Using Message Sequence Charts for Component-Based Formal Verification*

Bernd Finkbeiner
Computer Science Department
Stanford University
Stanford, CA 94305, USA
finkbein@cs.stanford.edu

Ingolf Krüger
Department of Informatics
Technical University of Munich
80290 Munich, Germany
kruegeri@in.tum.de

ABSTRACT

Message sequence charts (MSCs) are are a popular tool to informally explain the behavioral embedding of a component in its environment. In this paper we investigate if MSCs can also serve as a specification and reasoning technique for the composition of systems from components. We identify three challenges: (1) *Semantic Duality:* MSCs express global coordination properties as well as requirements on individual components for their correct participation in an interaction pattern. We show that the two semantics do not always agree and suggest syntactic constraints that ensure the represented property can be decomposed. (2) *Completeness:* we define a decompositional proof rule based on MSCs. We show that the rule is incomplete and discuss reasons and possible improvements. (3) *Compositionality:* in component-oriented system development, the different parts of the system are designed independently of each other. We suggest a composition operator for MSC specifications of such components and outline differences to operators used for the composition of scenarios.

1. INTRODUCTION

Component-based software development shortens the design process by allowing the software engineer to use black-box components. A prerequisite for the composition of systems from components is adequate information about their interface.

Here, with the notion of interface we associate not only the signatures of the operations a component offers to its environment; although popular, this interface notion offers much too little information to be of value in a more rigorous approach to software development. Instead, we add the component's observable behaviors as part of our interface notion. Approaches at component-oriented system development, such as ROOM [32] and UML-RT [33], have a similar but more informal interface notion. Our goal is to exploit the extra information our interface notion offers during system verification in the context of pragmatic and industrially accepted engineering approaches.

It is easy to describe the signature part of a component's interface. But how to capture, represent, and systematically develop the behavioral aspects of an interface? Message Sequence Charts (MSCs) have gained wide acceptance for scenario-based specifications of component interaction behavior (see, for instance, [20, 8, 31, 6, 29]). Due to their intuitive notation MSCs have proven useful as a communication tool between customers and developers, thus helping to reduce misunderstandings from the very early development stages.

In this paper we investigate if MSCs can also serve as a specification and reasoning technique for the composition of systems from components. When used in a formal setting, MSCs could provide the link between the verification of individual components and the correctness proof for the complete system.

MSCs capture the communication or collaboration among a set of components. Typically, an MSC consists of a set of axes, each labeled with the name of a component. An axis represents part of the existence of its corresponding component. Arrows in MSCs denote communication. An arrow starts at the axis of the sender or initiator of the communication; the axis at which the head of the arrow ends designates the communication's recipient or destination. Intuitively, the order in which the arrows occur within an MSC defines sequences of interaction among the depicted components. Figure 1 shows an MSC that displays a sequence of interactions among three components in a simple communication protocol.

The information that an MSC captures includes several structural and behavioral aspects. The separate axes indicate logical or physical component distribution. The presence of an arrow between two axes indicates the existence of a communication link between the corresponding components, as well as the occurrence of interaction itself. Finally, some MSC dialects, such as [31, 22, 10], allow the developer also to indicate state changes of individual components contained in an MSC. Composite MSCs (C-MSCs) extend the MSC language with additional structure, such as

*This research was supported in part by NSF grant CCR-99-00984-001, by ARO grants DAAG55-98-1-0471 and DAAD19-01-1-0723, by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014 and by the Deutsche Forschungsgemeinschaft within the priority program "SoftSpez" (SPP 1064) under project name *InTime*.

OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems Oct. 2001 Tampa, FL, USA

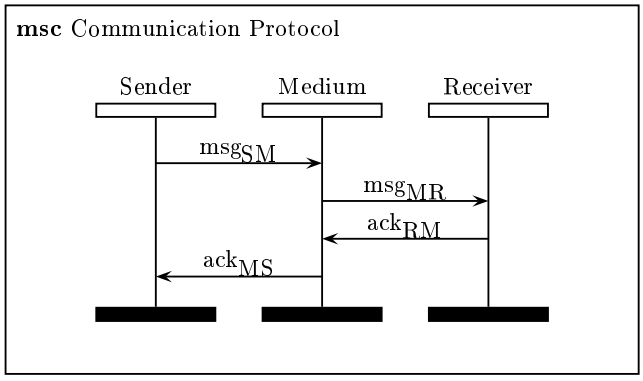

Figure 1: Basic MSC specifying a communication protocol.

loops, alternatives, or sequential composition. The example in Figure 2 shows a communication protocol with two alternative outcomes: the "Receiver" process may report success ("ack") or failure ("fail").

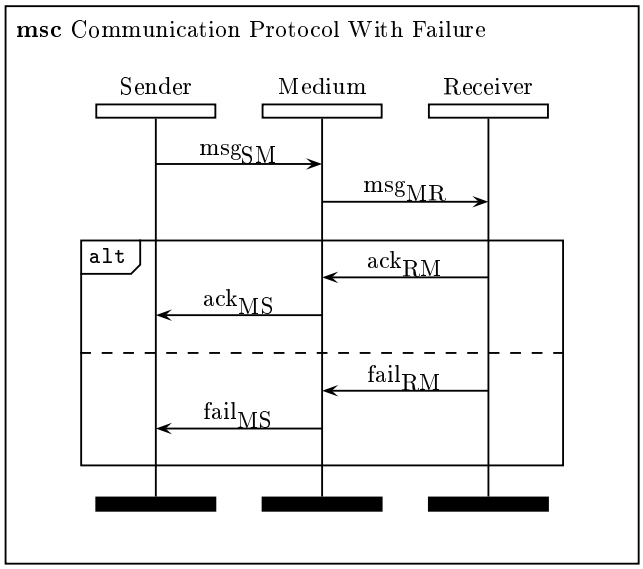

Figure 2: C-MSC specifying a communication protocol with failure.

MSCs describe the embedding of individual components into their environments, i.e., how components cooperate to achieve a certain task in a distributed system. They hide most of the details of local state changes of individual components and, instead, convey the "big picture" of the collaborations among the referenced components. This abstract, and integrated view on system behavior has resulted in the application of MSCs for use case specifications, particularly in object-oriented analysis and design, as well as for test-case specifications and simulation-run visualizations, especially in tools for telecommunication and embedded systems. In a sense, MSCs represent projections of the overall system behavior onto particular *services* or tasks of the system; automata, another popular description technique for behavioral aspects, typically represent projections of the overall system behavior onto individual components.

Of increasing importance is the use of MSCs as a description technique for complete behavior patterns, instead of for mere exemplary interaction scenarios, because this facilitates the MSC's seamless integration into an overall development process for distributed systems (cf. [24, 22], and the references contained therein). This is a particularly useful approach for the specification for component interfaces (as opposed to complete component behavior), because of the typically limited size of the corresponding interaction protocols.

An MSC describes both the global system behavior, and how each individual component should perform to establish the desired result. In this paper we explore how this duality can be used in the verification of distributed systems. Formally, MSCs here assume the role of a decompositional proof rule. They decompose a global specification into local specifications which are satisfied by the individual components. In the verification literature, this technique is known as the *assumption-commitment* paradigm (cf. [12, 7]): the environment of a component is specified only to the extent that is necessary so that the component can guarantee its correct operation; any implementation details about the environment are left unspecified at this point. Assumption-commitment reasoning shifts the burden of formal verification from the system-level down to the component-level.

While the motivation for MSC-based descriptions and assumption-commitment specifications is similar, there is a gap in the degree of formalization offered that must be closed before MSCs can be used as a formal reasoning tool. In the following sections we will discuss three challenges.

1. *Semantic Duality:* MSCs express global coordination properties as well as requirements on individual components for their correct participation in an interaction pattern. We say an MSC has the *decomposition property* if the two semantics agree. The decomposition property is necessary for MSCs used in component oriented proofs, since we infer the validity of a global property from the validity of local properties. Unfortunately, not all MSCs have the decomposition property. We suggest syntactic constraints that ensure the represented property can be decomposed.

2. *Completeness:* For MSCs that have the decomposition property, we can formulate a decompositional proof rule that reduces the proof of a global property to the verification of local component properties. The rule is incomplete: not all valid system properties can actually be proven with the rule. We discuss reasons for the incompleteness and possible improvements.

3. *Compositionality:* In component-oriented system development, the different parts of the system are designed independently of each other. Correspondingly, their MSC specifications are unlikely to be identical and we need a process to resolve differences. We suggest such a composition operator and discuss differences to operators used for the composition of scenarios.

In the following section we introduce MSCs formally, and give a simple semantics based on ω-automata; we address the three challenges in Sections 3, 4 and 5.

2. MESSAGE SEQUENCE CHARTS

MSCs have a wide spectrum of applications in the development process, ranging from analysis to implementation support. Correspondingly, many different interpretations have been proposed in the literature. An MSC language supporting requirements capture and analysis of interaction patterns requires a very liberal underlying semantics definition; it should, for instance, not exclude other possible interaction patterns too early in the development process.

In this paper, we focus on the verification of universal properties. Correspondingly, we are interested in the exclusion of undesired behaviors. In this section we describe a semantics that achieves this by identifying all possible interaction patterns: behaviors other than the ones that are explicitly depicted will be excluded.

We base our semantics on ω-automata. The ω-regular languages form a particularly useful class since it is closed under complementation and intersection, it is decidable and in fact well-supported by verification algorithms (cf. [13, 18]). We will work with a simplified definition of basic message sequence charts. Similar definitions appear in [2, 3]; a semantics for a richer dialect is given in [22].

DEFINITION 1 (MESSAGE SEQUENCE CHARTS). *A (basic)* message sequence chart *(MSC)* $M = \langle P, \mathcal{M}, E, C, O\rangle$ *is a labeled graph with the following components:*

- processes: *a finite set P of processes or components;*
- messages: *a finite set $\mathcal{M}$ of messages, we assume that the messages can be partitioned according to their sender $\mathcal{M} = \bigcup_{p\in P} S_p$ and according to their recipient $\mathcal{M} = \bigcup_{p\in P} R_p$; let $\mathcal{M}_p$ denote the union $S_p \cup R_p$;*
- events: *a finite set E of events, each process $p \in P$ has a single initial event e_p;*
- interprocess edges: *a set of directed edges connecting events, labeled by messages between processes $C \subseteq E \times \mathcal{M} \times E$; we assume that each event appears on exactly one edge;*
- intraprocess edges: *a function $O : E \to E \cup \{\bot\}$ connecting events in the order in which they are displayed. $\bot$ indicates that there is no subsequent event.*

The ω-regular languages are recognized by ω-automata. Different types of ω-automata are distinguished according to their acceptance conditions, in the following we will use the fairly simple Büchi acceptance condition on the transitions (for a survey on ω-automata see [36]).

DEFINITION 2 (BÜCHI AUTOMATON). *A Büchi automaton is a tuple $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F}\rangle$ with*

- nodes: *a finite set N of nodes,*
- input alphabet: *a finite set Σ of input symbols,*
- initial nodes: *a subset $I \subset N$,*
- transitions: *a finite set $T \subseteq N \times \Sigma \times N$ of labeled edges connecting nodes,*
- acceptance condition: *a subset $\mathcal{F} \subseteq T$.*

Acceptance of an input sequence is determined as follows.

DEFINITION 3 (ACCEPTING PATHS). *For an infinite sequence of input symbols $\sigma : s_0, s_1, s_2, \ldots$ an infinite sequence of transitions $\pi = (n_0, s_0, n_1), (n_1, s_1, n_2), \ldots$ is a* path *of $\mathcal{A}$ on σ if $n_0 \in I$. A path π is accepting if some edge in $\mathcal{F}$ occurs infinitely often in π.*

DEFINITION 4 (LANGUAGE). *The language* $\mathrm{L}(\mathcal{A})$ *of an automaton $\mathcal{A}$ is the set of all infinite sequences σ of input symbols that have an accepting path in $\mathcal{A}$.*

We represent an MSC as an automaton by using sets of "simultaneously active" events as states. There is a transition for each interprocess edge and an additional τ-transition that simulates an internal computation step. We assume zero-delay communication: the transitions respect the partial order on the intraprocess edges as well as the synchronization introduced by the interprocess edges.

DEFINITION 5 (GLOBAL SEMANTICS). *Given a basic MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$ the* global semantics *is given as the* associated global automaton $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F}\rangle$ *with*

- $N = 2^{E\cup\{\bot\}}$,
- $\Sigma = \mathcal{M} \cup \{\tau\}$,
- $I = \{\ \{e_p \mid p \in P\}\ \}$,
- *T contains a set of self-loops $\{(n, \tau, n) \mid n \in N\}$ and a set of transitions reacting to messages: $\{(n_1, s, n_2)\}$ such that*
  - *for each $e_1 \in n_1$ one of the following holds:*
    - *$e_1 \in n_2$ and there is no event e' with $(e_1, s, e') \in C$ or $(e', s, e_1) \in C$,*
    - *$O(e_1) \in n_2$ and there is an event $e' \in n_1$ with $(e_1, s, e') \in C$ or $(e', s, e_1) \in C$, and*
  - *for each $e_2 \in n_2$ one of the following holds:*
    - *$e_2 \in n_1$ and there is no event $e' \in E$ with $(e_1, s, e') \in C$ or $(e', s, e_1) \in C$*
    - *there is an event $e_1 \in n_1$ such that $e_2 = O(e_1)$ and there is an event e' with $(e_1, s, e') \in C$ or $(e', s, e_1) \in C$,*
- $\mathcal{F} = \{(\{\bot\}, \tau, \{\bot\})\}$.

Figure 3a shows the automaton associated with the MSC from Figure 1. Accepting transitions are depicted with double edges. The semantics of C-MSC constructs can be described as the corresponding transformations on the automata. Here we restrict ourselves to sequential composition, nondeterministic alternatives, and finite as well as infinite loops; these language constructs suffice for the purposes of this paper. We refer the reader to [22] for similar constructions for almost all of the MSC-96 standard [20].

DEFINITION 6 (AUTOMATA TRANSFORMATIONS). *For two Büchi automata $\mathcal{A}_1 = \langle N_1, \Sigma, I_1, T_1, \mathcal{F}_1\rangle$ and $\mathcal{A}_2 = \langle N_2, \Sigma, I_2, T_2, \mathcal{F}_2\rangle$ we define the result $\langle N', \Sigma, I', T', \mathcal{F}'\rangle$ of the following transformations:*

- *sequential composition $\mathcal{A}_1;\mathcal{A}_2$:*
  - $N' = N_1 \cup N_2$,
  - $I' = I_1$,

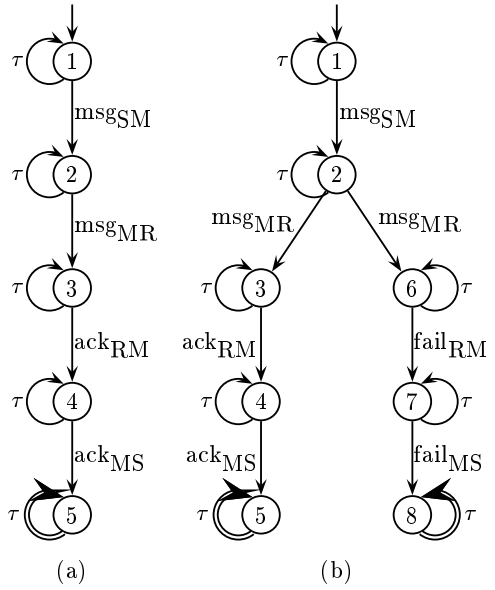

Figure 3: Automata associated with (a) the MSC from Figure 1 and (b) the C-MSC from Figure 2.

  - $T' = T_1 - \{(n_1, s, \{\bot\}) \in T_1\}$
    $\cup\ \{(n_1, s, n_2) \mid (n_1, s, \{\bot\}) \in T_1, n_2 \in I_2\}$
    $\cup\ T_2$
  - $\mathcal{F}' = \mathcal{F}_1 \cup \mathcal{F}_2$,

- *alternative* $\texttt{alt}(\mathcal{A}_1, \mathcal{A}_2)$*:*
  - $N' = N_1 \cup N_2$,
  - $I' = I_1 \cup I_2$,
  - $T' = T_1 \cup T_2$,
  - $\mathcal{F}' = \mathcal{F}_1 \cup \mathcal{F}_2$,

- *finite loop* $\texttt{loop}(\mathcal{A}_1^*)$*:*
  - $N' = N_1$,
  - $I' = I_1 \cup \{\{\bot\}\}$,
  - $T' = T_1 \cup \{(n, s, m) \mid (n, s, \{\bot\}) \in T_1, m \in I_1\}$,
  - $\mathcal{F}' = \mathcal{F}_1$,

- *infinite loop* $\texttt{loop}(\mathcal{A}_1^\omega)$*:*
  - $N' = N_1$,
  - $I' = I_1$,
  - $T' = \{(n, s, m) \mid (n, s, m) \in T_1, m \neq \{\bot\}\}$
    $\cup\ \{(n, s, m) \mid (n, s, \{\bot\}) \in T_1, m \in I_1\}$,
  - $\mathcal{F}' = \{(n, s, m) \mid (n, s, m) \in \mathcal{F}_1, m \neq \{\bot\}\}$
    $\cup\ \{(n, s, m) \mid (n, s, \{\bot\}) \in T_1, m \in I_1\}$.

As an example consider again the C-MSC from Figure 2: its associated automaton is shown in Figure 3b. Note that both the paths that stay in node 5, and the paths that stay in node 8 are accepting.

Definition 7 (Global Language). *A (finite or infinite) sequence of messages σ is accepted by an MSC M if there is an infinite sequence σ' of symbols in $\mathcal{M} \cup \{\tau\}$ such that σ' with all occurrences of τ removed is equal to σ and σ' is accepted by the automaton associated with M.*

3. CHALLENGE 1: SEMANTIC DUALITY

MSCs describe both the system behavior and how each individual component should perform to establish the desired result. A semantics reflecting this duality thus has both a global language and a *local language* for each process involved in the depicted collaboration.

In this section we study the relationship between the two languages. In a first step we distinguish the messages in whose sending or receipt a certain process is directly involved, and those that are sent and received in the process's environment. The local semantics reflects the fact that all messages that are not either sent or received by a given process are *hidden*: the process behavior is independent of hidden messages.

For each transition (n, s, m) in the global automaton with a hidden message s we add *all* transitions (n, s', m) with $s' \in (\mathcal{M} - \mathcal{M}_p) \cup \{\{\tau\}\}$: from the process's point of view it is indistinguishable if it was message s that was sent, or some other hidden message, or even no message at all.

Definition 8 (Local semantics). *For an MSC with processes P and global automaton $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F}\rangle$, the* local semantics *for a process $p \in P$ is given as the associated* local automaton $\mathcal{A}_p = \langle N, \Sigma, I, T', \mathcal{F}'\rangle$ *with*

- $T' = \{(n, s, n') \mid (n, s, n') \in T$ *and* $s \in \mathcal{M}_p \cup \{\tau\}\}$
  $\cup\ \{(n, s', n') \mid (n, s, n') \in T$ *and*
  $s \in \Sigma - \mathcal{M}_p - \{\tau\}$ *and* $s' \in \Sigma - \mathcal{M}_p\}$
- $\mathcal{F}' = \{(n, s, n') \mid (n, s, n') \in \mathcal{F}$ *and* $s \in \mathcal{M}_p \cup \{\tau\}\}$
  $\cup\ \{(n, s', n') \mid (n, s, n') \in \mathcal{F}$ *and*
  $s \in \Sigma - \mathcal{M}_p - \{\tau\}$ *and* $s' \in \Sigma - \mathcal{M}_p\}$

In component-oriented proofs, we infer the validity of a global property from the validity of local properties. Hence, we require that the global and local semantics are in agreement. However, not all MSCs have this property.

More formally, we say that an MSC has the *decomposition property* if the following equation holds for the global automaton $\mathcal{A}$, processes P and the local automata $\mathcal{A}_p$, $p \in P$:

$$\bigcap_{p \in P} \mathrm{L}(\mathcal{A}_p) = \mathrm{L}(\mathcal{A})$$

Figure 4 shows an MSC that does not have the decomposition property: consider an implementation in which process "A" first sends message "A1" and process "C" then sends message "C2": this interaction is not allowed by the global semantics. It is, however, accepted by all local automata.

Since equivalence between Büchi automata can be checked with standard verification techniques (cf. [13]), a practical solution is to check the decomposition property whenever the MSC is intended to be used in a decompositional proof.

An alternative solution is to restrict the MSC syntax so that the decomposition property is guaranteed. *Causality* is such a restriction. Consider again the example in Figure 4. There is an implicit causal relationship between messages "A1" and "C1," and "A2" and "C2," respectively. If the causalities were made explicit (for example with an extra message between process "A" and and process "C" in one of the alternatives), the decomposition property would hold.

We now give a syntactic characterization of a class of *causal* MSCs. We introduce a few auxiliary notions: the

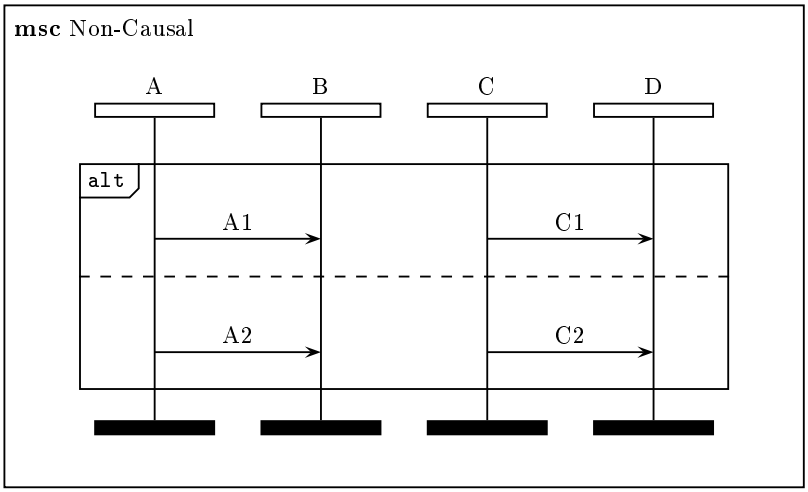

Figure 4: Non-causal MSC.

initial events $\text{init}(M)$ of an MSC M are those events that do not causally depend on any other event in M; dually, the *terminal* events $\text{term}(M)$ are those events that do not cause any other events in M. These sets of events serve as the basis for determining whether all message sequences expressed by an MSC are causally connected. Moreover, our aim is to distinguish clearly between different alternatives within a C-MSC by considering only the first message occurring within such an alternative; therefore, we also introduce a formal characterization for the set of first messages exchanged between two processes of an MSC.

We start by defining a causal order for the messages depicted in an MSC M. This serves as the basis for defining the sets $\text{init}(M)$ and $\text{term}(M)$, below.

Definition 9 (Causal Ancestor). *Let an MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$ be given. We define an order $\leq \subseteq E \times E$ on M's events as follows. Let $e, f \in E$, then*

$$e \leq f \equiv (e = f) \vee (\exists m \in \mathcal{M} : (e, m, f) \in C) \vee (f = O(e))$$

If we have $e \leq f$, we call e direct causal ancestor *of f. By $\leq^*$ we denote the reflexive, transitive closure of $\leq$. If we have $e \leq^* f$, we call e* causal ancestor *of f.*

Thus, $e \in E$ is a direct causal ancestor of $f \in E$, if either e and f coincide, or e and f are the send and corresponding receive event of the same message transmission, or e occurs immediately before f on the axis of the same process in the corresponding MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$. The causal order $\leq^*$ captures indirect causal dependencies. This allows us to define initial and terminal events by structural induction on the MSC syntax.

Definition 10 (Initial and terminal events). *For a basic MSC $M = \langle P, \mathcal{M}, E, C, O\rangle$ and its associated causal order $\leq^*$ we call an event $e \in E$* initial, *if $\forall f \in E : e \leq^* f$ holds; similarly, we call e* terminal, *if we have $\forall f \in E : f \leq^* e$. If e is M's initial event, we set $\text{init}(M) = \{e\}$; if M has no initial event we define $\text{init}(M) = \emptyset$. Similarly, we set $\text{term}(M) = \{e\}$ if e is M's terminal event, and $\text{term}(M) = \emptyset$ if no terminal event exists in M.*

For a C-MSC M the initial and terminal events are given as follows:

- $\text{init}(M_1;M_2) = \text{init}(M_1)$;
  $\text{term}(M_1;M_2) = \text{term}(M_2)$;
- $\text{init}(\texttt{alt}(M_1, M_2)) = \text{init}(M_1) \cup \text{init}(M_2)$;
  $\text{term}(\texttt{alt}(M_1, M_2)) = \text{term}(M_1) \cup \text{term}(M_2)$;
- $\text{init}(\texttt{loop}(M_1^*);M_2) = \text{init}(M_1) \cup \text{init}(M_2)$;
  $\text{term}(M_1;\texttt{loop}(M_2^*)) = \text{term}(M_1) \cup \text{term}(M_2)$;
- $\text{init}(\texttt{loop}(M_1^*)) = \text{init}(M_1)$;
  $\text{term}(\texttt{loop}(M_1^*)) = \text{term}(M_1)$
- $\text{init}(\texttt{loop}(M_1^\omega)) = \text{init}(M_1)$;
  $\text{term}(\texttt{loop}(M_1^\omega)) = \emptyset$.

In the definition of causal MSCs we will also constrain what messages may occur as a first message between two processes. We denote the set of first messages between process p and process q in the MSC M as $\text{fm}(M, p, q)$. Formally, let $\text{edges}(p, q)$ denote the set of interprocess edges between two processes p and q in a basic MSC:

$$\text{edges}(p, q) = \{(e_1, s, e_2) \in C, e_1, e_2 \in E_p \cup E_q\}.$$

The set of first messages is then defined as follows.

Definition 11 (First messages). *For a basic MSC M and two processes p, q with no interprocess edges between p and q, $\text{edges}(p, q) = \emptyset$, the set of first messages is empty: $\text{fm}(M, p, q) = \emptyset$. For non-empty $\text{edges}(p, q)$, we call the edge $(e_1, s, e_2) \in \text{edges}(p, q)$ where e_1 is a causal ancestor to all other send events e_1' with $(e_1', s', e_2') \in \text{edges}(p, q)$ the* first interprocess edge *and the message s the* first message *between p and q: $\text{fm}(M, p, q) = \{s\}$. For C-MSCs the first messages are the following sets:*

- $\text{fm}(M_1;M_2, p, q) = \text{fm}(M_1, p, q)$ *if* $\text{fm}(M_1, p, q) \neq \emptyset$ *and* $\text{fm}(M_2, p, q)$ *otherwise;*
- $\text{fm}(\texttt{alt}(M_1, M_2), p, q) = \text{fm}(M_1, p, q) \cup \text{fm}(M_1, p, q)$
- $\text{fm}(\texttt{loop}(M_1^*);M_2, p, q) = \text{fm}(M_1, p, q) \cup \text{fm}(M_1, p, q)$

- $\mathrm{fm}(\mathtt{loop}(M_1^*), p, q) = \mathrm{fm}(M_1, p, q)$
- $\mathrm{fm}(\mathtt{loop}(M_1^\omega), p, q) = \mathrm{fm}(M_1, p, q)$

Intuitively, each process in a causal MSC should always be able to infer which branch of the MSC is currently executed. This is ensured with the following syntactic constraints.

DEFINITION 12 (CAUSAL MSC). *An MSC M is a* causal MSC *if one of the following conditions holds.*

- *M is a basic MSC and has an initial event;*
- *M is a sequential composition $M = M_1;M_2$, M_1 has a terminal event e_1, M_2 has an initial event e_2 and e_1 and e_2 belong to the same process;*
- *M is an alternative between two causal MSCs, $M = \mathtt{alt}(M_1, M_2)$, and for all processes p and q, $\mathrm{fm}(M_1, p, q) \cap \mathrm{fm}(M_2, p, q) = \emptyset$*
- *M is a finite loop $\mathtt{loop}(M_1^*)$ or an infinite loop $\mathtt{loop}(M_1^\omega)$ of a causal MSC M_1.*

The MSC in Figure 4 is not causal, because the send-events for messages "A1" and "C1" do not have a common causal ancestor. In fact, the MSC would remain non-causal if were to remove the second alternative, even though the decomposition property holds for the resulting MSC. Causality is hence a sufficient but not necessary condition for the decomposition property.

Related work. The difficulty in mapping global properties to responsibilities of individual components has been considered in the literature (cf. [25, 26, 1] among others), sometimes under the keyword "nonlocal choice." Besides syntactic constraints as done for causal MSCs here, the problem can also be solved by partial or total distribution of an automaton representing the global property to all or part of the component implementation [19]; this ensures that all components synchronize their actions via the global automaton. This comes at the cost of increasing the complexity of the individual components considerably.

4. CHALLENGE 2: COMPLETENESS

In formal verification, we prove that a system satisfies its specification. If the system is the composition of a set of components $S = \{C_p \mid p \in P\}$ and the specification is given as an MSC M, verifying $S \models M$ corresponds to checking the language inclusion

$$\bigcap_{p \in P} \mathrm{L}(C_p) \subseteq \mathrm{L}(\mathcal{A})$$

where $\mathrm{L}(C_p)$ is the language accepted by the component implementing process p and $\mathcal{A}$ is the global automaton associated with M.

Analysis techniques for this problem are computationally expensive; the complexity of model checking [9], for instance, is exponential in n. It has therefore long been recognized that verification must be based on the decomposition of the system into its components.

We now discuss a decompositional proof rule for MSCs. Following the *assumption-commitment* paradigm, such a rule supplies two automata for each component: the *assumption* on the component's environment, represented by

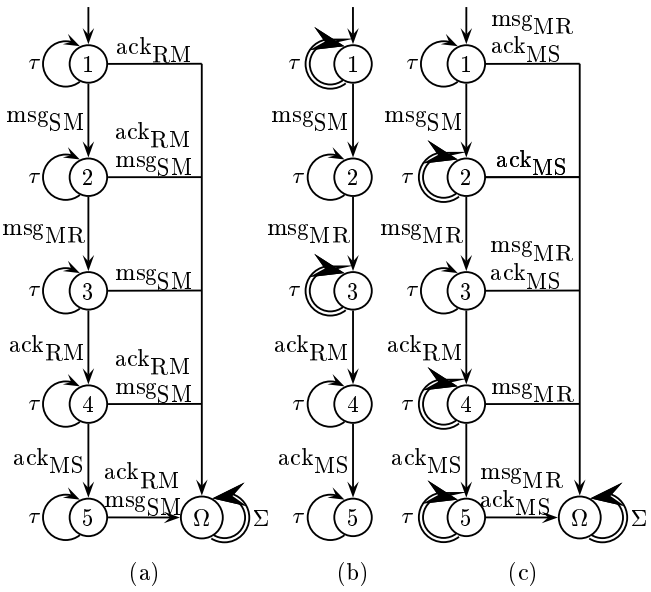

Figure 5: (a) environment-safety automaton, (b) environment-liveness automaton, (c) environment automaton for the "Medium" process from Figure 1.

an environment automaton $\mathcal{E}_p$, and the *commitment*, represented by the associated local automaton from the previous section. To prove $S \models M$ for a system $S = \{C_p \mid p \in P\}$ and an MSC M with global automaton $\mathcal{A}$ we find a second MSC M' that has the decomposition property. Let $\mathcal{A}'$ be the global automaton associated with M', and $\mathcal{A}'_p$ and $\mathcal{E}'_p$, the local automata and environment automata, respectively, for processes $p \in P$. The following rule reduces the global property to local proof obligations for each component:

DECOMPOSITIONAL PROOF RULE

if	(1) for all $p \in P$,	$\mathrm{L}(\mathcal{E}_p) \cap \mathrm{L}(C_p)$	$\subseteq$	$\mathrm{L}(\mathcal{A}'_p)$
and	(2)	$\mathrm{L}(\mathcal{A}')$	$\subseteq$	$\mathrm{L}(\mathcal{A})$
then		$\bigcap_{p \in P} \mathrm{L}(C_p)$	$\subseteq$	$\mathrm{L}(\mathcal{A})$

Since the component can *rely* on the environment to cooperate, we can exclude behaviors from the environment automaton in which the environment either illegaly sends a message (*safety* violation) or in which the component is kept waiting for the next message infinitely (*liveness* violation). We construct two automata, the environment-safety automaton $\mathcal{S}$ that recognizes all behaviors where the environment violates safety, and the environment-liveness automaton $\mathcal{L}$, that recognizes all behaviors where the environment violates liveness. Behaviors accepted by either automaton need not be considered in the verification of the component.

Safety violations can be recognized by considering finite prefixes of input sequences. Let $\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F} \rangle$ be the global automaton associated with an MSC M. The set of all finite prefixes of sequences in $\mathrm{L}(\mathcal{A})$, the *prefix language* of M, is accepted by the automaton $\langle N, \Sigma, I, T, T \rangle$. Because of the trivial acceptance condition it is possible to construct a deterministic Büchi automaton $\mathcal{P}_\mathcal{A}$ that accepts the prefix language (cf. [36]).

Definition 13 (Environment-safety). *Let the automaton $\mathcal{P}_\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F} \rangle$ be a deterministic Büchi automaton that accepts the prefix language of an MSC. The* environment-safety automaton *for process p is the automaton $\mathcal{S}_p = \langle N', \Sigma, I, T', \mathcal{F}' \rangle$ with*

- $N' = N \cup \{\Omega\}$
- $T' = T \cup \{(n, s, \Omega) \mid s \in \mathcal{M} - S_p \text{ and } \nexists n' \in N \,.\, (n, s, n') \in T\} \cup \{(\Omega, s, \Omega) \mid s \in \Sigma\}$
- $\mathcal{F}' = \{(\Omega, s, \Omega) \mid s \in \Sigma\}$

Figure 5a shows the environment-safety automaton for the "Medium" process from the communication protocol example. Note that the accepting paths stay in node Ω; a transition to Ω occurs whenever the environment illegally sends a message.

Figure 5b shows the environment-liveness automaton for the "Medium" process. The accepting paths stay in nodes 1 and 3: in node 1, the "Medium" process can count on the "Sender" process to eventually send a message; in node 3, the "Medium" process awaits the acknowledgement from the "Receiver" process.

Definition 14 (Environment-liveness). *Let the automaton $\mathcal{P}_\mathcal{A} = \langle N, \Sigma, I, T, \mathcal{F} \rangle$ be a deterministic Büchi automaton that accepts the prefix language of an MSC. The* environment-liveness automaton *for process p is the automaton $\mathcal{L}_p = \langle N, \Sigma, I, T, \mathcal{F}' \rangle$ with*

$$\mathcal{F}' = \{(n, \tau, n) \mid \nexists n' \in N, s \in S_p \,.\, (n, s, n') \in T \text{ and } \exists n' \in N, s \in (\mathcal{M} - S_p) \,.\, (n, s, n') \in T\}$$

Finally, the environment automaton $\mathcal{E}_p$ contains all behaviors in which the environment commits neither a safety nor a liveness violation. In the example, this combination results in the environment automaton shown in Figure 5c.

Definition 15 (Environment automaton). *Let $\mathcal{S}_p$ be the environment-safety automaton and $\mathcal{L}_p$ the environment-liveness automaton for a process p. The* environment automaton *$\mathcal{E}_p$ accepts the language*

$$\mathrm{L}(\mathcal{E}_p) = \overline{\mathrm{L}(\mathcal{S}_p) \cup \mathrm{L}(\mathcal{L}_p)}$$

We can now analyze the completeness of our rule. The decompositional proof rule is *complete* if for *any* system S and MSC M with $S \models M$, there is an MSC M' such that the conditions of the rule hold. So far, we have made no assumptions about the components allowed in the system composition. In this generality, the decompositional proof rule is clearly incomplete.

In Figure 6, the specification is satisfied if the two processes "A" and "B" exchange exactly one message. Now consider the following implementation: component "A" chooses at each point nondeterministically whether or not to send its message to "B" (unless it receives a message from "B" first). "B," on the other hand, applies a timeout-mechanism that guarantees that eventually a message is sent. There is no MSC M' such that the conditions of the decompositional proof rule hold: none of the two alternatives in Figure 6 can be removed since either message may occur.

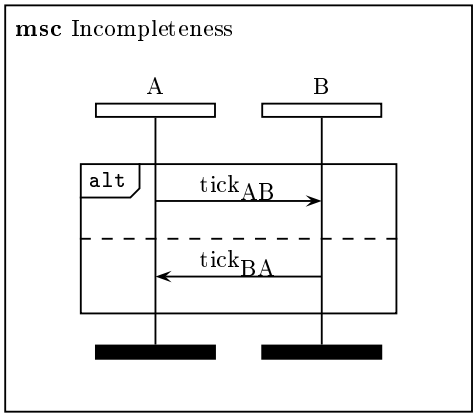

Figure 6: Incompleteness example.

The incompleteness of the proof rule stems from the difference in expressiveness of the MSCs we have considered so far, and the components implementing individual processes. This leaves us with two options for achieving completeness: one is to *restrict* our attention to a smaller class of systems, the other is to *add* to the expressiveness of MSCs.

Examples for restrictions are *regularity*: the language accepted by each component is ω-regular, *reactivity*: every component exchanges infinitely many messages with its environment, and *I/O directedness*: a component has control only over its output messages. If such restrictions are inadequate, it is certainly possible to make MSCs more expressive, for example with an explicit assignment of progress responsibilities: the property in Figure 6 could be proven for the described implementation by indicating in Figure 6 that process "B" is responsible for the progress beyond the interprocess edge (resulting in an appropriately modified environment automaton for process "A").

In practice, components often accept a non-regular language. Suggestions in the literature to extend MSCs to non-regular languages include extensions with data states [6, 22], as well as performance and real-time constraints (cf. [16, 17, 31]). However, any extension to the MSC's expressiveness comes at the price of increased complexity: many extended MSCs are undecidable. Care is also required to avoid syntactic clutter and to maintain the MSCs' intuitive appearance.

Related work. Decompositional proofs have been studied for a long time, starting with the *rely-guarantee* formalism [21] and proofs for networks of processes [27]. Since then, many assumption-commitment rules have been proposed, see [12] for an overview and [30] for a discussion of their completeness. Our decomposition of MSC properties into an assumption-commitment specification for individual components is similar to the one in [7]; the semantic framework used there includes the reactivity and I/O directedness requirements mentioned above. We are not aware of any work that formally analyzes the completeness of MSC languages. A closely related topic, however, is the "reverse engineering" of MSCs from systems; this is studied in [28].

5. CHALLENGE 3: COMPOSITIONALITY

In the preceding sections we have addressed the properties expressed by individual MSCs with respect to a certain system under consideration. Now we turn our attention to the composition of specifications from several, possibly *non-orthogonal* MSCs. Intuitively, two MSCs are non-orthogonal, if one contains a segment of an interaction pattern depicted by the other. We deal with this problem from two perspectives. First, we study the composition of "off-the-shelf" components specifically in the context of verification; here, the basic problem is to relate the already fixed interface specifications of already existing components. Second, we consider MSC composition in the more general context of scenario specifications.

In component-oriented system development, the different parts of a system are designed independently of each other; components may be retrieved from a database that was put together long before the system's conception. It is therefore unrealistic to expect that the MSCs documenting the different components will agree, and we need a process to resolve any differences.

In our communication protocol example, assume the "Sender" component is described by the simple MSC from Figure 1, and the "Medium" component has the richer functionality depicted in Figure 2. Which MSC describes the embedding of the composition of the two components in the system? Or should this combination of components be rejected altogether?

In assumption-commitment reasoning, the environment of a component is expected to show *at most* the behavior allowed by the environment assumption (cf. [12]). Hence, the combination of "Sender" and "Medium" component in our communication protocol example would be rejected, since the "Medium" component may send a "fail" message which is not allowed in the MSC of the "Sender" component. Semantically, this analysis corresponds to a *pessimistic* view of the environment [11]: The combination of two components is rejected because an environment exists that would violate the specification of one of the components. In this example, there is an implementation of the (so far not analyzed) "Receiver" process that corresponds to the MSC of the "Medium" component, but that would cause a violation of the MSC of the "Sender" component. A more liberal *optimistic* view allows the combination of two components as long as an implementation for the remaining environment exists that would allow all specifications to be satisfied.

The optimistic point of view can be implemented in a process for the composition of component MSCs. Given a system S and an MSC M_A specifying the behavior of a subset of the components $A \subseteq S$, and an MSC M_B specifying the components $B \subseteq S$, we construct an MSC $M_{A \cup B}$ specifying $A \cup B$. In this chart only those behaviors of $S - (A \cup B)$ are allowed that do not cause the components in A to violate environment assumptions of the components in B, or, vice versa, cause the components in B to violate environment assumptions of components in A.

There are automata-based solutions for optimistic composition (cf. [11]). It would be desirable to have purely syntactic combination operations for MSCs that implement this semantic construction and combinations for more expressive MSC languages. This would constitute a first step towards a thorough, seamless usage of MSCs as a specification and verification aid in the context of component composition.

We now turn to questions of MSC composition in a more general setting including analysis and design in addition to verification. As we have argued in the preceding sections we need a very strict MSC interpretation for promising MSC application in the verification task. The well-established usage of MSCs for capturing *scenarios*, on the other hand, is an example of a very liberal MSC interpretation. A scenario captures one possible segment of an overall system execution, projected onto the components referenced in the MSC. Because scenarios describe usually very specific instances of behavior, a corresponding composition operator must be very permissive; it cannot exclude alternative or even interleaved behaviors prematurely. [22] contains a composition operator, called "join", which matches the messages shared by the two operand MSCs; the resulting MSC's semantics contains only behaviors where this match is possible. This form of composition explicitly supports the combination of overlapping specifications; it is easily transferred into the semantic framework we have established in this paper.

Related work. The distinction between "optimistic" and "pessimistic" compositionality has been made in the verification literature, for example in *lazy compositional verification* [34] and, more recently, within the formalism of *interface automata* [11]. In the MSC literature certain dialects can be seen as closer to the pessimistic or optimistic point of view. [22] discusses MSC interpretations in the range from scenarios to exact component behavior; the latter excludes behaviors other than the explicitly depicted ones.

6. CONCLUSIONS AND OUTLOOK

Message sequence charts have been used for quite some time to *informally* describe the embedding of a component in its environment. In this paper we have formulated criteria the MSC language should satisfy so that the embedding furthermore qualifies as a *formal proof:* if this is achieved, then the correctness of the system is guaranteed once each individual component is verified.

Certain compromises must be made when choosing an MSC language. The simple MSCs described in Section 2 are attractive because their semantics is well-supported by verification methods; however, they do not provide a complete proof technique as discussed in Section 4. More expressive languages, such as the ones mentioned at the end of Section 4, on the other hand, are hard to analyze or even undecidable. For a given system and component model, a good compromise would be to first select a language on the basis of its completeness and then identify fragments according to their expressiveness.

Using MSCs as a verification tool as suggested in this paper should feel natural to designers familiar with MSC-based scenario descriptions. There is also a close resemblance to verification tools such as generalized verification diagrams [4, 5]: verification diagrams are similarly based on ω-automata, and they can also be used for component-oriented proofs [14]. MSCs and verification diagrams work, however, on different levels: verification diagrams are complete proofs of a certain property. MSCs, on the other hand, do not constitute complete proofs by themselves, since they are constructed independently of implementation details. Instead, they integrate the verification of individual properties in the correctness proof of the overall system.

Compositionality may be the hardest remaining challenge for a practical application of MSCs in component-based verification. In this paper we have addressed the composition of MSC specifications referencing concrete components of the system under consideration. Often, however, similar interaction patterns occur over and over again within the same system among different sets of components, and also within other systems. We can also identify and describe these interaction patterns by means of MSCs: we only have to interpret the axes of the MSCs more liberally. By parameterizing MSCs with respect to their axis labelings, i.e., the components they reference, we obtain a flexible language for such recurring interaction patterns. Instead of a single concrete component of a particular system under consideration, an axis then represents the "role" of a participant in the interaction pattern. The resulting MSCs describe interaction patterns abstractly, without references to concrete participants of a collaboration. We also speak of "connectors", when referencing abstract interaction protocols (cf. also [37, 33, 35, 7, 6]).

To use MSCs successfully in describing connectors (cf. [6, 7, 23, 15]) we need a way to relate abstract connectors and concrete component interfaces. One way to do so is to instantiate the roles in connectors by concrete components, whose interfaces are also specified by MSCs; in a second step we then have to match the behaviors allowed by the connector with those of the instantiating components.

Exploiting the information contained in a connector during component-oriented verification displays much potential for reducing the overall verification complexity, and is a promising area of future research.

Acknowledgments

The authors are grateful to Manfred Broy, César Sánchez, Bernhard Schätz and Henny Sipma for helpful discussions and comments on causality and MSCs in general.

7. REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proceedings of 22nd International Conference on Software Engineering*, pages 304–313, 2000.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *28th International Colloquium on Automata, Languages and Programming*, LNCS. Springer-Verlag, 2001.

[3] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software — Concepts and Tools*, 17:70 – 77, 1996.

[4] A. Browne, Z. Manna, and H. B. Sipma. Generalized temporal verification diagrams. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 484–498. Springer-Verlag, 1995.

[5] A. Browne, Z. Manna, and H. B. Sipma. Hierarchical verification using verification diagrams. In 2^{nd} *Asian Computing Science Conf.*, volume 1179 of *LNCS*, pages 276–286. Springer-Verlag, Dec. 1996.

[6] M. Broy, C. Hofmann, I. Krüger, and M. Schmidt. A graphical description technique for communication in software architectures. Technical Report TUM-I9705, Technische Universität München, 1997.

[7] M. Broy and I. Krüger. Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts. In J. Staples, M. G. Hinchey, and S. Liu, editors, *Formal Engineering Methods (ICFEM'98)*, pages 2–15. IEEE Computer Society, 1998.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley, 1996.

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.

[11] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[12] W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference. COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[13] B. Finkbeiner. Language containment checking using nondeterministic bdds. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*. Springer-Verlag, 2001.

[14] B. Finkbeiner, Z. Manna, and H. B. Sipma. Deductive verification of modular systems. In de Roever et al. [12], pages 239–275.

[15] J. Grabowski, P. Graubmann, and E. Rudolph. HyperMSCs with Connectors for Advanced Visual System Modelling and Testing. In *SDL Forum 2001*, pages 129–147. Springer, 2001.

[16] R. Grosu, I. Krüger, and T. Stauner. Hybrid sequence charts. Technical Report TUM-I9914, Technische Univerität München, 1999.

[17] R. Grosu, I. Krüger, and T. Stauner. Requirements Specification of an Automotive System with Hybrid Sequence Charts. In *WORDS'99F, Fifth International Workshop on Object-oriented Real-time Dependable Systems*. IEEE, 1999.

[18] R. Hardin, Z. Har'El, and R. Kurshan. COSPAN. In R. Alur and T. A. Henzinger, editors, *Proc.* 8^{th} *Intl. Conference on Computer Aided Verification*, volume 1102 of *LNCS*, pages 423–427. Springer-Verlag, July 1996.

[19] D. Harel and H. Kugler. Synthesizing object systems from lcs specifications, 1999. (submitted).

[20] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.

[21] C. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[22] I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[23] I. Krüger. Notational and Methodical Issues in Forward Engineering with MSCs. In T. Systä, editor, *Proceedings of OOPSLA 2000 Workshop:*

Scenario-based round trip engineering. Tampere University of Technology, Software Systems Laboratory, Report 20, 2000.

[24] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *DIPES'98*. Kluwer, 1999.

[25] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, (5):473–509, 1995.

[26] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering*. PhD thesis, Universität Bern, 1995.

[27] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[28] A. Muscholl and D. Peled. From finite state communication protocols to high level message sequence charts. In *28th Int. Col. on Automata Languages and Programming (ICALP'2001)*, volume 2076 of *LNCS*, pages 720–731. Springer-Verlag, 2001.

[29] R. Nahm. Designing and documenting componentware with message sequence charts. In T. Jell, editor, *Component-based Software Engineering*, pages 111–116. Cambridge University Press, 1998.

[30] K. S. Namjoshi and R. J. Trefler. On the completeness of compositional reasoning. In *12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 139–153. Springer-Verlag, 2000.

[31] Unified modeling language, version 1.1. Rational Software Corporation, 1997.

[32] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[33] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. `http://www.objectime.com/otl/technical`, April 1998.

[34] N. Shankar. Lazy compositional verification. In de Roever et al. [12].

[35] M. Shaw and D. Garlan. Software architectures. perspectives on an emerging discipline, 1996.

[36] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers (North-Holland), 1990.

[37] A. C. Wills and D. D'Souza. *Objects, Components, and Frameworks with UML– The Catalysis Approach*. Addison Wesley, 1998.