Towards the Methodical Usage of Message Sequence Charts

Ingolf Krüger*

Institut für Informatik
Technische Universität München
D-80290 München, Germany
http://www4.in.tum.de/~kruegeri/

Abstract — Message Sequence Charts (MSCs) and similar notations for component interaction have gained wide acceptance for scenario-based specifications of component behavior. However, most development methods use MSCs for exemplary interaction descriptions only, and leave many questions concerning a more seamless, methodical integration of MSCs into an overall software development process unanswered. Examples of such questions are: how complete is the information captured in the MSCs with respect to the possible component interactions? How do multiple MSCs relate? How to transit from exemplary to complete interaction descriptions? In this paper we address these questions from several different viewpoints. We define a formal semantics for individual MSC documents that captures information on component interaction, as well as on component state. On this semantics definition we base our discussion of a methodical application of MSCs. In particular, we describe how to derive individual component descriptions – in the form of assumption/commitment and state machine specifications – from MSCs, and discuss notions for MSC composition and refinement. Furthermore, we describe approaches at relating MSCs with a given system specification, and mention occurrence conditions for the interaction patterns that they contain.

*This work was supported with funds of the Deutsche Forschungsgemeinschaft under the Leibniz program, by Siemens Nixdorf within project SysLab, and by the Bayerische Forschungsstiftung.

1 Introduction

MSCs[1] have gained wide acceptance for scenario-based specifications of component behavior (see, for instance, [IT96, BMR+96, Rat97, BHKS97]) for all classes of systems, including both business and embedded systems. Due to their intuitive notation MSCs have proven useful as a communication tool between customers and developers, thus helping to reduce misunderstandings in early development stages; their predominant use today is in the requirements capture phase of the software development process.

However, up to now, most development methods and processes do not assign a precise meaning to this graphical description technique. Instead, they consider MSCs only a means for documenting interaction scenarios, and – if at all – establish only a vague relationship between MSCs and other description techniques, such as state-based component specifications. A lot of information that is illustrated using MSCs is not systematically exploited when transiting from requirements capture to design, let alone the transitions between the development activities occurring even later. This means that, in general, there exists no seamless integration of MSCs into an overall software development process.

Precise definitions of the MSC's syntax or semantics, and even development guidelines and methods for distributed systems leave some or all of the following important questions unanswered: are there other components in the system besides the ones depicted in an MSC? May a component involve in other interactions with components within and outside the scope of the MSC? What triggers the occurrence of the interaction sequence depicted in an MSC? Must or may the depicted interaction sequence occur in all or any executions of the system under development? How do multiple MSCs relate? How does an MSC relate with structural or state-based system specifications? How to transit from an exemplary to a complete interaction description? What are useful and effective refinement notions for MSCs? How to use the information contained in an MSC for quality assurance and validation?

In this paper, we discuss some of the methodical issues that underly solutions to the questions we have raised above. In Section 2 we present a formal basis for the precise definition of the semantics of individual MSCs. As the semantic domain for this definition we use streams over channel and state valuations. This gives us a handle on directly transforming an MSC into property specifications for the components it references, which we mention in Section 3.1. In Section 3.2 we describe a translation scheme that allows us to derive state machine representations for individual components directly from a set of given MSCs. Notions of refinement and composition are the topic of Section 3.3. Section 4 contains our conclusions.

[1] In the remainder of this document we subsume under the term "MSC" graphical description techniques that use axes to represent component existence and arrows to denote communication, not only the standard notation from [IT96, IT98].

2 Semantics of Individual MSCs

Over the past few years various different syntaxes and semantics for MSC dialects were proposed both by researchers from academia and industry, and by standardization committees [IT96, IT98, MR94, MR96, Rat97, BHKS97]. In particular, the MSC-96 standard [IT96], recommended by ITU, and the UML's Sequence Diagrams [Rat97] have received considerable attention in academia and industry. While MSC-96 has its roots in the specification and documentation of asynchronous telecommunication systems, Sequence Diagrams reflect the stronger focus on control- and method-call-oriented specifications that occur in most object-oriented development methods.

We aim at a methodical integration of MSCs into the overall, iterative development process – consisting of analysis, specification, design, implementation, and validation – that includes a thorough understanding of the semantics of individual MSCs and their integration with state-based system descriptions. Therefore, in Section 2.1, we define a mathematical model that serves as the basis for our semantics definition for individual MSCs. Our version of the MSC syntax and semantics (cf. Section 2.2) covers most of the features of MSC-96, including individual messages, sequential composition, inline expressions, and references. Moreover, we add meaning to conditions in MSCs (in the form of "guarded MSCs"), which allows us to add information on component state to interaction sequences.

2.1 System Model and Mathematical Preliminaries

The system class we target at is that of open, distributed reactive systems with static structure. Here, we define a simple, yet precise mathematical model for the description of such systems. Along the way we introduce the notation and concepts that we need to describe the model.

Structurally, a system consists of a set P of components or processes, and a set C of directed channels. Channels connect components that communicate with one another; they also connect components with the environment. With every $p \in P$ we associate a unique set of states, i.e. a component state space, S_p. We define the state space of the system as $S \stackrel{\mathsf{def}}{=} \Pi_{p \in P} S_p$; we do not distinguish between data and control states. For simplicity we represent messages by the set M of message identifiers.

Now we turn to the dynamic aspects of the system model. We assume that the system components communicate between one another and with the environment by exchanging messages over channels. We assume further that a discrete global clock drives the system. We model this clock by the set $\mathbb{N}$ of natural numbers. Intuitively, at time $t \in \mathbb{N}$ every component determines its output based on the messages it has received until time $t-1$, and on its current state. It then writes the output to the corresponding output channels and changes state. The delay of at least one time unit models the processing time between an input and the output that it triggers.

Formally, with every channel $c \in C$ we associate the histories obtained from collecting all messages sent along c in the order of their occurrence. Our basic assumption here is that communication happens asynchronously: the sender of a message does not have to wait for the latter's receipt by the destination component. This allows us to model channel histories by means of streams. Streams are finite and infinite sequences of messages. By X^* and X^∞ we denote the set of finite and infinite sequences over set X, respectively. $X^\omega \stackrel{\mathsf{def}}{=} X^* \cup X^\infty$ denotes the set of streams over set X. Note that we may identify X^* and X^∞ with $\bigcup_{i\in\mathbb{N}}([0..i] \to X)$ and $\mathbb{N} \to X$, respectively. This allows us, for $x \in X^\omega$ and $n \in \mathbb{N}$, to write $x.n$ for the n-th element of stream x. For $x \in X^*$ we denote the length of x by $|x|$, it is equal to some natural number. For $x \in X^\infty$, $|x|$ yields ∞. Furthermore, for $x \in X^\omega$ and $n \in \mathbb{N}$ we define $x{\downarrow}n$ to be the prefix of x with length n. By $x{\uparrow}n$ we denote the stream obtained from x by removing the first n elements. We write the concatenation of two streams $x_0, x_1 \in X^\omega$ as $x_0 \frown x_1$. If $|x_0| = \infty$ then $x_0 \frown x_1$ equals x_0.

We lift all the operators introduced above to tuples and sets of streams by interpreting them in a pointwise and elementwise fashion, respectively, in this setting.

Timed streams are infinite streams (time does not halt) whose elements at position $t \in \mathbb{N}$ represent the messages transmitted at time t. We define $\tilde{C} \stackrel{\mathsf{def}}{=} C \to \mathcal{P}(M)$ and obtain $\tilde{C}^\infty$ as an infinite valuation of all channels[2]. With timed streams over message sets we have a model for the communication among components over time. Similarly we can define a succession of system states over time as an element of set S^∞.

With these preliminaries in place, we can now define the semantics of a system with channel set C, state space S, and message set M as a subset of $\mathcal{P}((\tilde{C} \times S)^\infty)$. Any element (φ_1, φ_2) of a system's semantics consists of a valuation of the system's channels ($\varphi_1 \in \tilde{C}^\infty$) and a description of the state changes of the system's components over time ($\varphi_2 \in S^\infty$). The existence of more than one element in the semantics of a system indicates nondeterminism.

In summary, our system model consists of two parts: the system's static structure and its behavior. The set of channels, the set of component states, and the set of messages determine the system's structure. The channel valuations, i.e. the occurrences of messages on channels over time, and the sequences of states determine the system's behavior.

2.2 MSC Syntax and Semantics

In this section we introduce a textual syntax for the MSC dialect that we use in this paper. We also discuss the definition of the MSC's semantics along the lines of [BHKS97, BK98, GKS]. Because our main topic is the methodical usage of MSCs we keep the semantics definition short and rather informal. A much more detailed treatment of this MSC semantics will appear in [Krü99].

For the description of the MSC's syntax we assume given three sets:

[2] By $\mathcal{P}(X)$ we denote the power set of X.

$MSCNAME$	denotes the set of MSC names
$MSCT$	denotes the set of well-formed MSC terms
$MSCDEF \subseteq MSCNAME \times MSCT$	relates MSC names with the interaction sequence they represent

MSC-96 allows us to assign names to MSCs; $MSCDEF$ captures this association. The left column of Table 1 reflects the structure of set $MSCT$: $\mathbf{empty}, \mathbf{any} \in MSCT$ are the base cases; the following lines constitute an inductive grammar definition. Here, we assume $\alpha, \beta \in MSCT$, $ch \in C$, $m \in M$, $K \in \mathcal{P}(P)$, and $X \in MSCNAME$. p_K is a string that represents a predicate over the state of the components contained in K.

Syntax	Informal Meaning
empty	*no interaction*
any	*arbitrary interaction*
$ch \triangleright m$	occurrence of message m on channel ch
$\alpha \,;\, \beta$	*sequential composition* of the interactions that α and β represent
$p_K{:}\alpha$	*guarded MSC:* predicate p_K guards the occurrence of the interaction sequence that α represents
$\alpha \mid \beta$	*alternative* between the interactions that α and β represent
$\alpha \sim \beta$	*interleaving* between the interactions that α and β represent
$\alpha\uparrow_{<*>}$	unbounded but finite *repetition* of the interactions that α represents
$\rightarrow X$	*reference* to the MSC named X

Table 1: Textual syntax and intuitive semantics for the MSC dialect used here

For reasons of brevity we have limited ourselves to a subset of the syntax from [Krü99]; there we also describe, for instance, the extension of the loop construct for infinite and bounded finite repetition, preemption, the semantics of message parameters, and parameterized MSCs.

Given the syntax of our MSC dialect we now turn to the description of its semantics. The right column of Table 1 gives the informal meaning of each element of set $MSCT$. Intuitively, we associate with a given MSC a set of channel and state valuations, i.e. a set of system behaviors according to the system model we have introduced in Section 2.1. Put another way, we interpret an MSC as a constraint at the possible behaviors of the system under consideration. More precisely, with every $\alpha \in MSCT$ and every $u \in \mathbb{N}_\infty$, where $\mathbb{N}_\infty \stackrel{\mathsf{def}}{=} \mathbb{N} \cup \{\infty\}$, we associate a set $[\![\alpha]\!]_u \in \mathcal{P}((\tilde{C} \times S)^\infty \times \mathbb{N}_\infty)$; any element of $[\![\alpha]\!]_u$ is a pair of the form $(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. The first constituent, φ, of such a pair describes an infinite system behavior. u and the pair's second constituent, t, describe the time interval within which α constrains the system's behavior. Outside the time interval specified by u and t, with $u \leq t$, the MSC α makes no statement whatsoever about the interactions and state changes that occur in the system.

To give an idea of the inductive definition of $[\![.]\!]_u$ we give the semantics of **empty**, **any**,

message occurrence, and of sequential composition[3]:

$$[\![\mathbf{empty}]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi, u) : \varphi \in (\tilde{C} \times S)^\infty\}$$

$$[\![\mathbf{any}]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : t \geq u\}$$

$$[\![ch \triangleright m]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : t = \min\{v > u : m \in \pi_1(\varphi).v.ch\}\}$$

$$\begin{aligned}[\![\alpha\,;\beta]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty \;:\; & \exists \psi_1, \psi_2, t_1, t_2 : (\varphi, t) = ((\psi_1 \downarrow t_1) \frown (\psi_2 \uparrow t_1), t_2) \\ & \wedge \;\; (\psi_1, t_1) \in [\![\alpha]\!]_u \\ & \wedge \;\; (\psi_2, t_2) \in [\![\beta]\!]_{t_1}\}\end{aligned}$$

The definitions for the other syntactic elements will appear in [Krü99]. The semantics of a guarded MSC $p_K{:}\alpha$, i.e. $[\![p_K{:}\alpha]\!]_u$, contains only such channel and state valuations whose state component at time u fulfills predicate p_K, and, from u on, is as specified by α. We use guarded MSCs to assign meaning to MSCs with conditions. Note that the definition for sequential composition above differs from MSC-96's "weak sequential composition"; the latter may result in the interleaving of its operands. However, the semantic basis we have introduced here suffices to model almost all concepts of MSC-96.

3 Methodical Application of MSCs

The precise definition of the syntax and semantics of individual MSCs is an important issue. Just as important is, however, the seamless methodical integration of MSCs into the overall software development process. In this section we consider issues that bring us closer towards this goal: we discuss the direct derivation of individual component specifications and Statechart-like models from MSCs; we also present notions of MSC composition and refinement. We base the methodical suggestions that we describe here on the semantic foundations we have introduced in Section 2.

With the three topics covered in this section we directly address several gaps that exist in today's application of MSCs within and across development phases. The derivation of Assumption/Commitment (A/C) specifications for individual components that we mention in Section 3.1 aids in the transition from requirements capture to specification. While MSCs typically capture global communication behavior, the A/C specifications we derive describe precise requirements of individual components with respect to their environment. In Section 3.2 we go a step further; we describe how to derive state-based component behavior directly from MSCs. The result of this derivation is, in a sense, an implementation of the requirements derived in Section 3.1, and, thus, can serve as the basis for transiting from specification to design. The state machines that we construct play the role of "jump-start" models of the components under design. The composition and refinement notions we describe in Section 3.3 allow us to target the transition from simple scenarios to complete behavior descriptions using MSCs.

[3] π_1 denotes the projection on the first element of a (stream) tuple.

3.1 From MSCs to A/C-Specifications

Typically, MSCs describe global system behavior in the sense that they depict patterns or sequences of interaction among multiple components. Such global system descriptions are particularly useful during requirements capture, where they often serve as operation or use case specifications. In general, however, we are less interested in the overall system than in describing, or reasoning about, individual components. Formally, we capture this derivation of an individual component's properties from an MSC specification by means of Assumption/Commitment (A/C) specifications.

In [BK98] we have described in detail how to obtain A/C-specifications from "interaction interfaces". Intuitively, interaction interfaces are predicates over infinite channel valuations (see Section 2.2) that we can directly derive from a set of given MSCs. This suggests a methodical transition from the requirements capture or analysis phase to specification: (a) fix the distribution structure of the system, (b) capture the interaction requirements of the interesting components using MSCs, (c) derive A/C component specifications from the MSCs.

An interesting application of this strategy is to assign meaning to connector specifications as they appear, for instance, in various forms in Catalysis [WD98], and ROOM [SGW94]. ROOM's "protocol classes", for instance, consist only of a static list of input and output messages that describe a certain aspect of a ROOM component's interface in the form of message names and signatures. Protocol classes do not constrain the dynamic aspects of the collaborating components. Using MSCs to specify both the static and the dynamic protocol aspects, and deriving the specifications of each component's assumptions and commitments directly from the MSCs results in a much more expressive form of protocol classes, and in a much better integration of MSCs into ROOM. Similarly, we could enhance the Interface Description Languages (IDLs) used in middleware technologies like CORBA, Java RMI, and DCOM by supplying (semi)automatically derived behavior descriptions (in the form of A/C specifications) in addition to the static method signatures in use today.

3.2 MSCs and FSMs

The interaction interfaces that we used in [BK98] to obtain A/C specifications for individual components were based on the components' message exchange only and did not take their states into account. Sometimes, however, we are not interested in these rather abstract A/C specifications that we can obtain from a given (set of) MSC(s) according to Section 3.1. Instead, we want to have the component's implementation in the form of a (finite) state machine. Of course, we expect this implementation to fulfil the corresponding A/C specification. One approach at obtaining a state-based behavior description would, therefore, be to translate the A/C specification into an automaton that displays the adequate input/output behavior. Automata with finite state and transition sets are, however, hard to construct from A/C specifications. The problem is that determining the

component's "current" control state from (a finite prefix of) its interaction behavior is, in general, impossible.

Therefore, we derive state machines directly from the given MSCs according to the translation procedure we have described in [KGSB99]. This allows us to take into account state information that the MSCs' author has provided using guards. More precisely, we assume given a set of MSCs that describe all the interaction sequences among a set of components, i.e., we make a closed world assumption with respect to the interaction sequences that occur in the system under development. We translate each guard that appears in any of the MSCs into a corresponding automaton state. We assume further that we try to obtain an automaton for exactly one of the components, say $p \in P$, occurring in the MSCs. The second input we expect is the name of the initial state for p's automaton. The procedure for obtaining that automaton consists of four successive phases (see [KGSB99] for the details):

1. **Projection:** We project each of the given MSCs onto the component p, i.e., we remove all other instance axes, as well as message arrows that neither start nor end at p.

2. **Normalization:** We normalize the projected MSCs. An MSC in our normal form consists of exactly two guards and a (possibly empty) sequence of message arrows in between. We achieve this normalization by inserting the given initial state into MSCs without guards at their start or end, and by splitting MSCs at intermediate guards.

3. **Transformation into an Automaton:** We obtain the automaton corresponding to p in two steps. First, we introduce fresh guards into the normalized MSCs such that before and after each message arrow there is a guard symbol. Second, we interpret each guard as an automaton state and draw a transition from state s_0 to state s_1 if there is an MSC with a message arrow between states s_0 and s_1. We label the transition with an input or output action for the corresponding message according to whether the arrow is an incoming or outgoing one. If there is no message arrow between s_0 and s_1 we also introduce a transition and label it by ϵ. After performing these steps we obtain a nondeterministic automaton with input/output actions for component p.

4. **Optimization:** We can run any of the optimization algorithms known from automata theory on the resulting automata; examples are the standard ϵ-elimination, determinization, and minimization algorithms. Other forms of optimizations stem from knowledge about the underlying communication paradigm [KGSB99].

The automaton resulting from application of these steps can serve as a jump-start model for component p, which can result in deriving early prototypes from MSC specifications; this paves the way for transiting from analysis to specification or to design. Note that the developer can guide the quality of the translation by adding state information in the form of guards. The translation procedure is independent from the underlying communication

paradigm as long as the one chosen for the composition of automata is the same as that assumed when drawing the MSCs. For the details of the translation procedure and for an example application we refer the reader to [KGSB99].

3.3 MSC Refinement and Composition

In the two preceding sections we have discussed transformations on MSCs that mainly address the transition from one development phase to another. Here we consider MSC refinement, which mainly helps us to increase the level of detail that occurs in a single MSC. Furthermore, we briefly describe various composition operators for MSCs, as well as occurrence conditions for MSCs.

MSC refinement

We distinguish three kinds of refinement for MSCs: *property refinement*, *message* or *interface refinement*, and *structural refinement*.

Property refinement addresses the reduction of the possible behaviors of the overall system. We say that α refines β (with respect to property refinement) if we have $[\![\alpha]\!]_0 \subseteq [\![\beta]\!]_0$. Given the semantics definition from Section 2 it is easy to identify transformations on MSCs that result in refined versions; for instance, removing alternatives, removing interleaving, binding references, strengthening guards, and narrowing loop bounds each results in the property refinement of an MSC.

Message refinement denotes the replacement of a single message by a whole interaction sequence or protocol. We can capture a simple version thereof by substituting the message, say $ch{\triangleright}m$, by a reference to a fresh MSC $\beta_{ch{\triangleright}m}$ that specifies the protocol into which $ch{\triangleright}m$ decomposes. Given an MSC α and a pair $(X, \beta_{ch{\triangleright}m}) \in \mathit{MSCDEF}$ such that no reference to X occurs in α we denote the substitution of $ch{\triangleright}m$ with $\rightarrow X$ by $\alpha[\rightarrow X/ch{\triangleright}m]$. Under these circumstances we say that $\alpha[\rightarrow X/ch{\triangleright}m]$ refines α (with respect to message refinement). This refinement notion is very liberal with respect to the protocol that can replace the original message; stronger forms of message refinement could require, for instance, that $\beta_{ch{\triangleright}m}$ must begin or end with a message exchange on a channel that connects the same components and has the same direction as ch.

Structural refinement denotes replacing a single component that occurs in an MSC α, say p, by a set of (other) components, say $\{p_0, ..., p_n\}$ for some $n \in \mathbb{N}$. Intuitively, this results in substituting the source component of each output channel for messages emanating from p in α by a corresponding element from $\{p_0, ..., p_n\}$; we need an analogous substitution for the destination component of each input channel. Typically, the components into which p refines will exchange additional messages among each other besides the ones that they inherit from p. We capture structural refinement by introducing a function $f^p_{\{p_0,...,p_n\}} : \mathit{MSCT} \rightarrow \mathit{MSCT}$ that performs two operations on a given MSC: if either of a channel's source or destination component (but not both) is in $\{p_0, ..., p_n\}$ it maps it to p, and if

both the channel's source and its destination component are in $\{p_0, ..., p_n\}$ it removes the message completely from the MSC. Given these preliminaries we say that β refines α (with respect to structural refinement) if $[\![f^p_{\{p_0,...,p_n\}}.\beta]\!]_0 \subseteq [\![\alpha]\!]_0$ holds.

There is an interesting relationship between the refinement notions that we have introduced here and the refinement of individual components. Typically, a refinement step on an MSC affects multiple components. The removal of a message by performing a property refinement, for instance, weakens the assumptions of the recipient, while it strengthens the commitment of the sender. In [BK98] we describe the implications of this relationship and discuss corresponding notions of joint and individual component refinement.

"Advanced" MSC composition In Section 2.2 we have introduced various operators for composing an individual MSC from smaller units. Typically, however, we use multiple MSCs to specify different interaction scenarios, use cases or operation invocations. In general we cannot expect that each MSC provides an orthogonal view on the system's complete interaction behavior. Almost certainly will we face MSCs that show the same components in overlapping collaborations. Moreover, there may exist MSCs that complement the system properties specified by others, repeating part of the information contained in the latter. Here, we introduce two examples of composition operators that reflect these needs: MSC join and trigger composition.

The join of two MSCs α and β, which we denote by $\alpha \otimes \beta$, is similar to their interleaving with the exception that it identifies common messages on common channels of the two operands. This operator allows us to draw MSCs in which the same components can appear in different roles with overlapping interaction behavior. Join is a restriction of the parallel composition operator we have introduced in [BGH+98]; there we allowed to state explicitly which messages the operator should identify.

The trigger composition of two MSCs α and β, which we denote by $\alpha \mapsto \beta$, specifies that whenever the interaction sequence that α specifies has occurred in the system then the interaction sequence that β specifies is inevitable. [Krü99] contains a semantics definition for trigger composition along the lines of Section 2.2. This operator allows us to complement any MSC γ with additional liveness properties: we simply have to intersect $[\![\gamma]\!]_0$ with $[\![\alpha \mapsto \beta]\!]_0$. Given, for instance, an MSC that specifies the infinite repetition of the choice among a number of alternatives where one of these alternatives is **empty**, we can now, using trigger composition, specify that a nonempty alternative must eventually occur.

3.4 Occurrence Conditions

For any given MSC α (or, similarly, a given set of MSCs) the question arises how much information it provides about the system under development, or, put another way, when does an interaction as specified by α occur? We distinguish the following cases:

exact interpretation: α describes the complete interaction behavior of the system. This

corresponds to the closed world assumption we made in Section 3.2 for deriving an automaton representation for individual system components.

existential interpretation: there are system executions that exhibit the interaction sequences that α describes; others may display completely arbitrary behavior. This is the interpretation that underlies tools like SDT that allow the developer to check whether the system does not forbid a particular execution. However, this interpretation is of limited value in a constructive, refinement oriented development approach, because it is non-monotonic with respect to the refinement notions we have introduced above.

universal interpretation: the interaction sequence that α describes occurs in every execution of the system. This is a generalization of the exact interpretation, in that it allows other interaction sequences to occur in the system as long as they interleave with α's.

Of course, we can also consider negations of these interpretations to identify that α specifies a certain "undesirable" property of the system we develop.

4 Conclusion

In the preceding sections we have discussed several issues that arise in the seamless integration of MSCs into the overall software development process. The system model and the semantics definition we have given in Section 2 provides the basis for the methodical applications of MSCs that we have explored in Section 3. By discussing the derivation of both A/C specifications and state-automata for individual components from MSCs, and by defining refinement notions, advanced composition operators and occurrence conditions for MSCs, we have addressed the questions we have raised in Section 1.

Together, the techniques we have described in this paper constitute a first step towards the methodical application of MSCs in conjunction with other structural and behavioral system views. In particular, our approach allows us to push MSCs closer to becoming a description technique for complete component behavior, instead of being used for simple interaction scenarios only.

Acknowledgments We are grateful to Manfred Broy, Max Breitling, Bernd Finkbeiner, Radu Grosu, Jan Philipps, Andreas Rausch, Ekkart Rudolph, Bernhard Rumpe, and Bernhard Schätz, for interesting discussions on the topics presented here. We are especially indebted to Michael von der Beeck, Katharina Spies, and Thomas Stauner, who read a draft version of this text and provided very valuable feedback.

References

[BGH+98] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and complete object interaction descriptions. *Computer Standards & Interfaces*, 19:335 – 345, 1998.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. A graphical description technique for communication in software architectures. Technical Report TUM-I9705, Technische Universität München, 1997.

[BK98] Manfred Broy and Ingolf Krüger. Interaction Interfaces — Towards a scientific foundation of a methodological usage of Message Sequence Charts. In *ICFEM'98*. IEEE, 1998.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley, 1996.

[GKS] Radu Grosu, Ingolf Krüger, and Thomas Stauner. Hybrid Sequence Charts. (submitted to FM'99).

[IT96] ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.

[IT98] ITU-TS. Recommendation Z.120 : Annex B. Geneva, 1998.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In *DIPES'98*. Kluwer, 1999.

[Krü99] Ingolf Krüger. *Using MSCs for Design and Validation of Distributed Software Components*. PhD thesis, Technische Universität München, 1999. (in preparation).

[MR94] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), 1994.

[MR96] S. Mauw and M. A. Reniers. Refinement in Interworkings. In U. Montanari and V. Sassone, editors, *CONCUR'96*, volume 1119 of *LNCS*, pages 671–686. Springer, 1996.

[Rat97] Unified modeling language, version 1.1. Rational Software Corporation, 1997.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[WD98] Alan Cameron Wills and Desmond D'Souza. *Objects, Components, and Frameworks with UML– The Catalysis Approach*. Addison Wesley, 1998.