The formalization of Message Sequence Charts

S. Mauw[a]

[a] Dept. of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: `sjouke@win.tue.nl`

We discuss the state of affairs with respect to the formalization of Message Sequence Charts (MSC) and identify which parts of the definition of MSC are still candidate for formalization. Further, we give a tutorial on the formal semantics of MSC. The semantics of all features from the MSC language is treated by presenting small examples and the corresponding process algebra expressions.

Keywords: Message Sequence Charts; semantics; process algebra.

1. INTRODUCTION

Message Sequence Charts (MSC) is a graphical language for the specification of system traces. It is standardized by the ITU (International Telecommunication Union). The ITU maintains recommendation Z.120 [5], which contains a (partly informal) definition of MSC. MSCs are applied in the telecommunication sector for the specification of system requirements and for testing, often in combination with SDL [1, 4].

Because of its widespread use, discussions on the meaning of an MSC became unavoidable, so it was decided to start the formalization of MSC several years ago. The first part subject to formalization was the semantics. Several approaches have been studied (see [2, 3, 8, 13]) of which the process algebra approach was accepted for standardization [6, 9]. One of the purposes of this paper is to give an informal tutorial on this process algebra semantics.

Currently, the static requirements are formalized [10]. It is investigated whether other parts of Z.120 also need formalization. The second goal of this paper is to describe the state of affairs with respect to the formalization of MSC and to indicate which parts possibly need further formalization.

The main motivation for formalization is that users of the language need to understand each other precisely. An MSC should have only one interpretation. Ambiguities, inconsistencies and obscurities hamper proper use of the language. This implies not only that the semantics of MSC must have a formal base, but that also the appearance and use of MSCs should be formalized as much as possible. If two computer tools for MSC do not agree on the collection of admitted MSCs, they are not able to exchange MSCs. If there are two incompatible ways to transform an MSC from graphical representation into textual representation, this will lead to problems.

This paper is organized as follows. In Section 2 we discuss the state of affairs with respect to the formalization of MSC. In Section 3 we give a tutorial on the formal semantics. This paper does not contain an introduction to the MSC language. Please refer to the aforementioned literature for more information about MSC.

2. FORMALIZATION

In this section we will explore which parts of the MSC landscape are suited for formalization.

First we will summarize the current situation. Recommendation Z.120 [5] contains the definition of MSC, as accepted by the ITU. It contains a textual and graphical syntax and an informal description of the meaning of an MSC. Appendix B to recommendation Z.120 [6] contains a formal description of the semantics of MSC using process algebra. Finally, a proposal for a formal definition of the static requirements for MSC is in [10, 15].

Figure 1 displays the relevant parts of the MSC definition with respect to formalization. We consider the textual syntax, the graphical syntax and the semantics.
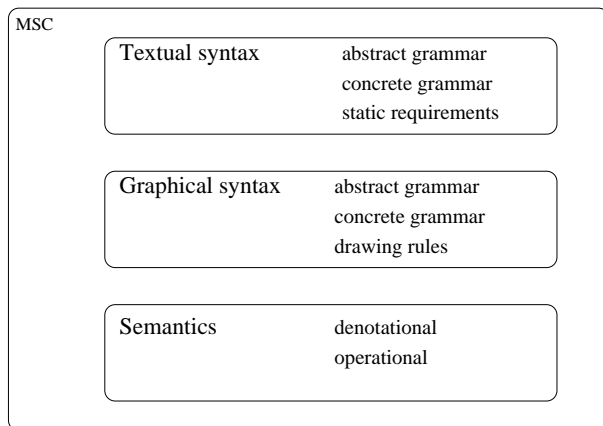

Figure 1. Parts of Z.120 possibly subject to formalization

2.1. Textual syntax

Although MSC is basically a graphical language, a textual syntax is defined for manipulation of MSCs, such as exchanging them between users or tools.

The abstract grammar describes the bare information contents of a textual MSC. Its main purpose is to provide a clean starting point for the definition of the semantics (although the formal semantics is based on the concrete grammar for ease of use). It can also be used for transformation and manipulation of the contents of an MSC (such as analysis), in contrast with manipulation of the occurrence of an MSC (such as pretty printing).

The concrete grammar describes the appearance of an MSC to a user. It defines the actual tokens (lexical syntax) which make up an MSC text and the order in which they

are allowed to appear (context free syntax). The abstract grammar is more or less equal to the concrete textual grammar from which all syntactic sugar is removed.

Both abstract and concrete grammar are given using a so-called BNF grammar, which is an accepted formal technique for defining the syntax of a language.

Not all MSCs generated by the grammar are considered *valid*. For instance, it is required that every instance referenced in an MSC has been declared. This gives raise to a set of requirements on MSC. These are often called *static semantics*, but we prefer *static requirements* or *context sensitive syntax*.[1]

The definition of the *static requirements* in Z.120 is in natural language and thus informal. Formalizing the requirements [10, 15] not only helped in removing ambiguities and making Z.120 more consistent, it also revealed some deficiencies.

We may conclude that the textual syntax will soon be formalized completely.

2.2. Graphical syntax

The graphical syntax from Z.120 is defined by BNF-like production rules. Informal texts are used to define the relation between the graphical constructs. Phrases such as "is connected to" and "is followed by" leave room for misinterpretations. For instance, a `<message in area>` consists a.o. of a `<message symbol>` (i.e. a line with an arrow pointing left or right) `connected to` a `<message out area>` (i.e. a horizontal line). One possible interpretation is that it is allowed to connect the horizontal line to the arrow head of the arrow-line. This is obviously not the intention.

A more serious problem is that the meaning of the set of rules is not clear. Whereas in the textual grammar the rules are viewed as productions (only expressions are considered which can be produced by applying the rules a finite number of times), in the graphical grammar the rules are also used to define the (spatial) relation with already produced graphical objects. For instance, an `<msc symbol>` is the bounding box of an MSC. It can contain several items, for example an instance with a message from the environment. This incoming message symbol is connected to the bounding box. This bounding box is simply repeated in the production rule for an incoming message. If we consider this a real production rule, we obtain more than one bounding box. Obviously, we must interpret the rules in such a way that this bounding box is the same one as the one produced earlier.

It is hard to define a two-dimensional structure using a set of linear production rules. A more natural way of dealing with this problem is to use, so-called, graph grammars. A promising approach is given in [14]. It defines both an abstract and concrete graphical syntax for Basic Message Sequence Charts.

The graphical grammar in Z.120 is accompanied by a series of informal *drawing rules*, which partly restrict the number of allowed MSCs and partly help to interpret MSCs whose meaning is not obvious. Whenever possible the restrictions were already covered in the static requirements for the textual syntax.

Our conclusion is that there is a need for a more formal treatment of the graphical syntax and the drawing rules of MSC.

---

[1]The term *static semantics* is somewhat overloaded. The term is also used for the semantics of static data types and for the preprocessing phase in which a language text is transformed into some normal form for which the *dynamic semantics* can be obtained easily (e.g. removing modular structure and expanding macro definitions).

2.3. Semantics

With respect to semantics of programming languages a distinction can be made between denotational semantics and operational semantics.

A denotational semantics consists of a translation of an expression from the language into an expression in some mathematical domain. This allows for formal manipulation (in case of MSC, e.g. expanding decomposed instances by sub-MSCs) and deriving properties (e.g. check if two MSCs are in fact identical up to some isomorphism, how many traces does the MSC generate, can the MSC be implemented synchronously).

A more dynamic interpretation of a language can be obtained by means of an operational semantics. It consists of a procedure to transform an expression into a behaviour (an execution step) and a new expression (the result after executing this step). Using the operational semantics one can prototype or test an expression.

The semantics of MSC is described in Z.120 [5] in an informal and often operational way. The formal semantics is given in [6] by means of a translation into process algebra expressions. This is a denotational semantics. However, one can easily assign an operational semantics to these process algebra expressions, by adding transition rules. This is elaborated for Basic Message Sequence Charts in [9]. It is not difficult to extend the formal semantics of the complete MSC language with operational rules. Instead of giving an indirect operational semantics by defining the operational behaviour of the process algebra expressions, it is also possible to give an operational semantics directly based on MSC. The main advantages of this one step approach are comprehensibility and ease of use. The disadvantage is that it is more difficult to establish a formal relation between the operational and denotational semantics in this way.

The current semantics does not define the composition and decomposition of MSCs. The inclusion of explicit mechanisms for structuring a specification are still under discussion [7].

We conclude that with respect to the semantics, work is needed on defining the operational semantics and defining the composition of MSCs.

2.4. Other issues

In this section we will discuss some issues concerning formalization that are not strictly related to one of the items from Figure 1.

First we will look at the relation between the textual syntax, the graphical syntax and the semantics. Basically, the textual syntax and the graphical syntax define the same language, so, ideally the abstract textual grammar and the abstract graphical grammar are more or less identical. However, in MSC there is a discrepancy between the textual approach and the graphical approach. The textual representation is *instance oriented*, whereas the graphical representation is *event oriented*. This means that the textual representation defines all events, ordered per instance. This makes it non-trivial to identify which message output corresponds to which message input. Extension of the message name is needed to solve this problem. In the graphical representation message inputs and message output are always connected, so the problem does not occur here.

Since the translation of both syntaxes into each other is non-trivial, the procedure for doing so should be defined explicitly and formally. Such conversion is also needed, since the formal semantics is only defined for the textual representation of MSC.

Next, we have a look at tools. The formal definition of the grammars can be viewed as a high level definition of tools for scanning, parsing, requirements checking, editing and drawing of MSC texts and graphs.

Tools for analyzing MSCs will be based on the denotational semantics and tools for simulation and prototyping will be based on the operational semantics.

In order to formalize tools which aid in designing SDL specifications from MSC diagrams, the relation between the two languages has to be studied and formalized. In exchanging MSCs between tools one often wants to include information that is not represented in the semantics, such as concrete positions of objects in the graphical representation. A (formalized) interchange language will be useful for this.

In [12] a collection of tools for Basic Message Sequence Charts was formally specified. Although it is possible to formalize all tools and uses of the MSC language, it is, in our opinion, not desirable to restrict users and tool builders too much.

Summarizing, a formal procedure for transforming textual and graphical syntax into each other is needed and the relation between MSC and related languages such as SDL and TTCN has to be formalized.

3. FORMAL SEMANTICS

The purpose of this section is to give an overview of the process algebra semantics of MSC, as defined in [6]. Process algebra theory will be avoided as much as possible. We present a series of MSC examples and explain for each example the process algebra expression that denotes its semantics. Most MSC features will be treated in such a way that the semantics of arbitrary MSCs can be derived simply by applying the rules of thumb. Although the formal semantics is defined on the textual syntax of MSC, we will only display the MSCs in the graphical representation.

3.1. One instance
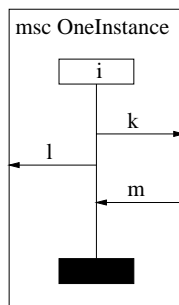First we look at an MSC with one single instance (see Figure 2).

Figure 2. An MSC with one instance

MSC *OneInstance* describes instance i, which has three communications with the environment. An MSC is completely characterized by the sequences of events it allows.

Obviously, MSC *OneInstance* describes only one unique trace: first we have an output of message k to the environment, after that there is an output of message l to the environment, and finally an input of message m from the environment. In process algebra notation, this gives the following trace:

$$out(i, env, k) \cdot out(i, env, l) \cdot in(env, i, m)$$

The operator $\cdot$ denotes strict sequential composition. So, the semantics of an MSC is a process algebra expression, which is built from operators (e.g. the sequential composition) and actions or simple events (i.e. events such as the sending of message k from instance i to the environment, denoted by $out(i, env, k)$). Since the events on one instance are ordered from top to bottom, the semantics of an instance is the sequential composition of its events.

3.2. Two instances

The semantics of an MSC is constructed from the semantics of its instances. In Figure 3 we have two instances exchanging one message.
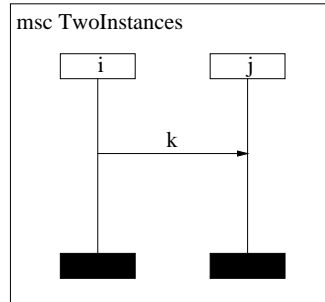

Figure 3. An MSC with two instances

The semantics of instance i is $out(i, j, k)$. The semantics of instance j is $in(i, j, k)$. The basic idea is that the two instances operate in parallel, independently of each other. So, as a first attempt, the semantics of MSC $TwoInstances$ is the parallel composition of the semantics of instances i and j. In process algebra notation:

$$out(i, j, k) \parallel in(i, j, k)$$

The operator $\parallel$ denotes parallel composition. It is defined as the interleaved execution of its operands. It means that we have the following traces: $out(i, j, k) \cdot in(i, j, k)$ and $in(i, j, k) \cdot out(i, j, k)$. In process algebra notation:

$$out(i, j, k) \cdot in(i, j, k) + in(i, j, k) \cdot out(i, j, k)$$

Here we use the operator $+$ to denote alternatives. This expression is the result of expanding the $\parallel$ operator from the previous expression. The precedence of the operators

is as follows: $\cdot$ has precedence over $\parallel$, which has precedence over $+$. So, the expression $a + b \cdot c \parallel d$ should be read as $a + ((b \cdot c) \parallel d)$.

Now, notice that our interpretation that the semantics of an MSC is the parallel composition of the semantics of its instances does not take into account the fact that a message can only be received after that it has been sent. So, in the above expression we allow too many traces. The trace $in(i, j, k) \cdot out(i, j, k)$ must be ruled out. We introduce the operator λ to enforce the basic static requirement that a message must be sent before it is received. This operator is often called the *state operator*.

The desired semantical expression is

$$\lambda(out(i, j, k) \parallel in(i, j, k))$$

which is equal to

$$\lambda(out(i, j, k) \cdot in(i, j, k) + in(i, j, k) \cdot out(i, j, k))$$

which is simply

$$out(i, j, k) \cdot in(i, j, k)$$

So, the semantics of an MSC is derived from the semantics of its instances by placing them in parallel and removing unwanted execution orders by applying the state operator. Please note, that applying the state operator to the MSC *ONeInstance* in Figure 2 does not change the result.

3.3. Two messages

The following example describes two instances exchanging two messages (see Figure 4).
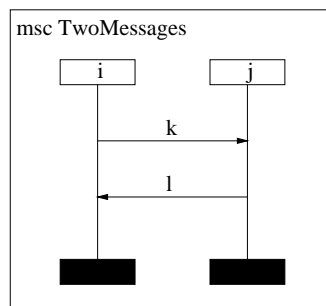

Figure 4. An MSC with two messages

The semantics of instance i is $out(i, j, k) \cdot in(j, i, l)$. The semantics of instance j is $in(i, j, k) \cdot out(j, i, l)$. The semantics of MSC $TwoMessages$ is as follows

$$\lambda(out(i, j, k) \cdot in(j, i, l) \parallel in(i, j, k) \cdot out(j, i, l))$$

This can be expanded as

$$\lambda(out(i,j,k)\cdot(\quad in(i,j,k)\cdot(in(j,i,l)\cdot out(j,i,l)+out(j,i,l)\cdot in(j,i,l))$$
$$+ \;\; in(j,i,l)\cdot in(i,j,k)\cdot out(j,i,l)$$
$$)$$
$$+$$
$$in(i,j,k)\cdot (\;\; out(i,j,k)\cdot(out(j,i,l)\cdot in(j,i,l)+in(j,i,l)\cdot out(j,i,l))$$
$$+ \;\; out(j,i,l)\cdot out(i,j,k)\cdot in(j,i,l)$$
$$)$$
$$)$$

The operand of the λ operator can be read as follows. Either we start with sending message k or receiving k. In the first case, there are two ways to proceed. The first possibility is that i receives message k. After that, sending and receiving of l can occur in any order. The second possibility after sending k is that instance j receives message l. After that, instance j first receives k and then sends l. The case that the process starts with receiving k is analogous.

It is clear that the operand of the λ operator defines several traces in which a message is received before it has been sent, e.g. $out(i,j,k)\cdot in(j,i,l)\cdot in(i,j,k)\cdot out(j,i,l)$. This is again the reason for applying the λ operator. After removing all unwanted execution orders, the semantics equals

$$out(i,j,k)\cdot in(i,j,k)\cdot out(j,i,l)\cdot in(j,i,l)$$

3.4. Two independent messages

Next, we give an example in which the messages are not causally related (see Figure 5).
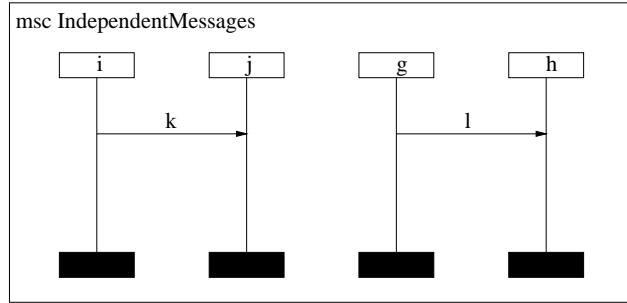

Figure 5. An MSC with two independent messages

The same technique as in the previous example works. First we determine the semantics of the instances i, j, g and h, which are $out(i,j,k)$, $in(i,j,k)$, $out(g,h,l)$ and $in(g,h,l)$, respectively.

Then, the semantics of MSC $IndependentMessages$ is as follows

$$\lambda(out(i,j,k) \parallel in(i,j,k) \parallel out(g,h,l) \parallel in(g,h,l))$$

After expanding the parallel composition and filtering out unwanted execution sequences, this gives

$$out(i,j,k) \cdot (\quad out(g,h,l) \cdot (in(i,j,k) \cdot in(g,h,l) + in(g,h,l) \cdot in(i,j,k))$$
$$+ \; in(i,j,k) \cdot out(g,h,l) \cdot in(g,h,l)$$
$$)$$
$$+$$
$$out(g,h,l) \cdot (\; out(i,j,k) \cdot (in(i,j,k) \cdot in(g,h,l) + in(g,h,l) \cdot in(i,j,k))$$
$$+ \; in(g,h,l) \cdot out(i,j,k) \cdot in(i,j,k)$$
$$)$$

3.5. Conditions

A condition marks a point in an MSC which is of special interest. As has been concluded in [7], conditions are used for many purposes, e.g. for composition of MSCs and for describing a particular state. Some problematic issues concerning the meaning of conditions were discussed in [11]. Due to this variety of uses and the wide range of interpretations, conditions are considered as meaningless in the formal semantics. This is expressed by defining the semantics of a condition as ε, which is the process that displays no activity. The semantics of the MSC in Figure 6 is equal to

$$in(env,i,k) \cdot \varepsilon \cdot out(i,env,l)$$

which is simply equal to
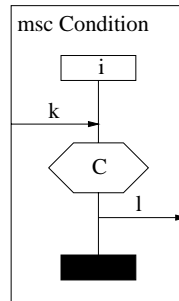
$$in(env,i,k) \cdot out(i,env,l)$$

Figure 6. An MSC with a condition

3.6. Timers

There are three different activities involved with timers. The setting of timer t on instance i is denoted by $set(i,t)$, a timeout is denoted by $timeout(i,t)$ and a reset by $reset(i,t)$. In the semantics they play the role of simple events.

Figure 7 shows a timer t which is set by instance i. Before the timer expires, instance i sends a message to instance j.
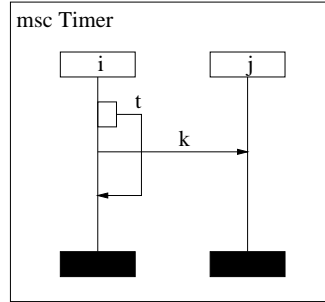
Figure 7. An MSC with a timer

The semantics of instance i is $set(i,t) \cdot out(i,j,k) \cdot timeout(i,t)$. The semantics of instance j is $in(i,j,k)$. The semantics of MSC *Timer* is

$$\lambda(set(i,t) \cdot out(i,j,k) \cdot timeout(i,t) \parallel in(i,j,k))$$

which evaluates to

$$set(i,t) \cdot out(i,j,k) \cdot (in(i,j,k) \cdot timeout(i,t) + timeout(i,t) \cdot in(i,j,k))$$

3.7. Actions

An action is also considered to be a simple event. If instance i performs action a, this is denoted by the event $action(i,a)$. The semantics of the MSC in Figure 8 is simply

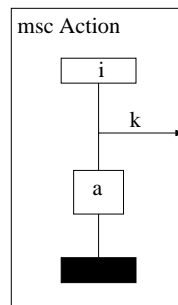$$out(i,env,k) \cdot action(i,a)$$

Figure 8. An MSC with an action

3.8. Instance creation and stop

Creation of an instance is treated as an asynchronous message. The event $create(i,j)$ denotes that instance i creates instance j and the event $start(j)$ denotes that instance
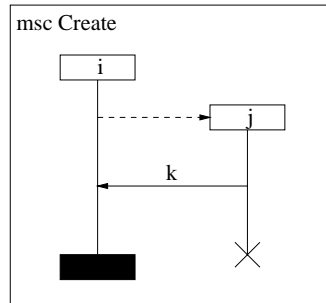
Figure 9. An MSC with creation and stop

j is a newly created instance starting its operation. Stopping instance j is denoted by $stop(j)$.

The semantics of the MSC in Figure 9 is

$$create(i,j) \cdot start(j) \cdot out(j,i,k) \cdot (in(j,i,k) \cdot stop(j) + stop(j) \cdot in(j,i,k))$$

3.9. Coregion

A coregion is used to relax the strict ordering of events along an instance axis. As stated before, the strict ordering is reflected in the semantics by using the operator for sequential composition ($\cdot$). If we use the operator for parallel composition ($\parallel$) instead, the ordering is completely free.
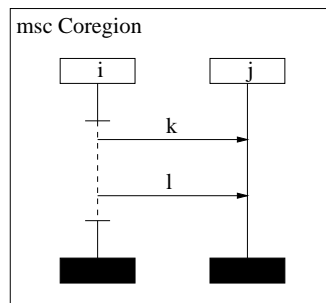

Figure 10. An MSC with a coregion

The semantics of instance j in Figure 10 is again $in(i,j,k) \cdot in(i,j,l)$. However, the semantics of instance i is $out(i,j,k) \parallel out(i,j,l)$. This is equal to $out(i,j,k) \cdot out(i,j,l) + out(i,j,l) \cdot out(i,j,k)$. Now, the semantics of MSC $Coregion$ is

$$\lambda(in(i,j,k) \cdot in(i,j,l) \parallel (out(i,j,k) \cdot out(i,j,l) + out(i,j,l) \cdot out(i,j,k))$$

This is equal to

$$out(i, j, k) \cdot (\quad out(i, j, l) \cdot in(i, j, k) \cdot in(i, j, l)$$
$$+ \ in(i, j, k) \cdot out(i, j, l) \cdot in(i, j, l)$$
$$)$$
$$+$$
$$out(i, j, l) \cdot out(i, j, k) \cdot in(i, j, k) \cdot in(i, j, l)$$

3.10. Sub-MSC

A modular design is obtained by using sub-MSCs and decomposed instances. An MSC with decomposed instances can be transformed into one without. This is done by simply replacing the behaviour of the decomposed instances by the behaviour of the corresponding sub-MSCs. Technically this is a complex operation, but the intuition behind it is quite clear.

Consider MSC *Decomp* in Figure 11. It has two normal instances, i and j, and one decomposed instance d. The decomposed instance refers to submsc d, which defines instances g and h.
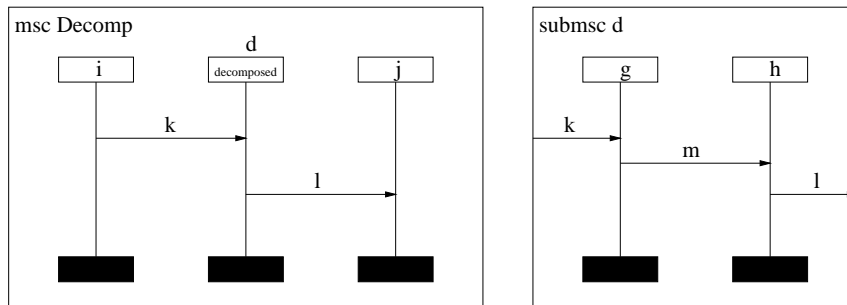

Figure 11. An MSC with decomposition

The resulting MSC (see Figure 12) now contains instances i, j, g and h. Notice that message k from i to d is repeated as a message k in submsc d from the environment to instance g. This means that in the resulting MSC there is one single message k from instance i to instance g. The same reasoning goes for message l.

The semantics of MSC *Decomp* in Figure 11 is thus equal to the semantics of MSC *Flat* in Figure 12, which is

$$out(i, g, k) \cdot in(i, g, k) \cdot out(g, h, m) \cdot in(g, h, m) \cdot out(h, j, l) \cdot in(h, j, l)$$

4. STATIC REQUIREMENTS

This section contains a short explanation of the static requirements for MSC, as formalized in [10].
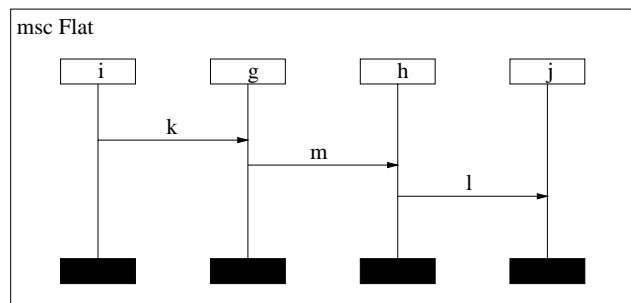
Figure 12. Result of flattening an MSC with decomposition

4.1. Requirements

Z.120 contains an informal description of a number of static requirements for MSC. The requirements treated in [10] either follow from recommendation Z.120 or from the discussion within the ITU study group maintaining this recommendation.

The requirements are stated in terms of predicates and functions on the textual syntax of 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.

Every requirement is expressed in one predicate. An MSC is *valid* if the conjunction of all these predicates yields *true*. Requirements are defined with respect to the following:

- Uniqueness of instances;
- Consistency of chart interfaces;
- Declaration of referenced instances;
- Unique correspondence between input and output messages;
- Completeness of condition definitions;
- Uniqueness of process creation;
- Correct typing of creating, created and stopped instances;
- Consistency of causal orderings;
- Consistency of condition order;
- Messages crossing conditions
- Consistency of decomposition

We will not treat all predicates. As an example we will explain a predicate that is used for checking the unique correspondence between input and output messages.

4.2. Correspondence between message input and output

The requirement which we explain here is informally stated as follows.

There is a unique correspondence between message inputs and message outputs.

This requirement is too complex to define in one step. After refining the statement, we obtain several predicates, one of which is the following.

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

Figure 13 contains an illegal MSC. It does not satisfy this requirement. Message k from instance i has a corresponding input on instance j, but message l has not.
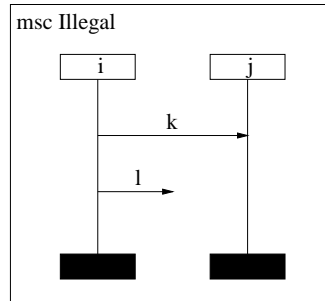

Figure 13. An illegal MSC

The textual representation of this illegal MSC is as follows.

```
msc Illegal;
   instance i;
      out k to j;
      out l to j;
   endinstance;
   instance j;
      in k from i;
   endinstance;
endmsc;
```

When formalizing the above stated requirement it is immediately clear that we need a function that determines the set of all outputs in a given MSC. This function is defined in two steps. First we determine the set of all instance definitions from a given MSC, say ch. This is calculated by the function $AllInsts(ch)$, which can be easily defined. Notice that this gives all instance *definitions*. The names of the instances can be determined by applying the function $Instname(i)$ to a given instance definition i.

Next, we define the function which, given an instance definition i, determines the set of outputs occurring in this instance definition (notation $Outputs(i)$). In the same way we can define the function $Inputs(i)$.

For the given example, we have

$AllInsts(Illegal) =$

```
    { instance i; out k to j; out l to j; endinstance;,
      instance j; in k from i; endinstance; }
```
$Outputs($instance i; out k to j; out l to j; endinstance;$) =$
```
    { out k to j;, out l to j; }
```
$Inputs($instance j; in k from i; endinstance;$) =$
```
    { in k from i; }
```
$Instname($instance j; in k from i; endinstance;$) = $ j

Now, notice that the information from the function $Outputs$ is not complete. The sender of the message is not represented in out k to j;. Therefore, we need a more complete and abstract representation of a message: (i, j, k). This means that instance i sends message k to instance j. We define the function $Message(i)(o)$ which, given an instance definition i and output o determines the abstract message (i, j, k). Using this function, we can check easily if a given input and a given output correspond. Simply check if they refer to the same abstract message.

In the example, we have

$Message($instance i; out k to j; out l to j; endinstance;$)($out k to j;$) =$
  (i, j, k)
$Message($instance j; in k from i; endinstance;$)($in k from i;$) =$
  (i, j, k)

So, for the output of message k there is a corresponding input on instance j.

Now, we have all ingredients for formally defining the requirement. We rephrase it as follows.

> *Let ch be an MSC, then for every instance i defined in ch, and for every output o defined on i find instance definition j that bears the name of the addressee of output o. Now j must have an input in that refers to the same abstract message as o.*

Stated formally (using the function $Addr(o)$ to find the addressee of an output message o):

$$\forall_{i \in AllInsts(ch)} \forall_{o \in Outputs(i)} \forall_{j \in AllInsts(ch)}$$
$$\left(Addr(o) = InstName(j) \Rightarrow \exists_{in \in Inputs(j)} Message(i)(o) = Message(j)(in) \right)$$

The example shows that for the output of message k there is a corresponding input on instance j, but for the output of message l there is not. Therefore, the example does not satisfy the requirement.

5. CONCLUSION

We conclude that, although much work with respect to formalization has already been done, there are still some issues that need clarification. We mentioned graphical syntax, drawing rules, the relation between graphical and textual syntax, operational semantics, composition of MSCs and the relation between MSC and other languages, such as SDL and TTCN.

With respect to the formal semantics of MSC, we conclude that although knowledge of process algebra is indispensable for a complete understanding, it is very easy to express *the* meaning of an MSC in a process algebra expression without having detailed knowledge of the underlying theory.

REFERENCES

1. F. Belina, D. Hogrefe, and A. Sarma. *SDL with applications from protocol specification*. Prentice Hall, 1991.

2. J. de Man. Towards a formal semantics of Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 Using Objects*, Proceedings of the Sixth SDL Forum, Darmstadt, 1993. Elsevier Science Publishers B.V.

3. J. Grabowski, P. Graubmann, and E. Rudolph. Towards a Petri Net based semantics definition for Message Sequence Charts. In O. Færgemand and A. Sarma, editors, *SDL'93 Using Objects*, Proceedings of the Sixth SDL Forum, Darmstadt, 1993. Elsevier Science Publishers B.V.

4. ITU-TS. *ITU-TS Recommendation Z.100: Specification and Description Language (SDL)*. ITU-TS, Geneva, 1988.

5. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1994.

6. ITU-TS. *ITU-TS Recommendation Z.120 Annex B: Algebraic semantics of Message Sequence Charts*. ITU-TS, Geneva, 1995.

7. ITU-TS. Minutes of MSC internet meeting on semantics of conditions. MSC internet meeting TD11, 1995.

8. P.B. Ladkin and S. Leue. Interpreting Message Sequence Charts. Technical Report IBM RJ 8965, IBM Almaden Research Center, San Jose, CA, 1992.

9. S. Mauw and M.A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The computer journal*, 37(4):269–277, 1994.

10. S. Mauw and M.A. Reniers. Formalization of static requirements for Message Sequence Charts. Joint rapporteurs meeting SG10, Geneva TD9010, ITU-TS, 1994.

11. S. Mauw and M.A. Reniers. Thoughts on the meaning of conditions. Experts meeting SG10, St. Petersburg TD9016, ITU-TS, 1995.

12. S. Mauw and E.A. van der Meulen. Generating tools for Message Sequence Charts. In *SDL'95 with MSC in CASE*, Proceedings of the Seventh SDL Forum, Oslo, 1995. Elsevier Science Publishers B.V.

13. S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous Interworkings. In O. Færgemand and A. Sarma, editors, *SDL'93 Using Objects*, Proceedings of the Sixth SDL Forum, Darmstadt, 1993. Elsevier Science Publishers B.V.

14. J. Rekers. A definition of the graphical syntax of Basic Message Sequence Charts. Technical report, Leiden University. (in preparation).

15. M.A. Reniers. Static semantics of Message Sequence Charts. In *SDL'95 with MSC in CASE*, Proceedings of the Seventh SDL Forum, Oslo, 1995. Elsevier Science Publishers B.V.