AN ALGEBRAIC SEMANTICS FOR MESSAGE SEQUENCE CHART DOCUMENTS

Thomas Gehrke, Michaela Huhn*,
Arend Rensink, Heike Wehrheim°

Universität Hildesheim, Institut für Informatik (Prof. U. Goltz),
Postfach 101363, D–31113 Hildesheim, Germany
*Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz (Prof. D. Schmid),
Postfach 6980, D–76128 Karlsruhe, Germany
°Universität Oldenburg, Fachbereich Informatik, Abt. Semantik (Prof. E-R. Olderog),
Postfach 2503, D–26111 Oldenburg,Germany

gehrke|rensink@informatik.uni-hildesheim.de,
Michaela.Huhn@informatik.uni-karlsruhe.de,
Heike.Wehrheim@informatik.uni-oldenburg.de

Abstract: Message Sequence Charts (MSCs) are a graphical and textual language for the specification of message passing systems, in particular telecommunication systems. MSCs are standardised by the Internal Telecommunication Union in standard Z.120. Included in the standard is a formal semantics for MSCs by means of a process algebra. This semantics covers the complete language of single MSCs but lacks an interpretation for *conditions* which are used as continuation points of MSCs within an MSC document (a collection of MSCs). In this paper, we give a process algebraic semantics for basic MSCs including conditions, enabling the formal interpretation of entire MSC documents.

1 INTRODUCTION

Message Sequence Charts (MSCs) are a widely used formalism for the specification of the communication behaviour of reactive systems. It allows for the graphical and textual representation of the communication structure of systems. MSCs focus on the temporal ordering of interaction among system components by specifying the executable traces of a system. They are for instance used as a graphical representation of executable traces of SDL [17] specifications but also as a specification language in their own right. The language has been standardised in the standard ITU-T Z.120 of the International Telecommunication

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

Union [7]. Communication in MSCs is asynchronous; the synchronous variant are Interworkings [15]. MSC-like diagrams have nowadays also been incorporated into various object-oriented specification technique, like for instance the Unified Modeling Language (UML) [18].

In order to give a precise meaning to MSCs and to allow verification, a formal semantics is needed. In the standard, the semantics of MSCs is given via a translation, originally developed in [13], transforming the textual representation of MSCs into a process algebra, based on [1], for which an axiomatic semantics exists. Other semantics based on Petri-nets or Büchi automata can be found in [11, 10, 6]. However, the standard process algebra semantics does not capture a specific feature of MSCs called *conditions*. Conditions are a rudimentary form of MSC composition: Within an MSC document (a collection of MSC diagrams), a condition describes possible continuation points of system behaviour. Every MSC describes a part of the interaction behaviour of the system, and an MSC ending with a specific condition can be "glued together" with every other MSC starting with this condition. This gives rise to a form of sequential composition, followed by a choice of follow-up MSCs, which is indispensable for the specification of infinite behaviour by means of a set of finite MSCs. The only semantics incorporating this interpretation of conditions is the automata semantics of [11]. The latter, however, presumes finite-stateness of the system under consideration, which in our opinion is not *a priori* fixed, given that MSCs are based on asynchronous communication and, even more important, conditions easily allow the specification of non-regular behaviour.

In this paper, we therefore present an alternative proposal for a process algebra semantics for MSCs, which is capable of handling the composition of MSCs via conditions. Conditions are translated into process names, which are interpreted according to the semantics of the MSCs starting with the condition. The composition operator used for glueing MSCs together is a form of *weak sequential composition* based on [19], which is essentially sequential composition on the level of *instances* (the MSC equivalent of a sequential process). This operator captures precisely the right interplay between sequential composition and the choice of the follow-up MSC.

Apart from communication behaviour, other aspects specifiable in MSCs are:

Local actions: actions going on within a single instance that do not influence and cannot be influenced by the environment. Our treatment coincides with the standard.

Timer actions: the setting of timers and the resulting timeouts. Since we have no timing aspects in our model, and timer actions are local to instances, their formalisation would coincide with that of local actions (see above). For that reason, we ignore timer actions in this paper.

General ordering: explicit orderings of actions of different instances, by unspecified means. We have not attempted to model such arbitrary ordering; in fact, we know of no formal semantics to date. A straightforward formalisation would be to use a special kind of communication for this purpose and hide it (in a process algebra sense) afterwards.

Coregions: segments of a given instance where the ordering of the actions is not fixed but may be arbitrarily interleaved. We model this by non-synchronising parallel composition, coincident with the standard treatment.

Instance creation: the generation of a new instance and its eventual termination. We ignore instance creation in this paper; in the conclusions we briefly discuss how it might be integrated, using a technique inspired by the standard semantics.

Instance decomposition: the replacement, within a given MSC, of a single instance by an entire sub-MSC, with corresponding redirection of messages sent to or received from the refined instance. Because of the consistency requirements involved, as well as the issue of redirection and various other technical questions (not all of which are answered or even addressed in the official standard), decomposition is a very complex matter. We intend to investigate instance decomposition in the future but omit it for now.

More involved structuring mechanisms mentioned in [7], also not modelled here, include *inline expressions*, *MSC references* and *High-level MSCs*. A semantics for the latter has recently been proposed in [14]. High-level MSCs involve the explicit composition of basic MSCs in a flow chart style, as opposed to their implicit composition through conditions. The formalisation of sequential composition in High-level MSCs in [14] is also based on [19], just as our approach, with the difference that we explicitly recognise the localities of the MSC instances, whereas they model them indirectly through *dependencies* – which is closer to the formalisation in [19] and more powerful than our locality-based approach, but for the purpose of formalising MSCs poses unwarranted complications. The combination of the standard basic MSC semantics in [7] and the High-level MSC semantics in [14] gives rise to a framework that is significantly more complicated than the one we present here.

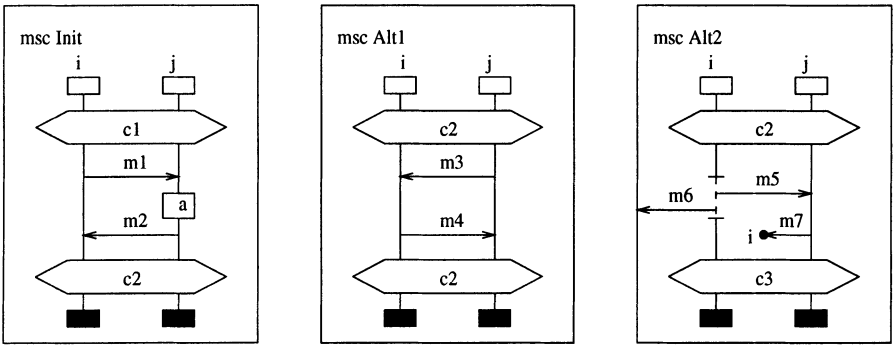
In Section 2, we start with a description of MSC documents. In Section 3, we present our process algebra with its structural operational semantics. Section 4 is concerned with a translation of MSC documents into our process algebra. Section 5 discusses the correspondence of our semantics with the standard one for single basic MSCs according to [13] and shows some algebraic properties of our translation. Finally, Section 6 contains conclusions and discusses the relation of our work with [14] in somewhat more detail.

2 MESSAGE SEQUENCE CHART DOCUMENTS

We start with a brief description of the functionality of MSC documents, as far as it is relevant for our approach. According to the ITU-T standard Z.120 [7], a Message Sequence Chart document consists of a collection of Message Sequence Charts and Message Sequence Chart diagrams (i.e., high-level MSCs). In this paper, we merely consider MSCs. They specify communication/interaction scenarios among a set of *instances* exchanging *messages*. Instances can be seen as

processes residing on different locations. In the graphical charts, the temporal behaviour of one instance is written along a vertical axis. The execution of different instances is assumed to be asynchronous, thus an axis denotes the *local* time of the associated instance running from top to bottom. The behaviour of the external *environment* of a system is modelled as a specific instance in MSCs whose time axis is the frame of the MSC. See for instance the MSC `Alt2` in Figure 1, in which an instance communicates with the environment.

Figure 1: Example: A simple MSC document



The actions an instance may execute are depicted as follows: Communications are denoted by horizontal or diagonal arrows linking the sender of a message to the receiver. The message to be sent is written as a name upon the arrow. For instance, in the MSC `Init` in Figure 1, instances i and j exchange messages $m1$ and $m2$. In case a message is lost, the head of the arrow does not end at the receiver instance, but in a bullet with the name of the intended receiver associated. Vice versa, a message may be spontaneously generated. *Internal* (or *local*) actions are drawn as rectangles containing the name of the action inside; see for instance the action a in the MSC `Init`. In the ITU-T standard Z.120, internal actions are simply called "actions" and describe some activity local to an instance. A further class of actions are *timer* actions; however, as mentioned in the introduction, we follow [13] and (essentially) [7] in regarding these as special cases of internal actions.

Instances are assumed to run sequentially. Thus, the order of action occurrences along the time axis specifies a *total* order of execution for an instance. An exception to this rule are so-called *coregions*, which specify unordered events of an instance. In a coregion, the time axis is depicted as a dashed line. See for instance the sending of $m5$ and $m6$ in `Alt2`.

An important concept for the composition of MSCs are *conditions*. Graphically, conditions are represented as horizontally elongated hexagons, with the name of the condition inside. Conditions can be *shared* between instances; a condition that is shared by all instances of an MSC is called *global*. A global condition is *initial* if it is the first item of all instances, and *final* if it is the last. According to the ITU standard [7], conditions can be used either for informal annotations or for the composition of different MSCs of a document. The latter role is reserved for (global) initial and final conditions: a chart ending with

a specific global condition can be continued by any other chart starting with the same global condition. Thus, conditions determine possible continuations of MSCs. Since the standard semantics does not take MSC composition into account, it treats all kinds of conditions as empty steps. In this paper, we deal with global conditions only; local conditions would still be translated to empty steps.

It is important to note that conditions are *not* intended as a synchronisation device: although they are sometimes referred to as "global states" in the standard, there is *no* requirement that all instances partaking in a condition are simultaneously at that point of their time axis.

Figure 1 shows an MSC document which consists of three MSCs connected by conditions. The MSCs describe the interaction between two instances i and j. Assume that execution is started at the initial MSC `Init`.[1] The final condition $c2$ of `Init` is also the initial condition of the MSCs `Alt1` and `Alt2`; therefore, after performing `Init` either `Alt1` or `Alt2` can be executed. In `Alt1`, the condition $c2$ is also its final condition, which means that this MSC describes a loop.

Events which are not causally related may be executed independently; for example, after composing `Init` and `Alt1` through the condition $c2$, the output event of message $m3$ and the input event of message $m2$ are independent, which means that they may occur in any order. In particular, since the condition $c2$ is not a synchronisation point, there is nothing to prevent the sending of $m4$ (by instance j) to occur before the reception of $m2$ (by instance i). The same holds for the input event of $m4$ and the output events of $m5$ and $m6$.

In the case where there are several "follow-up" MSCs with the same initial condition, there is an interesting, subtle issue involved in their composition: namely, the choice of the actual follow-up MSC during a concrete run of the system. For instance, in Figure 1, after `Init` has finished $m3$ can be sent; this automatically involves continuing with `Alt1` and rejecting `Alt2`. Another possibility is that either $m5$ or $m6$ is sent, deciding the choice in favour of `Alt2`. However, *sending* $m3$ *and* $m5/m6$ *are independent actions*, since they occur in different instances. Yet the case where instances i and j simultaneously decide to send $m3$ resp. $m5/m6$ is not a valid system run. The choice between the two follow-up charts is apparently taken on a global level. This is an effect that any formal semantics has to take into account; for instance, [12] follows precisely this solution.[2]

Finally, we want to comment on an issue raised by [12]. In our opinion, and consequently in our semantics, MSC documents do not *a priori* specify finite-state systems, even though, of course, finite-state behaviour is a desirable property. For a counterexample, consider an MSC over two instances with

[1] Note that the standard does not prescribe an unambiguous starting point; rather, single MSCs are thought to represent possible fragments of behaviour.

[2] Another point of view is that such a global choice is a specification error and should be ruled out in advance. However, we are of the opinion that formulating a semantics comes properly *before* deciding whether the specified behaviour is implementable.

identical initial and final condition (i.e., a loop) containing just one message sent from one instance to the other. The execution traces of this MSC are all traces σ over $\{in(i,m)@j, out(j,m)@i\}$ such that in each prefix of σ the number of $in(i,m)@j$'s never exceeds the number of $out(j,m)@i$ and finally in σ the amount of $in(i,m)@j$'s and $out(j,m)@i$'s is equal. Clearly, this is not a finite-state recognisable language. In fact, we conjecture that it is undecidable whether the behaviour specified by an MSC document is finite-state. As in the issue of global choice, we feel that the semantics should be well-defined regardless of whether or not the behaviour is finite-state.

Table 2: Textual representation of MSC documents.

```
<document>   ::= mscdocument <docid>; { <msc>; }* endmscdocument
<msc>        ::= msc <mscid>; { <inst def>; }* endmsc
<inst def>   ::= instance <iid>; { <event>; }* endinstance
<event>      ::= <comm event>
               | action <aid>
               | condition <cid> shared all
               | concurrent { <comm event>; }* endconcurrent
<comm event> ::= in <mid> from [found] <address>
               | out <mid> to [lost] <address>
<address>    ::= <iid> | env
```

The relevant fragment of the textual grammar of MSCs is reproduced in Table 2.[3] The grammar contains the following undefined non-terminals:

- <docid>: MSC document identifiers, ranged over by $\mathcal{D}$;
- <mscid>: message sequence chart identifiers, ranged over by M;
- <iid>: instance identifiers, collected in *Inst*, such that $env \notin Inst$. We denote $Addr = Inst \cup \{env\}$ for the set of *addresses*, ranged over by i, j.
- <mid>: message identifiers, collected in *Mess* and ranged over by m;
- <aid>: internal action identifiers, collected in *Int* and ranged over by τ;
- <cid>: condition identifiers, collected in *Cond* and ranged over by c, d.

It should be clear that not every syntactically correct MSC document is acceptable: the above discussion contains a large number of "consistency requirements". In fact, one additionally needs a static semantics for MSC documents, for instance in the form of a type system, such that a document is well-formed (well-typed) if and only if it satisfies all those criteria. Some of the crucial points are:

- Outgoing and incoming messages must be matched precisely;
- The ordering imposed by messages may not be circular;

[3] In fact, the grammar is an adapted version of that in [7], but the syntax it generates is a subset of the standard.

- Global conditions (which are the only ones we model) must occur in all instances;
- All MSCs in a document must have the same set of instances.

3 THE PROCESS ALGEBRA $\mathcal{L}_{MSC}$

We now introduce the process algebra $\mathcal{L}_{MSC}$ into which we will translate MSC documents. The basic building blocks are *events* of one of the following kind:

- $out(i, m)$: normal output of $m \in Mess$ to $i \in Addr$;
- $lost(i, m)$: discarded output of $m \in Mess$ to $i \in Addr$;
- $in(i, m)$: normal reception of $m \in Mess$ from $i \in Addr$;
- $found(i, m)$: spontaneous input of $m \in Mess$ from $i \in Addr$;
- $act(\tau)$: an internal action $\tau \in Int$;
- $\surd$: a termination event.

Events of the first five kinds are collected in Evt, ranged over by e; furthermore, $Evt_{\surd} = Evt \cup \{\surd\}$ is ranged over by α. In each case, $\alpha@j$ denotes the event α taking place at instance $j \in Inst$, and $\alpha@Inst$ the collection of all such $\alpha@j$. Additionally, we assume a set of process names $Names$ to allow recursive definitions. $\mathcal{L}_{MSC}$ is then given by the following abstract grammar:

$$B ::= \delta \mid \varepsilon \mid e@i \mid B \cdot B \mid B + B \mid B \parallel_A B \mid X \mid \bot \ ,$$

where $A \subseteq Evt@Inst$ and $X \in Names$. The operators have the following intuition:

- δ is an empty, non-terminated (i.e., deadlocked) process.
- ε is an empty process, terminated at all instances (corresponding to "skip").
- $e@i$ specifies an event at instance i, and is, furthermore, *terminated* at all instances *except for* i.
- $\cdot$ specifies *weak sequential composition* of its operands. It has been introduced in [19] in a more general setting and we adapt it here to handle MSC composition. A similar operator has also been used for Interworking composition (which are the synchronous variant of MSCs) in [15] and as a sequencing operator on High-level MSCs [14]. The effect is that of ordinary sequential composition *within each instance*, whereas different instances are allowed to proceed independently. That is, termination of the first operand at a given instance i (signalled by a $\surd@i$-transition) allows the second operand to perform events $\alpha@i$ but not $\alpha@j$ for $j \neq i$. (It is important to keep in mind that a term may be terminated at one instance but not at another, so that termination of a term at one instance does not imply the inability to do some real event at another; witness $e@i$ above.)

- $+$ specifies a *choice* between its operands, which is resolved by the first non-termination event that occurs. The behaviour with respect to termination is somewhat more complex, based on the principles developed in [19]: $B_1 + B_2$ terminates at i if either of its operands does, but the choice is only resolved thereby if the other operand does *not* terminate at i. We let $\sum_{n \in N} B_n$ denote a choice over all terms B_n where n is out of some finite set N; consequently, $\sum_{n \in \emptyset} B_n$ equals δ.
- $\|_A$ is a TCSP-parallel composition [2] requiring synchronisation on all events in A; that is, the operands may do events $e@i \in A$ together (i.e., both at the same time) or events $e@i \notin A$ on their own. Moreover, we require implicit synchronisation on termination, i.e., events $\surd@i$ may also only be performed by both operands together.
- $X \in \mathit{Names}$ stands for the invocation of the process names X. Processes are defined by a process environment $\theta : \mathit{Names} \to \mathcal{L}_{MSC}$, which is assumed to be given.
- Finally, $\perp$ stands for an empty *message pool.* This is the only really non-standard operator in $\mathcal{L}_{MSC}$; it is defined especially to model MSCs. Message pools will be used for the modelling of the asynchronous communication of MSCs using the synchronous communication of $\mathcal{L}_{MSC}$. Message pools are inspired by the modelling of asynchronous communication in of coordination languages; see, e.g., [3]. A message pool is used to buffer the sent but not yet received messages. Operationally, this is done by generating a parallel $in(i,m)@j$-event whenever an $out(j,m)@i$-event is sent.

As usual, the formal semantics of $\mathcal{L}_{MSC}$ will be derived via SOS rules generating a labelled transition system over $\mathit{Evt}_{\surd}@\mathit{Inst}$. We recall the general definition:

1 Definition. A *labelled transition system* over a set of labels L is a tuple $\langle S, \to, q \rangle$ such that S is a set of *states*, $\to \subseteq S \times L \times S$ is a *transition relation* and $q \in S$ is the *initial state.*

In our case, the labels are taken form the localised alphabet $\mathit{Evt}_{\surd}@\mathit{Inst}$. A transition $B \xrightarrow{\surd@i} B'$ means that the term B may signal termination of instance i, thereby evolving into B'. For instance, a term B not referring to instance i in any of its events (and not containing δ) may always signal termination of i; it in fact has no information on i and assumes it to be terminated. Table 3 gives the structural operational semantics of $\mathcal{L}_{MSC}$. This gives rise to a transition system semantics for each $\mathcal{L}_{MSC}$-term. We briefly discuss the intuitive meaning of some of the operational rules. The first rule for weak sequential composition equals the one for (normal) strong sequential composition: the first process is allowed to proceed. The second rule, on the other hand, describes the fact that the second component may also execute events at instance i if in the first component instance i is terminated, as specified by the second rule. For a term that only refers to events from one instance i, weak sequential composition coincides with strong sequential composition; i.e., the second operand starts execution only if the first one is completely terminated.

Table 3: Operational semantics for $\mathcal{L}_{MSC}$

Empty process

$$\frac{}{\varepsilon \xrightarrow{\surd@i} \varepsilon}$$

Events

$$\frac{}{e@i \xrightarrow{e@i} \varepsilon} \qquad \frac{j \neq i}{e@j \xrightarrow{\surd@i} e@j}$$

Weak sequential composition

$$\frac{B_1 \xrightarrow{e@i} B_1'}{B_1 \cdot B_2 \xrightarrow{e@i} B_1' \cdot B_2} \qquad \frac{B_1 \xrightarrow{\surd@i} B_1' \quad B_2 \xrightarrow{\alpha@i} B_2'}{B_1 \cdot B_2 \xrightarrow{\alpha@i} B_1' \cdot B_2'}$$

Choice

$$\frac{B_1 \xrightarrow{e@i} B_1'}{B_1 + B_2 \xrightarrow{a@i} B_1'} \qquad \frac{B_2 \xrightarrow{e@i} B_2'}{B_1 + B_2 \xrightarrow{a@i} B_2'}$$

$$\frac{B_1 \xrightarrow{\surd@i} B_1' \quad B_2 \xrightarrow{\surd@i} B_2'}{B_1 + B_2 \xrightarrow{\surd@i} B_1' + B_2'}$$

$$\frac{B_1 \xrightarrow{\surd@i} B_1' \quad B_2 \not\xrightarrow{\surd@i}}{B_1 + B_2 \xrightarrow{\surd@i} B_1'} \qquad \frac{B_2 \xrightarrow{\surd@i} B_2' \quad B_1 \not\xrightarrow{\surd@i}}{B_1 + B_2 \xrightarrow{\surd@i} B_2'}$$

Parallel composition

$$\frac{B_1 \xrightarrow{e@i} B_1' \quad e@i \notin A}{B_1 \parallel_A B_2 \xrightarrow{e@i} B_1' \parallel_A B_2} \qquad \frac{B_2 \xrightarrow{e@i} B_2' \quad e@i \notin A}{B_1 \parallel_A B_2 \xrightarrow{e@i} B_1 \parallel_A B_2'}$$

$$\frac{B_1 \xrightarrow{\alpha@i} B_1' \quad B_2 \xrightarrow{\alpha@i} B_2' \quad \alpha@i \in A \cup \surd@Inst}{B_1 \parallel_A B_2 \xrightarrow{\alpha@i} B_1' \parallel_A B_2'}$$

Process names

$$\frac{\theta(X) \xrightarrow{\alpha@i} B'}{X \xrightarrow{\alpha@i} B'}$$

Empty pool

$$\frac{j \in Inst}{\bot \xrightarrow{out(j,m)@i} in(i,m)@j \parallel_\emptyset \bot} \qquad \frac{}{\bot \xrightarrow{\surd@i} \bot}$$

Choices can be resolved both by normal events and by termination signals within one operand, i.e. a term containing choices is terminated if one of its components is. This is in accordance with the usual interplay of termination and choice; see, for instance, [1]. However, if both operands of a choice may terminate for an instance i, the choice is not yet resolved and both components execute their termination transition. In this way, we avoid that a choice can be resolved by the termination of an instance not participating in an execution. For instance, using the rules in Table 3, we can derive $(e@1 + e'@2) \cdot e''@3 \xrightarrow{e''@3} (e@1 + e'@2) \cdot \varepsilon$ rather than $(e@1 + e'@2) \cdot e''@3 \xrightarrow{e''@3} e@1 \cdot \varepsilon$ or $(e@1 + e'@2) \cdot e''@3 \xrightarrow{e''@3} e'@2 \cdot \varepsilon$.

Process names behave according to their instantiation by θ.

The semantics of $\bot$ shows a process algebraic modelling of a message pool: If the pool receives a message from a sender, i.e., the pool performs $out(j,m)@i$,

it stores it for the receiver by evolving into the term $in(i,m)@j \parallel_\emptyset \perp$. Delivering the message to the receiver will be done by synchronising on $in(i,m)@j$; this does not cause any interference with the rest of the message pool. The details of the semantics of $\perp$ will become clearer in the next section. There we describe how MSC documents can be translated into $\mathcal{L}_{MSC}$.

4 TRANSLATION OF MSC DOCUMENTS INTO $\mathcal{L}_{MSC}$

We assume that the document in question has instances *Inst* (i.e., all MSCs in the document contain definitions of the instances *Inst*), messages *Mess* and conditions *Cond*; we let $Cond \subseteq Names$, i.e., conditions serve as names for recursive processes. For the corresponding process definition θ see below. We let $init(M)$ equal the initial condition of an MSC M, if it exists, and ε otherwise; likewise, $fin(M)$ equals the final condition of M, if it exists, and ε otherwise. The translation is defined in Table 4. MSCs, instances and events are mapped

Table 4: Translation functions.

$[.]_{\texttt{doc}} : \texttt{<doc>} \to 2^{\mathcal{L}_{MSC}}$

$[\texttt{document } \mathcal{D};\ \texttt{<m>}_1;\ \cdots;\ \texttt{<m>}_n;\ \texttt{endmscdocument}]_{\texttt{doc}} = \{[\texttt{<m>}_k]_{\texttt{msc}} \mid 1 \le k \le n\}$

$[.]_{\texttt{msc}} : \texttt{<msc>} \to \mathcal{L}_{MSC}$

$[\texttt{msc } M;\ \texttt{<i>}_1;\ \cdots;\ \texttt{<i>}_n;\ \texttt{endmsc}]_{\texttt{msc}} = (\ \perp \parallel_{pool} ([\texttt{<i>}_1]_{\texttt{inst}} \parallel_\emptyset \cdots \parallel_\emptyset [\texttt{<i>}_n]_{\texttt{inst}})\ ) \cdot fin(M)$

where $pool = \{in(i,m)@j, out(i,m)@j \mid i,j \in Inst, m \in Mess\}$

$[.]_{\texttt{inst}} : \texttt{<inst def>} \to \mathcal{L}_{MSC}$

$[\texttt{instance } i;\ \texttt{<e>}_1;\ \cdots;\ \texttt{<e>}_n;\ \texttt{endinstance}]_{\texttt{inst}} = [\texttt{<e>}_1]^i_{\texttt{event}} \cdot \ \ldots\ \cdot [\texttt{<e>}_n]^i_{\texttt{event}}$

$[.]^i_{\texttt{event}} : \texttt{<event>} \to \mathcal{L}_{MSC}$

$[\texttt{<ce>}]^i_{\texttt{event}} = [\texttt{<ce>}]^i_{\texttt{comm}}$

$[\texttt{action } \tau]^i_{\texttt{event}} = act(\tau)@i$

$[\texttt{condition } c \texttt{ shared all}]^i_{\texttt{event}} = \varepsilon$

$[\texttt{concurrent <ce>}_1;\ \cdots;\texttt{<ce>}_n;\ \texttt{endconcurrent}]^i_{\texttt{event}} = [\texttt{<ce>}_1]^i_{\texttt{comm}} \parallel_\emptyset \cdots \parallel_\emptyset [\texttt{<ce>}_n]^i_{\texttt{comm}}$

$[.]_{\texttt{comm}} : \texttt{<comm event>} \to \mathcal{L}_{MSC}$

$[\texttt{in } m \texttt{ from } j]^i_{\texttt{comm}} = in(j,m)@i$

$[\texttt{in } m \texttt{ from found } j]^i_{\texttt{comm}} = found(j,m)@i$

$[\texttt{out } m \texttt{ to } j]^i_{\texttt{comm}} = out(j,m)@i$

$[\texttt{out } m \texttt{ to lost } j]^i_{\texttt{comm}} = lost(j,m)@i$

to $\mathcal{L}_{MSC}$-terms and MSC documents to *sets* of $\mathcal{L}_{MSC}$-terms. The following abbreviations occur in the table:

- $B_1 \cdot \ \ldots\ \cdot B_n$, denoting the right-associative weak sequential composition of a series of terms B_k for $1 \le k \le n$. The combined term equals B_1 if $n = 1$ and ε if $n = 0$.

- $B_1 \parallel_\emptyset \ldots \parallel_\emptyset B_n$, denoting the right-associative parallel composition of the terms B_k, $1 \le k \le n$. The combined term equals B_1 if $n = 1$ and ε if $n = 0$.

B^M denotes the translation of an MSC M of a given document $\mathcal{D}$ according to Table 4, and B_i^M denotes the sub-term of instance i of MSC M. The terms resulting from the translation of $\mathcal{D}$ are to be interpreted in a process environment $\theta_{\mathcal{D}}$ defined as follows:

$$\text{for all } c \in \mathit{Cond} \qquad \theta_{\mathcal{D}} : c \mapsto \textstyle\sum_{\mathit{init}(M)=c} B^M \ .$$

In words, the behaviour of a condition equals the sum of the MSCs of which it is the initial condition. Since the translation of an MSC ends in the invocation of its final condition (if any), the "glueing together" of MSCs works as planned: after an MSC is terminated, a choice of continuations exists, as determined by the condition names. Moreover, since we are using weak composition, termination is local to an instance.

2 Example. For the MSC document in Figure 1 we get the following instance and MSC terms:

$$\begin{array}{lcl}
B_i^{\mathtt{Init}} & = & \mathit{out}(j,m1)@i \cdot \mathit{in}(j,m2)@i \\
B_j^{\mathtt{Init}} & = & \mathit{in}(i,m1)@j \cdot \mathit{act}(a)@j \cdot \mathit{out}(i,m2)@j \\
B_i^{\mathtt{Alt1}} & = & \mathit{in}(j,m3)@i \cdot \mathit{out}(j,m4)@i \\
B_j^{\mathtt{Alt1}} & = & \mathit{out}(i,m3)@j \cdot \mathit{in}(i,m4)@j \\
B_i^{\mathtt{Alt2}} & = & \mathit{out}(j,m5)@i \parallel_{\emptyset} \mathit{out}(\mathit{env},m6)@i \\
B_j^{\mathtt{Alt2}} & = & \mathit{in}(i,m5)@j \cdot \mathit{lost}(i,m7)@j \\
B^{\mathtt{Alt2}} & = & (\bot \parallel_{\mathit{pool}} (\mathit{out}(j,m5)@i \parallel_{\emptyset} \mathit{out}(\mathit{env},m6)@i \parallel_{\emptyset} \\
 & & \qquad \mathit{in}(i,m5)@j \cdot \mathit{lost}(i,m7)@j)) \cdot c3
\end{array}$$

Moreover, we get the following process environment:

$$\begin{array}{rcl}
\theta_{\mathcal{D}} : c1 & \mapsto & (\bot \parallel_{\mathit{pool}} (\mathit{out}(j,m1)@i \cdot \mathit{in}(j,m2)@i \parallel_{\emptyset} \\
 & & \qquad \mathit{in}(i,m1)@j \cdot \mathit{act}(\tau)@j \cdot \mathit{out}(i,m2)@j)) \cdot c2 \\
c2 & \mapsto & (\bot \parallel_{\mathit{pool}} (\mathit{in}(j,m3)@i \cdot \mathit{out}(j,m4)@i \parallel_{\emptyset} \\
 & & \qquad \mathit{out}(i,m3)@j \cdot \mathit{in}(i,m4)@j)) \cdot c2 \\
 & & + (\bot \parallel_{\mathit{pool}} ((\mathit{out}(j,m5)@i \parallel_{\emptyset} \mathit{out}(\mathit{env},m6)@i) \parallel_{\emptyset} \\
 & & \qquad \mathit{in}(i,m5)@j \cdot \mathit{lost}(i,m7)@j)) \cdot c3 \\
c3 & \mapsto & \delta
\end{array}$$

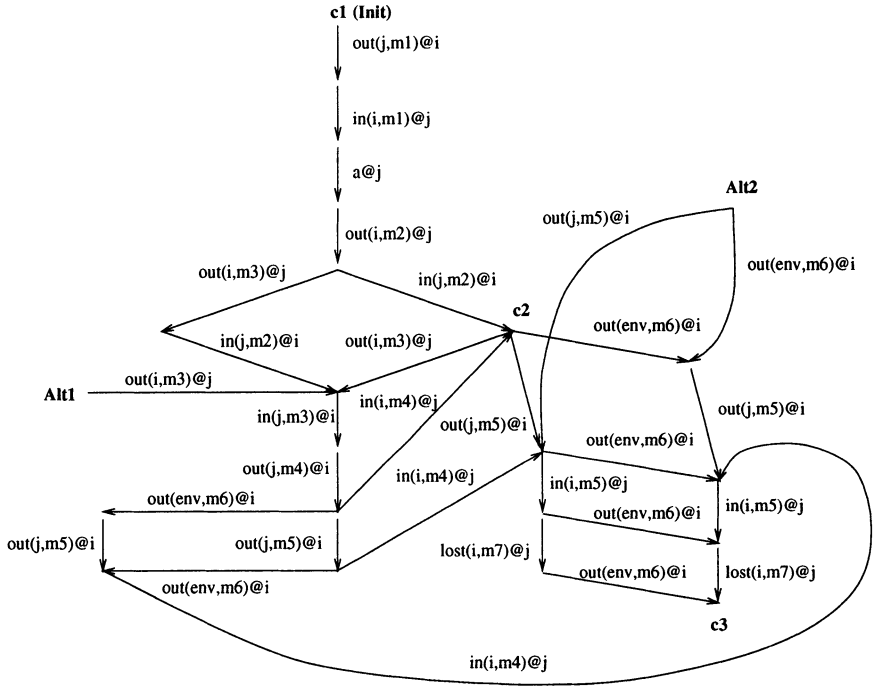
It can be seen from the translation that the instances may proceed independently in parallel (except that they have to synchronise on termination) but have to synchronise with the message pool on all communication (and termination) events. Thus, the role of $\bot$ is as described in the previous section: It takes the message from a sender by performing a synchronised $\mathit{out}(i,m)@j$-event and stores the corresponding receive event $\mathit{in}(j,m)@i$ until the receiving instance, j, wants to synchronise on that event.

In Fig. 5 the transition system for our example is given, as derived from the structural operational semantics. The states corresponding to the three MSCs are marked, as are the states corresponding to the conditions. An interesting parts of the behaviour is the region surrounding the state marked $c2$.

- The global choice between the outgoing $out(i, m3)@j$- and $out(j, m5)@i$-transitions is as described in Section 2.
- Starting from `Init`, this choice can be taken prematurely in favour of $out(i, m3)@j$, which can already be done before $in(j, m2)@i$ occurs. This is the effect of weak sequential composition.

Other noteworthy aspects are that the event $in(i, m4)@j$ can be executed concurrently to the events $out(j, m5)@i$ and $out(env, m6)@i$, and that the combined behaviour may loop around, i.e., is infinite.

Figure 5: Example: Transition system for the document in Fig. 1.



5 RESULTS

In this section we discuss two issues concerning our semantics: its consistency with the standard semantics of [13], and an algebraic property concerning message pools.

The standard semantics The standard MSC semantics in [7], based on the work of Mauw and Reniers in [13], is also process algebraic. We show consistency of our semantics with the standard by comparing the resulting transition systems up to strong bisimulation equivalence [16]. Since the standard semantics does not capture continuations via conditions, a comparison can be made only for single MSCs and not for documents.

For the purpose of comparison, we recall the relevant part of [13]. We refrain from giving the full semantics but instead focus on the major differences. The

Table 6: Operational semantics for additional operators.

Termination	$\dfrac{}{\varepsilon\downarrow} \qquad \dfrac{B_1\downarrow \quad B_2\downarrow}{(B_1;B_2)\downarrow} \qquad \dfrac{B_1\downarrow \quad B_2\downarrow}{(B_1 \parallel B_2)\downarrow} \qquad \dfrac{B\downarrow}{\lambda_M(B)\downarrow}$
Sequential composition	$\dfrac{B_1 \xrightarrow{e@i} B_1'}{B_1;B_2 \xrightarrow{e@i} B_1';B_2} \qquad \dfrac{B_1\downarrow \quad B_2 \xrightarrow{e@i} B_2'}{B_1;B_2 \xrightarrow{e@i} B_2'}$
State operator	$\dfrac{\alpha@i \notin pool \quad B \xrightarrow{e@i} B'}{\lambda_M(B) \xrightarrow{e@i} \lambda_M(B')} \qquad \dfrac{B \xrightarrow{out(j,m)@i} B'}{\lambda_M(B) \xrightarrow{out(j,m)@i} \lambda_{M+out(j,m)@i}(B')}$
	$\dfrac{out(j,m)@i \in M \quad B \xrightarrow{in(i,m)@j} B'}{\lambda_M(B) \xrightarrow{in(i,m)@j} \lambda_{M-out(j,m)@i}(B')}$

semantics of events is essentially the same up to some renaming. Coregions are not handled in [13], but in [7] they are translated into a free-merge of (the semantics of) all events in the coregion. The free-merge of ACP (denoted $\|$) coincides with TCSP parallel composition with an empty synchronisation set ($\|_\emptyset$ above) —except for termination, on which more below. From now on, we ignore the differences in the translation of events.

The major differences start on the level of instances. In [13], instances are interpreted as the *strong* sequential composition of their events, in contrast to weak sequential in our semantics. To define the operational semantics of strong sequential composition, instead of termination transitions local to instances ($\xrightarrow{\surd@i}$ above), Mauw and Reniers use a global termination predicate: $B\downarrow$ denotes the successful termination of process B. The operational semantics are given in Table 6. To avoid confusion, we use ; to denote strong sequential composition instead of $\cdot$ as in [13].

The second difference concerns the composition of instances into MSCs. For modelling asynchronous communication, Mauw and Reniers use a *state operator* λ_M [1]. This operator plays the role of the *message pool* in our semantics. M is a multiset containing *out*-events: If $\lambda_M(B)$ performs an *out*-event, this event is added to the set M. An *in*-event of B can only be performed if the corresponding *out*-event is an element of M; this element is then removed. (We denote addition and subtraction of multisets by $+$ and $-$, respectively). Therefore, the state operator ensures that the sending of a message occurs before its receipt.

The translation function of [13] is given by

$$S[\![\texttt{msc}\ M;\ \texttt{<i>}_1;\ \cdots;\ \texttt{<i>}_n;\ \texttt{endmsc}]\!]_{\texttt{msc}} = \lambda_\emptyset(S[\![\texttt{<i>}_1]\!]_{\texttt{inst}} \parallel \cdots \parallel S[\![\texttt{<i>}_n]\!]_{\texttt{inst}})$$

$$S[\![\texttt{instance}\ i;\ \texttt{<e>}_1;\ \cdots;\ \texttt{<e>}_n;\ \texttt{endinstance}]\!]_{\texttt{inst}} = [\![\texttt{<e>}_1]\!]^i_{\texttt{event}};\cdots;[\![\texttt{<e>}_n]\!]^i_{\texttt{event}}$$

The function $S[\![.]\!]_{\texttt{inst}}$ translates a single instance i into a process term which consists of the strong sequential composition of the events performed by i. As before, the events are translated by the function $[\![.]\!]^i_{\texttt{event}}$ given in Table 4. The function $S[\![.]\!]_{\texttt{msc}}$ translates a MSC into a parallel composition (free merge) of the terms resulting from the translation of the instances. Moreover, the term is enclosed in a state operator $\lambda_\emptyset$ to achieve the correct causal order of corresponding *out*- and *in*-events.

As mentioned above, we compare our semantics with the standard one up to strong bisimulation, where, however, we have to take the different notions of termination into account somehow. The natural thing is to consider the global termination used in the standard semantics (as indicated by $B{\downarrow}$) to be equivalent to the termination of all instances ($B \xrightarrow{\surd@i} B$ for all $i \in \mathit{Inst}$). This gives rise to the following definition:

3 Definition. Two transition systems $T_i = \langle S_i, \rightarrow, q_i \rangle$ for $i = 1, 2$ are called *bisimilar*, denoted $T_1 \sim T_2$, if there exists a relation $\mathcal{R} \subseteq S_1 \times S_2$ such that $(q_1, q_2) \in \mathcal{R}$ and whenever $(s_1, s_2) \in \mathcal{R}$ we have

- $s_1 \xrightarrow{\alpha@i} s_1'$ implies $\exists s_2' \colon s_2 \xrightarrow{\alpha@i} s_2'$ such that $(s_1', s_2') \in \mathcal{R}$;
- $s_2 \xrightarrow{\alpha@i} s_2'$ implies $\exists s_1' \colon s_1 \xrightarrow{\alpha@i} s_1'$ such that $(s_1', s_2') \in \mathcal{R}$.

Note that our operational semantics (Table 3) as well as the standard one (Table 6) fall into the SOS format of [20], and therefore bisimulation is a congruence. Consistency of the semantics holds for all *well-formed* MSC definitions, which we defined above to mean that for every outgoing message, a corresponding incoming one is specified as well. Since we are just comparing single MSCs, we interpret all process names (i.e., MSC conditions) in the environment $\theta_{\mathcal{D}}$ where $\theta_{\mathcal{D}}(c) = \varepsilon$ (no continuation).

1 Theorem. Let <i> be an instance definition and <m> a well-formed MSC definition. Then $[\![\texttt{<i>}]\!]_{\texttt{inst}} \sim S[\![\texttt{<i>}]\!]_{\texttt{inst}}$ and $[\![\texttt{<m>}]\!]_{\texttt{msc}} \sim S[\![\texttt{<m>}]\!]_{\texttt{msc}}$.

The proof can be found in the full version of the paper [5].

Distribution of the message pool Our translation of MSC documents into $\mathcal{L}_{MSC}$-terms introduces a message pool per MSC: control is transferred from one MSC to the next (as directed by the conditions) only if the message pool of the first contains no more remaining elements. This does not correspond to the actual assumption about the communication structure underlying MSCs, where there is a single global medium. Now we indicate that our model is equivalent to such a global medium. The crucial algebraic properties necessary to show this are the following:

$$(B_1 \parallel_{pool} \perp) \cdot (B_2 \parallel_{pool} \perp) = (B_1 \cdot B_2) \parallel_{pool} \perp \quad (1)$$

$$(B_1 \parallel_{pool} \perp) + (B_2 \parallel_{pool} \perp) = (B_1 + B_2) \parallel_{pool} \perp \quad (2)$$

The first of these states that communication with an empty message pool may be distributed over weak sequential composition. This is valid up to $\sim$ if B_1 and B_2 are well-formed in the sense that they contain as many $\mathit{out}(i,m)@j$-events as $\mathit{in}(j,m)@i$-events. In fact, (1) is reminiscent of the *communication closed layers* law of [4], which also plays an important role in the work of Janssen and Zwiers [8]. The second equation states a similar distribution property for the choice operator.

As a consequence of these laws, we can give an alternative translation function featuring a global message pool rather than local ones to each MSC, by replacing the functions $[\![.]\!]_{\mathtt{doc}}$ and $[\![.]\!]_{\mathtt{msc}}$ of Table 4 by the following:

$G[\![\mathtt{document}\ \mathcal{D};\ \mathtt{<m>}_1;\ \cdots;\ \mathtt{<m>}_n;\mathtt{endmscdocument}]\!]_{\mathtt{doc}} = \{\bot \parallel_{pool} G[\![\mathtt{<m>}_k]\!]_{\mathtt{msc}} \mid 1 \leq k \leq n\}$

$G[\![\mathtt{msc}\ M;\ \mathtt{<i>}_1;\ \cdots;\ \mathtt{<i>}_n;\ \mathtt{endmsc}]\!]_{\mathtt{msc}} = ([\![\mathtt{<i>}_1]\!]_{\mathtt{inst}} \parallel_\emptyset \cdots \parallel_\emptyset [\![\mathtt{<i>}_n]\!]_{\mathtt{inst}}) \cdot \mathit{fin}(M)$

The following theorem states the equivalence of $[\![.]\!]$ and $G[\![.]\!]$

2 Theorem. Let <m> be an MSC definition. Then $\bot \parallel_{pool} G[\![\mathtt{<m>}]\!]_{\mathtt{msc}} \sim [\![\mathtt{<m>}]\!]_{\mathtt{msc}}$.

We regard the fact that this equivalence can be shown by relying on the algebraic distribution properties (1) and (2) as evidence of the power of the process algebraic approach.

6 CONCLUSION

We have presented a structural operational semantics for Message Sequence Chart documents based on process algebras. The semantics, which is consistent with the standard semantics, covers all basic MSC features *including* the use of conditions as composition operators. Instance creation can quite easily be mimicked with asynchronous communication. The creating instance sends a special message `create` to the instance to be created which as an initial action has to perform a receive action of the create message (a kind of "await creation"). Instance decomposition can be incorporated in a way similar to the standard semantics, namely by syntactic substitution in the MSC term.

The main advantage of our semantics is the conceptually clear modelling of the issues termination and sequential composition. Sequential composition of MSCs in the context of high-level MSCs, also based on [19], has been given in [14]. In contrast to [14], we use a single concept of termination and just one sequential composition operator; the latter can be used both on the level of instances and for MSCs. Moreover, we have adapted the weak sequential composition operator of [19] to the specific setting of MSCs, allowing to replace the complex notion of "permission" by a concept of *local termination* which is the natural translation of termination into the area of MSCs. Starting from our semantics, it should be easy to develop a semantics for high-level MSCs; sequential composition in high-level MSCs is already present here.

Acknowledgements. We are grateful to Peter Niebert for cooperation in an initial draft version of this paper.

References

[1] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, July 1984.

[3] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. Technical Report UBLCS-97-6, University of Bologna, May 1997. To appear in: Theoretical Computer Science.

[4] T. Elrad and N. Francez. Decomposition of distributed progams into communication closed layers. *Science of Computer Programming*, 2, 1982.

[5] T. Gehrke, M. Huhn, A. Rensink, and H. Wehrheim. An Algebraic Semantics for Message Sequence Chart Documents. Hildesheimer Informatik-Bericht 5/98, Universität Hildesheim, Institut für Informatik, May 1998.

[6] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net Based Semantics Definition for Message Sequence Charts. In *Proceedings of the 6th SDL Forum (SDL '93)*, 1993.

[7] International Telecommunication Union. *Message Sequence Chart (MSC)*, z.120 edition, 1996.

[8] W. Janssen and J. Zwiers. From sequential layers to distributed processes. In *Principles of Distributed Computing*, pp. 215–227. ACM, 1992.

[9] B. Jonsson and J. Parrow, eds., *Concur '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[10] P. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[11] P. B. Ladkin and S. Leue. What do Message Sequence Charts mean? In R. L. Tenney, P. D. Amer, and M. Ü. Uyar, eds., *Formal Description Techniques VI*, IFIP Transactions C, pp. 301–316. North-Holland, 1994.

[12] P. B. Ladkin and S. Leue. Four issues concerning the semantics of Message Flow Graphs. In D. Hogrefe and S. Leue, eds., *Formal Description Techniques VII*, pp. 355–369. Chapman & Hall, 1995.

[13] S. Mauw and M. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[14] S. Mauw and M. Reniers. High-level message sequence charts. In A.Cavalli and A.Sarma, eds., *SDL'97: Time for testing - SDL, MSC and Trends*. Elsevier, 1997.

[15] S. Mauw, M. v. Wijk, and T. Winter. A formal semantics of synchronous Interworkings. In O. Færgemand and A. Sarma, eds., *SDL'93: Using Objects*, pp. 167–178, North-Holland, 1993.

[16] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[17] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. Smith. *Systems Engineering Using SDL-92*. North Holland, 1994.

[18] Rational Software Corporation. *UML Notation Guide – version 1.1*, Sept. 1997. URL: http://www.rational.com/uml.

[19] A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In Jonsson and Parrow [9], pp. 226–241.

[20] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. In Jonsson and Parrow [9], pp. 433–448.