from the requirement that the

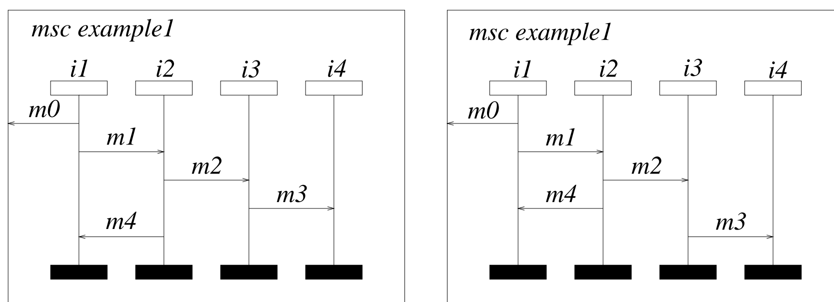

Figure 1. Example Basic MSCs

output of a message occurs before its input. The second Basic MSC in Figure 1 defines the same Basic MSC (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 $m3$.

Although the application of MSC is mainly focussed on the graphical representation, they have a concrete textual syntax. This representation was originally intended for exchanging MSCs 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 MSC is instance oriented. This means that a Basic MSC 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 MSCs of Figure 1 have the following textual representation.

```
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;
      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 MSC is given in Table 1. The nonterminals `<inst name>`, `<msc name>`, and `<msgid>` represent identifiers. The symbol `<>` denotes the empty string. The following identifiers are reserved keywords: `endinstance`, `endmsc`, `env`, `from`, `in`, `instance`, `msc`, `out`, and `to`. The language generated by a nonterminal `X` in the grammar of Table 1 will be denoted by $\mathcal{L}(\mathtt{X})$.

Table 1
The concrete textual syntax of Basic MSC

```
<msc>         ::=  msc <msc name> ; <msc body> endmsc;
<msc body>    ::=  <> | <inst def> <msc body>
<inst def>    ::=  instance <inst name> ; <inst body> endinstance;
<inst body>   ::=  <> | <event> <inst body>
<event>       ::=  <out> | <in>
<out>         ::=  out <msgid> to <address>;
<address>     ::=  <inst name> | env
<in>          ::=  in <msgid> from <address>;
```

3. PRELIMINARIES

Before we turn to the discussion and formalization of the syntax requirements of MSC, we first introduce some basic notions on relations and multisets. Those will be used frequently in the formalization. A binary relation on a set A is a subset of $A \times A$. In this paper we will only consider binary relations. Therefore, the adjective binary is left implicit in the remainder. Next, we introduce some special relations. These are all well known from literature. Let $R \subseteq A \times A$ be a relation.

1) R is *reflexive* if for all $a \in A$: $(a, a) \in R$.
2) R is *symmetric* if for all $a, b \in A$: if $(a, b) \in R$, then $(b, a) \in R$.
3) R is *transitive* if for all $a, b, c \in A$: if $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$.
4) R is *strict* if for all $a \in A$: $(a, a) \notin R$.
5) R is an *equivalence relation* if it is reflexive, symmetric, 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 $(a, c) \in R^+$.

Multisets are a generalization of sets, by allowing elements to have multiple occurrences. 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 either one of $[a, a, b]$, $[a, b, a]$, or $[b, a, a]$. The empty multiset is denoted by $\boxplus$. The union of two multisets M and N is denoted by $M \sqcup N$. The membership test is denoted by $\sqsubseteq$. For a set A, the set of all multisets over A is denoted by $\mathbb{M}(A)$. Let A be a set and let $a \in A$ and $\sim$ an equivalence relation on A. For $M \in \mathbb{M}(A)$, $\sharp^a_\sim(M)$ denotes the number of elements from M that are $\sim$-equivalent to a, and is defined inductively by

$$
\begin{aligned}
\sharp^a_\sim(\boxplus) &= 0 \\
\sharp^a_\sim([b] \sqcup M) &= \begin{cases} \sharp^a_\sim(M) & \text{if } a \not\sim b, \\ 1 + \sharp^a_\sim(M) & \text{if } a \sim b. \end{cases}
\end{aligned}
$$

Example 2 Suppose that we are given a multiset $M = [0,0,1,2,2,2]$ over the natural numbers $I\!N$. Denote the equality relation on $I\!N$ by $=$. Then we have, for $n \in I\!N$, the following equations.

$\sharp^0_=(M) = 2 \qquad \sharp^1_=(M) = 1 \qquad \sharp^2_=(M) = 3 \qquad \sharp^{n+3}_=(M) = 0$

4. SYNTAX REQUIREMENTS FOR MESSAGES

The rules for messages which will be discussed in this section express properties such as references to instances, the unambiguous connection of message outputs and message inputs, the completeness of message specification, and the absence of conflicts in the order in which message sending and message reception must be dealt with. Before we turn to these syntax requirements we will discuss messages.

4.1. Abstract messages

A message is completely determined by its sender instance, its receiver instance and its message identifier. Therefore, a message can be represented by a triple which consists of these three identifiers. Such a triple will be called an *abstract message*. A type *Msg* is defined from which the elements represent abstract messages.

Definition 1 *The type Msg is defined by*

$Msg = \mathcal{L}(\texttt{<address>}) \times \mathcal{L}(\texttt{<address>}) \times \mathcal{L}(\texttt{<msgid>})$

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 name of an instance from its definition the function *InstName* is used. To obtain the address specification and the message identifier from a given communication event the functions *Addr* and *MsgId* are used. Their definitions can be found in Appendix A. Given the instance a communication event is specified on, it is possible to determine the abstract message which corresponds with the communication event.

Definition 2 *Let* $i \in \mathcal{L}(\texttt{<inst def>})$. *The function* $Message(i) : \mathcal{L}(\texttt{<out>} \mid \texttt{<in>}) \rightarrow Msg$ *is for all* $out \in \mathcal{L}(\texttt{<out>})$ *and* $in \in \mathcal{L}(\texttt{<in>})$ *defined by*

$$\begin{aligned} Message(i)(out) &= (InstName(i), Addr(out), MsgId(out)) \\ Message(i)(in) &= (Addr(in), InstName(i), MsgId(in)) \end{aligned}$$

Example 3 The abstract message associated to the communication $m4$ between instances $i2$ and $i1$ in Basic MSC *example*1 from Figure 1 is given by $(i2, i1, m4)$.

4.2. References to instances

The instances that are referenced by the communication events are given by the address specification parts of the communication events. The communication events which are sent to or received from the environment do not reference an instance. These communication events are called *external*, whereas communication events between instances are called

internal. Since the external communication events do not reference instances we only need to consider internal communication events.

Only instances which are specified within a chart may be referenced by the communication events of that chart. In order to formalize this requirement we define functions *DeclInstNames* and *IntAddrSpec* which determine the names of the declared instances and the names of the instances referenced by the communication events respectively.

Definition 3 *The function* $DeclInstNames : \mathcal{L}(\texttt{<msc>}) \rightarrow \mathbb{P}(\mathcal{L}(\texttt{<inst name>}))$ *is, for all* $ch \in \mathcal{L}(\texttt{<msc>})$*, defined by* $DeclInstNames(ch) = \{InstName(i) \mid i \sqsubseteq AllInsts(ch)\}$*. The function* $IntAddrSpec : \mathcal{L}(\texttt{<msc>}) \rightarrow \mathbb{P}(\mathcal{L}(\texttt{<inst name>}))$ *is, for all* $ch \in \mathcal{L}(\texttt{<msc>})$*, defined by* $IntAddrSpec(ch) = \{Addr(com) \mid com \sqsubseteq Outputs(ch) \sqcup Inputs(ch) \wedge Addr(com) \neq env\}$.

The functions *Outputs* and *Inputs* (see Appendix A) collect all message output events and message input events from an instance in a multiset and the function *AllInsts* collects all instance definitions of a Basic MSC in a multiset. Then the syntax requirement is formulated as follows: $IntAddrSpec(ch) \subseteq DeclInstNames(ch)$.

4.3. Uniqueness of messages

An internal message is divided 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.

```
msc example2;                         instance j;
   instance i;                           in m from i;
      out m to j;                        in m from i;
      out m to j;                     endinstance;
   endinstance;                    endmsc;
```

Within this chart it is not clear which message output corresponds to which message input. One could associate either one of the Basic MSCs shown in Figure 2 to this textual description.

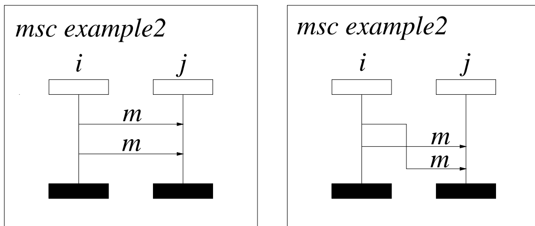

Figure 2. Two graphical versions of Basic MSC *example*2

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 that there

are no two message inputs which refer to the same abstract message. We will consider message outputs only. The formalization of the requirement for message inputs follows the same line. Two message output events which are defined on different instances cannot concern the same abstract message. Therefore, the syntax requirement is formulated as follows.

> *On an instance there must not be two or more message outputs with the same address specification and the same message identifier.*

First, an equivalence relation $\sim$ is defined on message output events. Two message output events are output equivalent (or $\sim$-equivalent) if both the message identifier and the address specification are identical.

Definition 4 *The relation* $\sim \subseteq \mathcal{L}(\texttt{<out>}) \times \mathcal{L}(\texttt{<out>})$ *is for all* $o, o' \in \mathcal{L}(\texttt{<out>})$ *defined by* $o \sim o'$ *iff* $MsgId(o) = MsgId(o') \wedge Addr(o) = Addr(o')$.

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 they are $\sim$-equivalent. Two message outputs which are specified on different instances do not refer to the same abstract message.

Definition 5 *The predicate* $UMO : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathbb{B}$ *is for all* $i \in \mathcal{L}(\texttt{<inst def>})$ *defined by* $UMO(i) = \forall_{out \in Outputs(i)} \sharp^{out}_{\sim}(Outputs(i)) \leq 1$.

4.4. Completeness of messages

The syntax requirement on 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. Together the syntax requirements express that there is *exactly one* way to connect message outputs and message inputs. Consider the following chart.

```
msc example3;                       instance j;
   instance i;                         in m3 from env;
      out m1 to env;                   in m4 from i;
      out m2 to j;                  endinstance;
   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. For the message $m4$ only the message input is specified. One could associate the graphical representation from Figure 3 to 'Basic MSC' $example3$. However, this is not a Basic MSC. The following syntax requirement is formulated.

> *To each message output that is sent to an instance there has to be a corresponding message input specified on that instance. To each message input that is received from an instance there has to be a corresponding message output specified on that instance.*

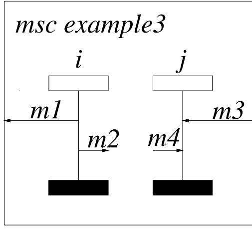

Figure 3. Graphical representation of 'Basic MSC' *example*3

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.

Next, a predicate *CorOut* is defined which determines whether there is a corresponding message input for each message output. The correspondence the other way around, a predicate very similar to *CorOut*, can also be defined. The uniqueness rule for messages guarantees that such a correspondence, if it exists, is unique.

Definition 6 *The predicate* $CorOut : \mathcal{L}(\texttt{<msc>}) \rightarrow I\!B$ *is for all* $ch \in \mathcal{L}(\texttt{<msc>})$ *defined by*

$$CorOut(ch) = \forall_{i,j \sqsubseteq AllInsts(ch)} \; \forall_{o \sqsubseteq Outputs(i)} \\ \Big(Addr(o) = InstName(j) \Rightarrow \exists_{in \sqsubseteq Inputs(j)} \; Message(i)(o) = Message(j)(in)\Big)$$

4.5. Causal dependency of messages

A message is sent before it is consumed. Also with MSC, this convention is followed. This means that it is not allowed that the partial ordering of the communication events specified by the chart states that a message input occurs (in time) before its corresponding message output. Consider for example the charts shown in Figure 4.

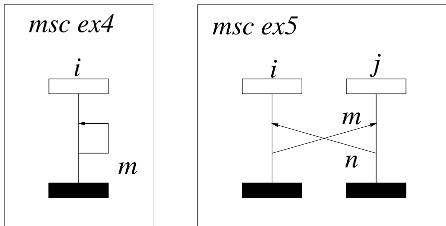

Figure 4. Charts *ex*4 and *ex*5

It is clear that the first chart specifies that the input of message m is executed before the output of message m. For the second chart, 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.

A chart specifies a partial ordering on the set of events being contained. This partial ordering restricted to communication events is described in a minimal form by the *connectivity graph*. We will formalize this notion which is already described in [2]. The nodes of the connectivity graph represent the message output and message input 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 labelled 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. The arrows between these nodes represent the partial ordering of the communication events as specified by the chart.

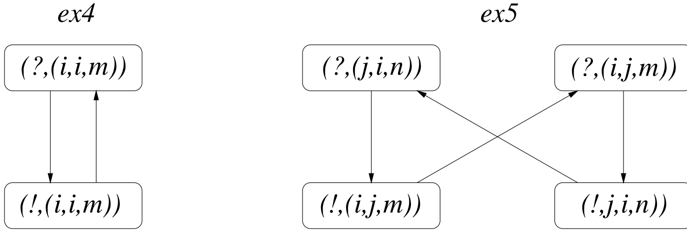

Figure 5. Connectivity graphs for the example charts

In Figure 5 the connectivity graphs of the example charts from Figure 4 are given. In both cases it is clear that it contains a loop. For the labels of the nodes of the connectivity graph the following type is introduced.

Definition 7 *The type MsgLabel is defined as follows:* $MsgLabel = \{!, ?\} \times Msg$.

Next, a function *MsgEvent* is defined which, given an instance, 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.

Definition 8 *Let* $i \in \mathcal{L}(\texttt{<inst def>})$. *The function* $MsgEvent : \mathcal{L}(\texttt{<out>} \mid \texttt{<in>}) \rightarrow MsgLabel$ *is for all* $out \in \mathcal{L}(\texttt{<out>})$ *and* $in \in \mathcal{L}(\texttt{<in>})$ *defined by*

$$\begin{array}{lcl} MsgEvent(i)(out) & = & (!, Message(i)(out)) \\ MsgEvent(i)(in) & = & (?, Message(i)(in)). \end{array}$$

First, the ordering on communication events specified by an instance is computed. This is done by scanning the instance and relating those communication actions which are specified immediately adjoining. The function *Graph* associates to an instance a set of label pairs. Such a pair represents an arrow in the connectivity graph. The first argument of the auxiliary function $Graph(i)$ acts as a 'memory'. It records the communication event immediately preceding the communication event under consideration. We extend

the message labels with an event λ denoting the *empty memory*. This value is supposed to act as an initial memory for *Graph*.

Definition 9 *Let* $i \in \mathcal{L}(\texttt{<inst def>})$ *be an instance definition. The function* $Graph(i) : MsgLabel_\lambda \times \mathcal{L}(\texttt{<inst body>}) \rightarrow \mathbb{P}(MsgLabel \times MsgLabel)$ *is, for all* $l \in MsgLabel_\lambda$, $e \in \mathcal{L}(\texttt{<event>})$ *and* $ib \in \mathcal{L}(\texttt{<inst body>})$, *defined inductively by*

$$\begin{aligned} Graph(i)(l, \texttt{<>}) &= \varnothing \\ Graph(i)(l, e\ ib) &= \begin{cases} Graph(i)(MsgEvent(i)(e), ib) & \text{if } l \equiv \lambda, \\ \{(l, MsgEvent(i)(e)\} \cup Graph(i)(MsgEvent(i)(e), ib) & \text{if } l \not\equiv \lambda. \end{cases} \end{aligned}$$

The function $Graph : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathbb{P}(MsgLabel \times MsgLabel)$ *is for* $i \in \mathcal{L}(\texttt{<inst def>})$ *defined by* $Graph(i) = Graph(i)(\lambda, InstBody(i))$. *The function InstBody (See Appendix A) associates to an instance definition its body.*

Next, the set of pairs of labels is interpreted as a relation on labels. Besides the ordering on communication events specified explicitly by the instances, there is also the ordering between corresponding message outputs and message inputs. The relation $\xrightarrow{ch}$ specifies both the ordering specified by the instances of the chart (as expressed by *Graph*) and the implicit ordering on corresponding message outputs and message inputs.

Definition 10 *Let* $ch \in \mathcal{L}(\texttt{<msc>})$. *The relation* $\xrightarrow{ch} \subseteq MsgLabel \times MsgLabel$ *is the smallest relation such that: for all* t_1, t_2

1) if $(t_1, t_2) \in Graph(i)$ *for some* $i \sqsubseteq AllInsts(ch)$, *then* $t_1 \xrightarrow{ch} t_2$

2) if $Message(i)(out) = Message(j)(in)$ *for some* $i, j \sqsubseteq AllInsts(ch)$, $out \sqsubseteq Outputs(i)$ *and* $in \sqsubseteq Inputs(j)$, *then* $MsgEvent(i)(out) \xrightarrow{ch} MsgEvent(j)(in)$

In terms of the connectivity graph the syntax requirement is formulated as: the connectivity graph does not contain loops or, alternatively, there must not be a path from a node to itself. Next, this formulation is translated in terms of the relation $\xrightarrow{ch}$. With the notions of transitive closure and strictness of relations, as introduced in Section 3, the syntax requirement is formalized as follows: $\xrightarrow{ch}^{+}$ is strict.

5. CONCLUSIONS

We presented a formalization of four syntax requirements of MSC by defining predicates and functions on the textual syntax of MSC. Although we needed many auxiliary functions the formalization presented is straightforward and intuitive. Most of the auxiliary functions introduced were only necessary to obtain the information needed for the verification of a specific requirement from the complete MSC. Also, in formalizing the syntax requirements of the language MSC they are reused.

The treatment of the formalization of the syntax requirements in this paper illustrates the treatment of the syntax requirements of the complete language. As a result of the formalization of syntax requirements there have been discussions on these requirements. In several cases this has led to changes in the requirements.

Besides the discussions on the syntax requirements the chosen approach also enables rapid prototyping. The generating of prototype tools implementing the syntax requirements is shown to be feasible in [5].

REFERENCES

1. Z.100 (1993). CCITT Specification and description language (SDL). ITU-T Jun. 1994.
2. Z.120 (1993). Message Sequence Chart (MSC). ITU-T Sep. 1994.
3. Z.120 B (1995). Message Sequence Chart algebraic semantics. ITU-T Publ. sched.: May 1995.
4. S. Mauw and M.A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.
5. S. Mauw and E. van der Meulen. Generating Tools for Message Sequence Charts. Technical Report TD60, ITU-TS SG10 Interims Meeting, Geneva, Switzerland, 1994.
6. M.A. Reniers. Syntax Requirements of Message Sequence Charts. Technical Report TD59, ITU-TS SG10 Interims Meeting, Geneva, Switzerland, 1994.

A. AUXILIARY FUNCTIONS AND PREDICATES

In this appendix the definitions of the following functions and predicates are given: *Addr*, *AllInsts*, *Inputs*, *InstBody*, *InstName*, *MsgId*, *Outputs*. The definitions are presented in alphabetic order.

The function $Addr : \mathcal{L}(\texttt{<in>|<out>}) \rightarrow \mathcal{L}(\texttt{<address>})$ is for all $msgid \in \mathcal{L}(\texttt{<msgid>})$ and $address \in \mathcal{L}(\texttt{<address>})$ defined by

$$\begin{aligned} Addr(\texttt{out}\ msgid\ \texttt{to}\ address;) &= address \\ Addr(\texttt{in}\ msgid\ \texttt{from}\ address;) &= address \end{aligned}$$

The function $AllInsts : \mathcal{L}(\texttt{<msc body>}) \rightarrow I\!M(\mathcal{L}(\texttt{<inst def>}))$ is for all $i \in \mathcal{L}(\texttt{<inst def>})$ and $mscbody \in \mathcal{L}(\texttt{<msc body>})$ defined by

$$\begin{aligned} AllInsts(\texttt{<>}) &= \boxplus \\ AllInsts(i\ mscbody) &= [i] \sqcup AllInsts(mscbody) \end{aligned}$$

The function $AllInsts : \mathcal{L}(\texttt{<msc>}) \rightarrow I\!M(\mathcal{L}(\texttt{<inst def>}))$ is for $mscname \in \mathcal{L}(\texttt{<msc name>})$ and $mscbody \in \mathcal{L}(\texttt{<msc body>})$ defined by

$$AllInsts(\texttt{msc}\ mscname;\ mscbody\ \texttt{endmsc};) = AllInsts(mscbody)$$

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

$$\begin{aligned} Inputs(\texttt{<>}) &= \boxplus \\ Inputs(e\ ib) &= \begin{cases} Inputs(ib) & \text{if } e \in \mathcal{L}(\texttt{<out>}) \\ [e] \sqcup Inputs(ib) & \text{if } e \in \mathcal{L}(\texttt{<in>}) \end{cases} \end{aligned}$$

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

$$Inputs(i) = Inputs(InstBody(i))$$

The function $Inputs : \mathcal{L}(\texttt{<msc body>}) \rightarrow \mathbb{M}(\mathcal{L}(\texttt{<in>}))$ is for all $i \in \mathcal{L}(\texttt{<inst def>})$ and $mscbody \in \mathcal{L}(\texttt{<msc body>})$ defined by

$$\begin{array}{lcl} Inputs(\texttt{<>}) & = & \boxplus \\ Inputs(i \ mscbody) & = & Inputs(i) \sqcup Inputs(mscbody) \end{array}$$

The function $Inputs : \mathcal{L}(\texttt{<msc>}) \rightarrow \mathbb{M}(\mathcal{L}(\texttt{<in>}))$ is for all $mscname \in \mathcal{L}(\texttt{<msc name>})$ and $mscbody \in \mathcal{L}(\texttt{<msc body>})$ defined by

$$Inputs(\texttt{msc}\ mscname;\ mscbody\ \texttt{endmsc;}) = Inputs(mscbody)$$

The function $InstBody : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathcal{L}(\texttt{<inst body>})$ is for $iname \in \mathcal{L}(\texttt{<inst name>})$ and $instbody \in \mathcal{L}(\texttt{<inst body>})$ defined by

$$InstBody(\texttt{instance}\ iname;\ instbody\ \texttt{endinstance;}) = instbody$$

The function $InstName : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathcal{L}(\texttt{<inst name>})$ is for $iname \in \mathcal{L}(\texttt{<inst name>})$ and $instbody \in \mathcal{L}(\texttt{<inst body>})$ defined by

$$InstName(\texttt{instance}\ iname;\ instbody\ \texttt{endinstance;}) = iname$$

The function $MsgId : \mathcal{L}(\texttt{<in>|<out>}) \rightarrow \mathcal{L}(\texttt{<msgid>})$ is for all $msgid \in \mathcal{L}(\texttt{<msgid>})$ and $address \in \mathcal{L}(\texttt{<address>})$ defined by

$$\begin{array}{lcl} MsgId(\texttt{out}\ msgid\ \texttt{to}\ address;) & = & msgid \\ MsgId(\texttt{in}\ msgid\ \texttt{from}\ address;) & = & msgid \end{array}$$

The function $Outputs : \mathcal{L}(\texttt{<inst body>}) \rightarrow \mathbb{M}(\mathcal{L}(\texttt{<out>}))$ is for all $e \in \mathcal{L}(\texttt{<event>})$ and $ib \in \mathcal{L}(\texttt{<inst body>})$ defined by

$$\begin{array}{lcl} Outputs(\texttt{<>}) & = & \boxplus \\ Outputs(e \ ib) & = & \begin{cases} Outputs(ib) & \text{if } e \in \mathcal{L}(\texttt{<in>}) \\ [e] \sqcup Outputs(ib) & \text{if } e \in \mathcal{L}(\texttt{<out>}) \end{cases} \end{array}$$

The function $Outputs : \mathcal{L}(\texttt{<inst def>}) \rightarrow \mathbb{M}(\mathcal{L}(\texttt{<out>}))$ is for all $i \in \mathcal{L}(\texttt{<inst def>})$ defined by

$$Outputs(i) = Outputs(InstBody(i))$$

The function $Outputs : \mathcal{L}(\texttt{<msc body>}) \rightarrow \mathbb{M}(\mathcal{L}(\texttt{<out>}))$ is for all $i \in \mathcal{L}(\texttt{<inst def>})$ and $mscbody \in \mathcal{L}(\texttt{<msc body>})$ defined by

$$\begin{array}{lcl} Outputs(\texttt{<>}) & = & \boxplus \\ Outputs(i \ mscbody) & = & Outputs(i) \sqcup Outputs(mscbody) \end{array}$$

The function $Outputs : \mathcal{L}(\texttt{<msc>}) \rightarrow \mathbb{M}(\mathcal{L}(\texttt{<out>}))$ is for all $mscname \in \mathcal{L}(\texttt{<msc name>})$ and $mscbody \in \mathcal{L}(\texttt{<msc body>})$ defined by

$$Outputs(\texttt{msc}\ mscname;\ mscbody\ \texttt{endmsc;}) = Outputs(mscbody)$$