URL: http://www.elsevier.nl/locate/entcs/volume65.html 17 pages

Formal Techniques for Analysing Scenarios using Message Sequence Charts

Purandar Bhaduri, R. Venkatesh and Girish K. Palshikar

TRDDC, Tata Consultancy Services
54 B, Hadapsar Industrial Estate,
Pune 411 013, India
Email: {pbhaduri,rvenky,girishp}@pune.tcs.co.in

Abstract

This paper describes light-weight formal techniques based on Message Sequence Charts (MSCs) for capturing and validating early requirements and design. Our focus is on ease of use in specifying, simulating and validating scenarios, and checking their desired properties efficiently. We discuss how the formalism of High Level Message Sequence Charts (HMSCs or MSC'96), can be used to capture scenarios in use cases, thus enabling the use of tools for analysing them. We then present two formal semantics for HMSCs – an intuitive linear time semantics based on runs, and an operational semantics in terms of a labelled transition system. Next we present a way of describing desired properties of use case scenarios using templates, for validating scenarios with respect to informal requirements. The correctness properties of a collection of MSCs can then be established by efficient algorithms for finding paths in a directed graph representing the precedence relation on the events of the MSCs. We have implemented the operational semantics and the verification algorithms in the form of a simulation and verification tool for analysing scenarios.

1 Introduction

Our aim in this paper is the application of formal methods in the capture and validation of user requirements. The goal is to formalize requirements captured by use cases [8], as employed in object-oriented modelling based on UML [4]. We intend to provide a rigorous semantics to use cases in UML using the formal notation of Message Sequence Charts (MSCs) [7,11,12]. This brings to bear several benefits to the modelling of use case scenarios. Since MSCs are a formal notation, with a formal syntax and semantics, tools can be built for analysing them. Moreover, MSCs extend UML sequence diagrams in essential ways, by permitting multiple scenarios with branching and loops to be presented in one model or diagram, and by offering the power of hierarchical structuring for ease of understanding and reuse.

MSCs offer an intuitive and visual way of describing requirements by focusing on message exchanges among communicating entities in software systems. The recent standardization of syntax and semantics of MSCs (MSC'96 or Z.120) by ITU [7] and their popularity for capturing early design requirements (see [1,6]) have resulted in commercial tools for analysing their properties. Extensions of MSCs, called *Live Sequence Charts* (LSCs) have been proposed as a semantically sound foundation for use cases and scenarios in [5].

This paper has goals similar to [2,5], namely to develop formal analysis techniques and tools for effective use of scenario-based requirements in software engineering. However, our focus is on *ease of use* in specification and validation rather than *expressive power*. Since most users, designers and developers are unlikely to be familiar with formal specification and verification techniques, we try to hide as much of the formality as possible behind a tool, and present the user with a friendly and intuitive visual interface for analysing scenarios. In particular, we offer the user the ability to validate system requirements by simulating the various scenarios in a use case in an interactive fashion. To this end, we have proposed an operational semantics of the HMSC notation, which has been implemented in a simulator. As another possible way of validating requirements, we propose a small specification language of templates for stating desirable properties of a collection of MSCs. This language is intuitive, and can be presented to the user in the form of drop-down menus and list boxes. The verification of these properties is achieved by searching for appropriate paths in the directed graph representing the precedence relation on the events of the MSCs. Our specification language of templates is powerful enough to capture both *precedence* and *response* (or *consequence*) properties [9]. A precedence property states that the occurrence of one state/event is a necessary pre-condition for an occurrence of another. An example of a precedence property is "A phone must never alert when it is off-hook." A response property, on the other hand, states that when one state occurs then an occurrence of another must follow. An example is "If a phone is alerting when it goes off-hook, then a connection is established to another phone."

While the idea of template matching in MSCs has been explored in [10], the focus there has been on patterns specifying the relative ordering of events and efficient algorithms for the matching process. The authors specify templates as MSCs and matching is defined as the existence of a homomorphic embedding between precedence relations. Moreover, the difference between *visual* and *precedence order* adds complexity to the template matching procedure. In contrast, our templates are stated in terms of linearizations of event orderings, where the events are sending and receiving of named messages. As a result, our algorithms have polynomial complexity for the template properties we have considered.

The work reported here is part of a programme for incorporating behaviour modelling in the context of a meta-model based case tool. Since scenarios

present just one facet of system dynamics, they have to integrate with other development views, both static (e.g., classes, objects) and dynamic (e.g., state machines). Our goal is to arrive at a common semantic domain for both static and dynamic views of a system. Each individual view, such as MSCs or state machines, would map into this common semantic domain. This unified semantic modelling will ensure consistency between views and compatibility of code generated from them. In this paper, we do not pursue this general framework and concentrate on MSC based analysis techniques in formal software engineering.

The main contribution of the paper is two-fold. First, the operational semantics of HMSCs we have presented in the form of a labelled transition system (lts) is new. We have built a simulator for HMSCs which implements this semantics by running the abstract machine corresponding to the lts. This allows the user to explore various scenarios described by an HMSC, including simulating the behaviour of branch points and iterative loops. Second, the verification framework we have proposed, based on property specification using patterns and efficient verification algorithms, has not been explored before. The main advantage of such an approach to formal verification is the ease of use and efficiency. The user does not have to be conversant with temporal logic or model checking technology in order to carry out simple verification and validation tasks. Because of the polynomial time complexity of the algorithms for property checking, the state explosion problem encountered in traditional model checking approaches is avoided.

This paper is organized as follows. In Section 2 we briefly describe the MSC formalism and discuss how it can formally capture the flows in use case scenarios. In Section 3 we describe a linear time semantics of HMSC based on runs and an operational semantics suitable as a basis for a simulation tool for HMSCs. Section 4 describes techniques for specification and verification of properties of MSCs. Section 5 puts our work in the perspective of our programme for formal software engineering.

2 Message Sequence Charts

Message Sequence Charts (MSCs) are a graphical language for the description of the interactions between entities, standardized by the ITU-TS. Figure 1 is an example of a basic MSC depicting a typical scenario in a telephone call.

The vertical lines in the MSC, labelled *Caller*, *Switch* and *Callee* denote communicating entities, also known as *instances*. An instance in an MSC shows the flow of time from top to bottom. The horizontal arrows between the instances denote messages. An arrow starts at the sending instance and ends at the receiving instance. The hexagons in the figure, labelled *Idle*, *Talking* and *Connected* are known as *conditions*. For details about the notation refer to the ITU recommendation Z.120 [7].

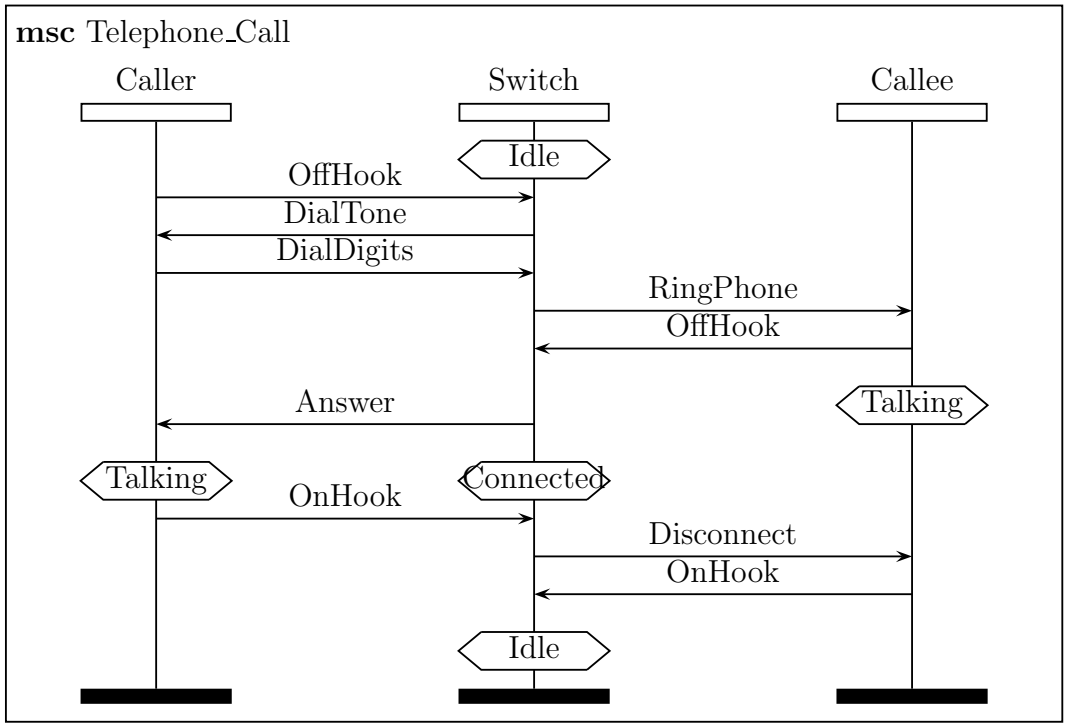

Fig. 1. Message Sequence Chart for a Typical Scenario in a Telephone Call

The ITU recommendation also defines High-level MSCs (HMSCs) for hierarchically structuring multiple scenarios using iteration and branching. An example of an HMSC is *Telephone_Call_Scenarios*, shown at the left of Figure 2, where each of the rounded boxes, known as *nodes* or *MSC references*, refers to either a basic MSC or another HMSC. The edges between nodes depict flow of control. Branches and loops depict alternative flows and iterative actions in scenarios. In the Figure, *Cancel* and *Disconnect* are MSC references which refer to basic MSCs, shown on the right. All the other nodes in HMSC *Telephone_Call_Scenarios* refer to basic MSCs which are not shown.

The HMSC *Telephone_Call_Scenarios* in Figure 2 depicts more than one scenario that can occur in a telephone call, in addition to the normal flow depicted in Figure 1:

(i) The caller can disconnect before the callee answers. This situation is captured by the outgoing branch from the node *Request* to the node *Cancel*. The normal flow corresponds to the outgoing branch from *Request* to *Reply*.

(ii) The backward loop from *Cancel* and *Disconnect* to *Request* captures the possibility of repetitive or iterative actions.

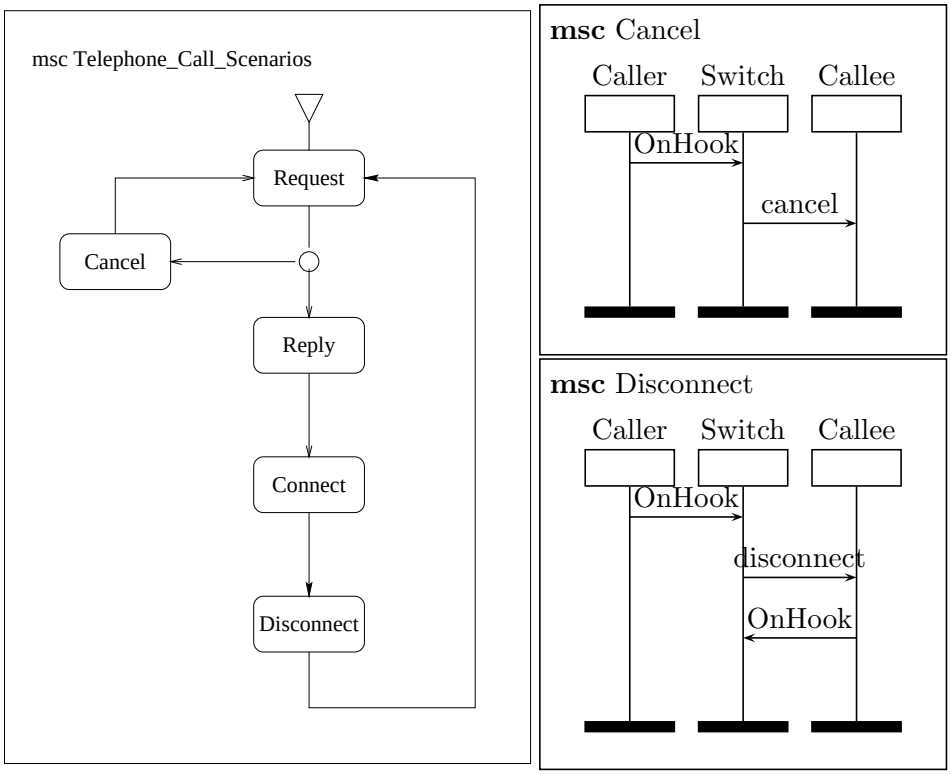

Fig. 2. High Level MSC

Our MSC-based formalization of use cases is aimed at developing tools for describing and analysing use case scenarios. The benefits gained are in the form of tool support for simulation, analysis, test case generation and formal verification. A prerequisite for such analyses is the existence of a formal semantics for the MSC and HMSC notations, the subject of the next section.

3 Semantics of HMSCs

In this section we present two semantics for HMSCs – a linear time semantics based on the notion of runs and an operational semantics in terms of a labelled transition system. While the former is useful for understanding and reasoning about MSCs, the latter is more suitable for implementing the behaviour of HMSCs in a simulator.

3.1 Linear Time Semantics

We present a *linear time* semantics of our MSC language based on runs, as is done by Damm and Harel in [5]. For us, a *run* of an HMSC is a finite or

infinite sequence of events. The events are the sending or the receiving of a message msg from instance i to instance j, or timer events (set, reset and timeout). Message passing can be either synchronous or asynchronous, and we assume that there is a way to annotate messages with this information.

We start with the abstract syntax of basic MSCs as in [5]. With each instance i in an MSC m we associate a finite number of discrete *locations* l, which are numbered from the top of the instance i to the bottom, using an index in $\{0, \ldots, lmax(m, i)\}$. Each location l on instance i in an MSC m, denoted $\langle i, l\rangle_m$, is labelled with either a condition or an event. In other words, a location refers to a region on an MSC instance that contains a condition, a message send or a message receive, or a timer event (set, reset, timeout). We will drop the subscript m from a location when the MSC being referred to is clear from the context.

The semantics of a basic MSC m is defined in terms of the partial order $\leq_m$ induced by m on its set of locations $\langle i, l\rangle$. The partial order is obtained from the following precedence relation R_m:

- order along an instance line:
  $\langle i, l\rangle R_m \langle i, l+1\rangle$, unless the two consecutive locations are in a co-region;
- order induced from message sending:
  if $\langle i, l\rangle$ is a send event and $\langle i', l'\rangle$ is the corresponding receive event for the message, then $\langle i, l\rangle R_m \langle i', l'\rangle$;
- synchronous messages block sender until receipt:
  If $\langle i, l\rangle$ is a send event and $\langle i', l'\rangle$ is the corresponding receive event for a synchronous message, then $\langle i', l'\rangle R_m \langle i, l+1\rangle$;
- shared conditions induce synchronization barrier:
  if the locations $\langle i, l\rangle$ and $\langle i', l'\rangle$ share a condition c then $\langle i, l\rangle R_m \langle i', l'+1\rangle$.

We assume that the MSC m is *well-formed* i.e., the relation R_m is acyclic. We call R_m the *precedence relation* of the MSC m. The partial order $\leq_m$ is then the reflexive transitive closure of R_m. Figure 3 shows a basic MSC and its associated partial order.

Definition 3.1 The *semantics of a basic MSC* m is the set of runs compatible with $\leq_m$, i.e., the set of all linearizations of the partial order $\leq_m$. □

Note that this means that we have an interleaving semantics for concurrency – if two events are independent, then they are assumed to take place in an arbitrary order. Also note that the semantics of a basic MSC is a finite set of *finite sequences*.

Next we define an MSC-graph [2] as a restricted form of HMSC where each node is a basic MSC.

Definition 3.2 An *MSC-graph* G is a tuple $(V, \rightarrow, v^I, v^T, \mu)$ where V is a set of vertices, $\rightarrow$ a binary relation over V, v^I an initial vertex, v^T a terminal vertex and μ a labelling function that maps each vertex v to an MSC m. □

From this MSC-graph one obtains finite paths that start at the initial vertex and end at the terminal vertex, representing finite executions of the system. In addition, because of the presence of loops in the MSC-graph there are infinite paths starting at the initial vertex that represent the infinite executions of the system. In defining the semantics of an MSC graph, we have to define what it means to concatenate two MSCs asynchronously [2]. Intuitively, this corresponds to concatenating MSCs instance by instance.

Definition 3.3 The *asynchronous concatenation* of two MSCs m_1 and m_2 with the same set of instances is the partial order $\leq_{m_1;m_2}$ on $locations(m_1) \uplus locations(m_2)$ given by the transitive closure of the following relation:

$$\leq_{m_1} \uplus \leq_{m_2} \cup \{ (\langle i,l\rangle_{m_1}, \langle i,l'\rangle_{m_2}) \mid \langle i,l\rangle_{m_1} \in locations(m_1) \wedge \langle i,l'\rangle_{m_2} \in locations(m_2)\}$$

where $X \uplus Y$ denotes the disjoint union of sets X and Y. □

Definition 3.4 The *semantics of an MSC-graph G* is the set of all finite and infinite runs obtained by:

(i) asynchronously concatenating each basic MSC along each path, and then

(ii) taking the disjoint union over all paths, both finite and infinite, of the sets of runs obtained from the partial orders in the first step.

□

There is a subtlety in the above definition. A path in an MSC-graph may involve loops, with some of the nodes repeated. Sufficient care must be taken to rename the locations in the same basic MSC while performing the asynchronous concatenation operation. This is the reason for using the disjoint union operation.

Finally, an HMSC consists of a graph whose nodes are either basic MSCs or are labelled with another HMSC, allowing for nesting of graphs.

Definition 3.5 An *HMSC H* is a tuple (N, B, v^I, v^T, μ, E) where N is a finite set of nodes, B is a finite set of boxes (or supernodes), $v^I \in N \cup B$ is the initial node or box, $v^T \in N \cup B$ is the terminal node or box, μ is a labelling function that maps each node in N to an MSC, and each box in B to an already defined HMSC and E is the set of edges that connect nodes and boxes to each other.□

The meaning of an HMSC H is defined by recursively substituting each box by the corresponding HMSC to obtain an MSC-graph. The details are worked out in [2].

3.2 Operational Semantics of HMSCs

The linear time semantics of HMSCs presented in the last section, being non-constructive, is not suitable as a basis for simulation. Here we present an operational semantics which can be directly realized in the form of an abstract machine for simulating HMSCs. This semantics is presented via a labelled

transition system $(C, \iota, E, \longrightarrow)$ where C is the set of *configurations* or *states*, ι is the *initial configuration*, E is the set of *events* and $\longrightarrow \subseteq C \times E \times C$ is the *transition relation*.

As with the linear time semantics, we start with basic MSCs. The operational semantics for a basic MSC m is directly obtained from the partial order $\leq_m$ defined in Section 3.1 by using a standard techniques for obtaining an automaton from a partial order of events.

Definition 3.6 A *cut* c in a partial order $(E, \leq)$ of events is a downward closed subset of E, i.e., $e \in c$ and $e' \leq e$ imply $e' \in c$. The set of cuts obtained from E is denoted $\mathcal{C}(E)$. □

Intuitively, cuts represent consistent global states or configurations. The empty cut represents the initial configuration, when no event has occurred.

Definition 3.7 Given a partial order $(E, \leq)$ of events we define a transition relation $(\mathcal{C}(E), \emptyset, E, \longrightarrow)$ as follows: for $c, d \in \mathcal{C}(E)$ there is a transition $c \xrightarrow{e} d$ iff $d = c \cup \{e\}$. We say that the event e is *enabled* in configuration c of MSC m, denoted $enabled(e, c, m)$. □

Note that $enabled(e, \emptyset, m)$ is equivalent to e being a minimal element in the partial order $(locations(m), \leq_m)$.

Definition 3.8 The operational semantics of a basic MSC m is given by the transition system $(\mathcal{C}(E), \emptyset, E, \longrightarrow)$ where E is the partial order $(events(m), \leq_m)$, where $events(m)$ is the set of locations of m with send, receive and timer events only i.e., conditions are ignored. □

We now define the operational semantics of HMSCs. As in Section 3.1 we start with the simpler case of MSC-graphs where the level of nesting of MSCs is one.

In contrast to basic MSCs, the operational semantics of MSC-graphs presents several complications and subtleties. Due to the asynchronous concatenation used in the linear time interpretation, at any time control may reside in more than one vertex in an MSC-graph. These vertices need not even form a contiguous chain or path – there may be "holes" in them. In addition, because of loops in the MSC-graph, more than one incarnation of a vertex may be active. Further, because of branches in the graph, two different incarnations of a vertex may chose to make different choices at a branching point. The following definitions take all these complications into account.

Definition 3.9 A *configuration* C of an MSC-graph $G = (V, \rightarrow, v^I, v^T, \mu)$ is a pair of the form $< p, c >$ where $p = [v_1, \ldots, v_n]$ is a finite path in G starting at $v_1 = v^I$ and c is a configuration, in the sense of Definition 3.7, in the basic MSC $m = m_1; \ldots; m_n$, obtained by asynchronously concatenating the basic MSCs $m_1, \ldots, m_n$, where $\mu(v_i) = m_i$. □

Note that the same vertex v may occur more than once in a path p.

We define the operational semantics of an MSC-graph G through the transition system $(\mathcal{C}, < v^I, \emptyset >, E \uplus \tau, \longrightarrow)$ where $\mathcal{C}$ is the set of configurations defined as above, $E = \bigcup_{\substack{v \in V \\ m=\mu(v)}} events(m) \times \mathbf{N}$ is the set of events in all the basic MSCs in G indexed by natural numbers and τ is a special event called the *silent event*. The initial configuration $< v^I, \emptyset >$ consists of the empty path starting with the initial vertex v^I in G and the empty set of events.

Definition 3.10 The transition relation (with τ-labelled transitions) between configurations is defined by the following cases:

(i) $< [v_1, \ldots, v_n], c > \xrightarrow{<e,i>} < [v_1, \ldots, v_n], c \cup \{e\} >$ if $e \in events(m_i)$ is enabled in the configuration c.

(ii) $< [v_1, \ldots, v_n], c > \xrightarrow{\tau} < [v_1, \ldots, v_n, v_{n+1}], c >$ if the configuration c contains a maximal event on any instance in the basic MSC $m = m_1; \ldots; m_n$ or if an instance in m has no event at all. The vertex v_{n+1} is any successor of v_n in the MSC-graph G.

□

The above definition captures the intuition that one step of execution of an MSC-graph G either executes an enabled event in one of the vertices of the path $p = [v_1, \ldots, v_n]$ of G already visited, or adds a new vertex v_{n+1} to the end of p when an event in v_{n+1} becomes enabled.

The operational semantics of an HMSC $H = (N, B, v^I, v^T, \mu, E)$ is obtained by using the construction described above for flattening the HMSC into an MSC-graph. We omit the details as there is no conceptual complexity involved.

Proposition 3.11 The linear time and the operational semantics of HMSCs presented above coincide, i.e., the set of runs defined by the linear time semantics is identical to the set of traces of the labelled transition system for the operational semantics. □

3.3 Simulation of HMSCs

We have built a prototype tool for simulation of HMSCs that directly implements the operational semantics of HMSCs presented above. Due to space constraints, we present only a brief outline of the simulation algorithm. The details can be found in [3].

The simulation of basic MSCs takes as input the precedence relation R_m corresponding to MSC m defined in Section 3.1. The algorithm essentially does a topological sort of this relation, where the minimal elements at each stage represent the enabled events. Whenever there is more than one minimal element the user is offered a choice of which event to execute next. When the event is executed it is deleted from the set of events. The algorithm stops when all the events in the MSC have been executed.

The algorithm for simulating an HMSC does a nested traversal of the

corresponding directed graph, starting at the start vertex. It starts simulating the first reachable basic MSC in the graph, and appends a new MSC to it whenever a maximal event on an instance is executed. This corresponds to the τ transition in the labelled transition system corresponding to the HMSC. Whenever there is a branch in the HMSC graph, the user is prompted with a choice for the next HMSC node to be considered for execution. The simulation ends when the end vertex is reached.

4 Verification of Properties of Basic MSCs

4.1 Approach

When capturing requirements via scenarios, the requirements engineer is often interested in validating a set $S = \{M_1, ..., M_n\}$ of basic MSCs. An alternative to simulation, which could be impractical for a large n, is to allow the user to state properties which the MSCs in S should obey, and to automatically verify them. The problem is to define a verification algorithm, that takes a set S of basic MSCs and a statement P of a property (in some well defined syntax) as input, and returns yes if the set S satisfies the property P and returns no otherwise (along with a counter-example, if necessary).

In the following presentation, we assume, for simplicity, that S consists of a single basic MSC m. The techniques presented below can be easily extended to handle the general case of n basic MSCs. The extension to HMSCs is being investigated. There is a trade-off between ease of use and expressive power in the choice of notation used to specify the properties. Following [9], we assume that the properties fall into various types of pre-defined *templates*, thus sacrificing generality for ease of use and efficiency of verification. The properties are stated in terms of relative ordering of events in the traces of the input MSC. Although the property templates cannot express all the properties that may be of interest in practice, they do cover broad classes of typical properties. Moreover, it is possible to design efficient algorithms for verifying properties stated using these templates. We present the syntax of the property templates and simple graph-theoretic algorithms (based on the partial-order semantics for MSCs) for verification of these properties.

Every event in an MSC has a unique ID given by its location, as in Section 3.1. A *user-defined event* corresponds, in general, to a set of internal events. For instance, in Figure 3, the user-defined event "p1 sends *inc*" corresponds to the set of internal events $\{e_1, e_3\}$. We assume that the properties are specified in terms of user-defined events and their negation. We say that a user-defined event e *occurs* whenever any of the internal events corresponding to it *occurs*. Since a user-defined event e stands for a non-empty set $S = \{e_1, ..., e_k\}$ of internal events, the negative event $not(e)$ stands for the event $not(e_1) \wedge ... \wedge not(e_k)$.

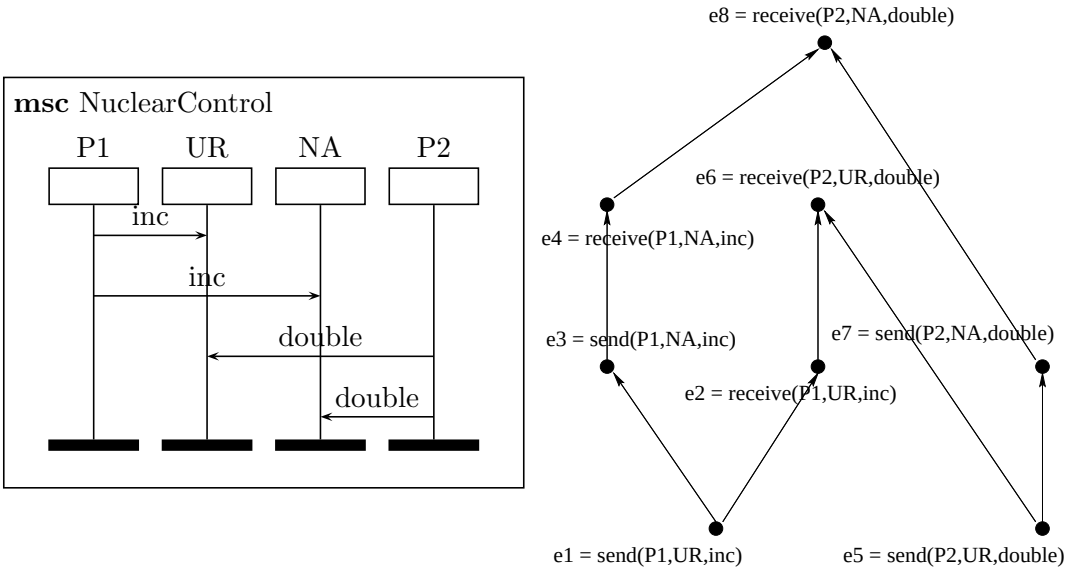

Fig. 3. An MSC and associated partial order

4.2 Syntax of Property Templates

4.2.1 Tracing

The *tracing* property asserts the occurrence of a sequence of events in the specified order in all or some of the traces of the input MSC m. The template for the property is shown below. The terms in bold separated by slashes are options and one of them has to be chosen by the user. Here each $a_i, 1 \leq i \leq n$, is either a user-defined event (i.e. a non-empty set of internal events) or its negation [1] .

The **sub-sequence** / **packed sub-sequence** of events	$[a_1, ..., a_n]$	occurs **at the beginning** / **at the end** / **somewhere**	**ever** / **always** / **never**

This template actually corresponds to several possible *patterns*, depending on the options chosen, as discussed below. We also define the semantics of each of these patterns in terms of the linearizations of the partial order corresponding to the given MSC m. For example, in Figure 3, the following

[1] For simplicity, we forbid negative events when the **packed sub-sequence** (i.e., a contiguous sub-sequence) option is selected. Similarly, when the **beginning** or **end** options are chosen, the first event in the sequence should not be a negative event.

property states that it is possible (in at least one linearization) to receive the *double* message flanked by receipts of the two *inc* messages.

The **sub-sequence** of events	["UR receives *inc*", "UR receives *double*", "NA receives *inc*"]	occurs **somewhere**	**ever**

When the user-defined events in quotes are replaced by the associated sets of internal events, this property statement translates to the following pattern.

The **sub-sequence** of events	$[\{e2\}, \{e6\}, \{e4\}]$	occurs **somewhere**	**ever**

Clearly, this property is true and there is at least one linearization - for example, $< e1, e5, e2, e3, e6, e4, e7, e8 >$ in which the sequence of events $[e2, e6, e4]$ occurs as a sub-sequence. As another example, in Figure 3, the following property states that it is possible to send two *inc* messages without any send of a *double* message in between.

The **sub-sequence** of events	["p1 sends *inc*", not("p2 sends *double*"), "p1 sends *inc*"]	occurs **somewhere**	**ever**

When the user-defined events in quotes are replaced by the associated sets of internal events, this property statement translates to the following pattern.

The **sub-sequence** of events	$[\{e1, e3\}, not(\{e5, e7\}), \{e1, e3\}]$	occurs **somewhere**	**ever**

4.2.2 Consequence

Another common and useful kind of property, called *consequence*, is specified using the following template.

The **sub-sequence** / **packed sub-sequence** / [**each** / **all** / **an**] **event(s)** from	X	leads to **sub-sequence** / **packed sub-sequence** / [**an** / **all**] **event(s)** from	Y

Here $X, Y \subseteq E$ are user-specified sets of events (X and/or Y are sequences if the sub-sequence options are chosen). This template corresponds to several possible patterns, which are used to check that certain actions are executed when their precedences occur. For example, the consequence property can be used to say that whenever a process (UR or NA) receives an *inc* message, it subsequently receives a *double* message. There is a converse *precedence* property template which asserts that if X has occurred then Y must have occurred previously in that run.

4.3 Semantics and Verification of Property Templates

Each template corresponds to several patterns, depending on which options are chosen by the user. The semantics of these patterns is defined in terms of the linear-time semantics for MSC. We also define algorithms for checking each of these property patterns. We illustrate this process for some of the patterns for tracing and consequence property templates. Recall from Section 3.1 that R_m is the precedence relation associated with an MSC m. We write $precedes(a_i, a_j)$ when $(a_i, a_j) \in R_m^*$, the transitive closure of R_m. The following algorithms assume that the transitive closure has been pre-computed. This can be done using Warshall's algorithm in time $O(n^3)$.

To keep the presentation simple, in the following we assume that (i) each event a_i is a single internal event (not a user-defined event) and (ii) none of the events a_i is a negative event. The general algorithms handling user-defined and negative events are described in the full paper [3].

(i) *Property Pattern:* **The sub-sequence** of events $\alpha = [a_1, ..., a_n]$ occurs **somewhere ever**. *Meaning:* Is there at least one linearization σ of R_m such that α is a sub-sequence of σ?

(ii) *Property Pattern:* **The sub-sequence** of events $\alpha = [a_1, ..., a_n]$ occurs **somewhere always**. *Meaning:* Is α a part- sequence of every linearization σ of R_m?

(iii) *Property Pattern:* **The sub-sequence** of events $\alpha = [a_1, ..., a_n]$ occurs **somewhere never**. *Meaning:* Is there no linearization σ of R_m such that α is a sub-sequence of σ?

The following algorithm can be used to check property pattern (1). The algorithm can be easily modified to output an actual linearization that contains α as a sub-sequence.

algorithm *tracing_a*
input set $E = \{e_1, ..., e_n\}$ of all possible events in an MSC m, $n \geq 1$
input binary precedence relation R_m on E
input $\alpha = [a_1, ..., a_k]$ a given non-empty finite sequence of events such that $a_i \in E$ for all $1 \leq i \leq k$ and there are no duplicates in α
output **true** if there is at least one linearization that contains α as a sub-sequence; **false** otherwise

```
for (j = 2; j <= k; j++)
        for (i = 1; i < j; i++)
                if (precedes(a_j, a_i))
                        return false;
                return true;
```

The correctness of the algorithm follows from the following observation.

Proposition 4.1 *Let E be a finite non-empty set containing n elements and R be a precedence relation on E. Then two elements a_1, a_2 in E occur (in that order) in some linearization of R if and only if a_2 does not precede a_1.*

The following algorithm can be used to check property pattern (2), under the same assumptions as for (1).

algorithm *tracing_b*
input set $E = \{e_1, ..., e_n\}$ of all possible events of MSC m, $n \geq 1$
input binary precedence relation R_m on E
input $\alpha = [a_1, ..., a_k]$ a given non-empty finite sequence of events such that $a_i \in E$ for all $1 \leq i \leq k$ and there are no duplicates in α
output **true** if all possible linearizations contain α as a sub-sequence; **false** otherwise

```
for (i = 1; i < k; i++)
        if ( ! precedes(a_i, a_{i+1}) )
                return false;
        return true;
```

The correctness of the algorithm follows from the following observation.

Proposition 4.2 *Let E be a finite non-empty set containing n elements and R be a precedence relation on E. Then two elements a_1, a_2 in E occur (in that order) in every linearization of R if and only if a_1 precedes a_2.*

The complexity of both algorithms is $O(k^2)$.

The consequence property template also stands for several property patterns, depending on which options are chosen by the user. We illustrate the semantics of one of the patterns for the consequence template.

Property Pattern: **An event from** X leads to **all events from** Y. *Meaning:* Is it true that for every linearization σ of R_m, there exists some

$x \in X$ and some ordered permutation Y_1 of Y such that the sequence $x \bullet Y_1$ (obtained by concatenating x with Y_1) is a sub-sequence of σ?

The following algorithm verifies the property specified by this pattern for consequence.

algorithm *consequence_b*
input set $E = \{e_1, ..., e_n\}$ of all possible events in MSC m, $n \geq 1$
input binary precedence relation R_m on E
input Two non-empty sets $X, Y \subseteq E$
output **true** if an event from X leads to all events from Y in all possible linearizations; **false** otherwise

```
for ( i = 1; i <= |X|; i++ )
{
        x := X(i); // i^th element of X
        // check if x is followed by all elements of Y in some order
        // in all linearizations
        for ( j = 1; j <= |Y|; j++ )
        {
                if ( !precedes(x, Y(j)) ) then // j^th element of Y
                        break; // x is not followed by all elements of Y
        } // end for
        if ( j > |Y| ) then // above for loop checked all elements of Y
                return true; // x is the one
} // end for
return false
```

The complexity of this algorithm is $O(|X||Y|)$. Note that none of the algorithms explicitly check all possible linearizations. Additional algorithms for other property patterns are defined similarly.

5 Conclusion

Scenario based specifications such as MSCs offer an intuitive and visual way of describing requirements. Since MSCs can be provided a formal semantics, they can be subjected to a variety of analyses. In this paper we have proposed a formal semantics for HMSCs, based on which a simulation and verification tool has been designed. Our focus is the application of this toolset in requirements validation as part of an industrial CASE tool.

In order to integrate the MSC based toolset into the software development process several problems have to be addressed. First, they have to be integrated with the use case model, a popular notation employed by UML for requirements capture. One of the challenges faced by developers in maintaining scenarios for use cases is that the requirements and hence the use cases keep on changing, and it is difficult to keep the related MSCs in sync. To over-

come this problem, we propose to design a common meta-model from which the meta-model for use cases and MSCs would be derived as views. This would make it possible to maintain the relationship between use cases and MSCs, thus enabling changes in one model to be automatically reflected in the other. Moreover, the algorithms for simulation, verification of properties and other analysis methods would work at the common meta-model level, rather than MSCs.

Another challenge in using scenarios in the software development process is maintaining their relationship to other dynamic models, such as state machines for describing object behaviour. MSCs and state machines can be understood as partial views of a system, specifying inter-object collaboration and intra-object reactive behavior respectively. Since these two views are not independent, there must be a formal relationship that must be preserved between the two in order that they describe a coherent system. We propose to address this issue by mapping these dynamic views to a common semantic framework of labelled transition system with a structure on the states. This semantic domain can, in principle, interpret all dynamic views of a system, possibly with suitable parameterization or variation points, for example, with respect to concurrency or granularity of transitions. The semantics of MSCs presented in Section 3 is a special case, which can be thought of as specializing the common semantic domain with respect to certain parameters.

Our scenario based analysis algorithms and tools should be seen as a first step towards an integrated behaviour modelling and analysis framework outlined above. This would integrate scenarios with both structural and behavioural models and aid in forward engineering –refinement, code generation and test case generation, in addition to detecting faults early in the life cycle.

6 Acknowledgements

We are grateful to Prof. S. Ramesh for his discussions and feedback on the work. We thank Prof. Mathai Joseph for his guidance and encouragement throughout the project.

References

[1] R. Alur, G.J. Holzmann, and D.A. Peled. An Analyzer for Message Sequence Charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[2] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proceedings of the Tenth International Conference on Concurrency Theory, CONCUR'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[3] Purandar Bhaduri, R. Venkatesh, and Girish Palshikar. Formal techniques for analysing scenarios using message sequence charts. Internal report, TRDDC, 2001.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[5] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99: Proc. Third IFIP Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems*, March 1999.

[6] G.J. Holzmann, D.A. Peled, and M.H. Redberg. Design Tools for Requirements Engineering. *Bell Labs Technical Journal*, pages 86–95, Winter 1997.

[7] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), April 1996.

[8] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley/ACM Press, Reading, Mass., 1992.

[9] Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, and Petra van der Stappen. Model checking for managers. In *Proceedings of the 6th International SPIN Workshop on Practical Aspects of Model Checking (Toulouse, France)*, September 1999.

[10] Vladimir Levin and Doron Peled. Verification of message sequence charts via template matching. In *TAPSOFT (FASE)'97, Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 652–666, Lille, France, 1997. Springer.

[11] E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

[12] E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts (MSC'96). In *Tutorials of the First joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96)*, October 1996.