Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs

Ingolf H. Krüger

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA, 92093-0114, USA
ikrueger@ucsd.edu

Abstract. Message Sequence Charts (MSCs) and related notations have found wide acceptance for scenario-oriented behavior specifications. However, MSCs lack adequate support for important aspects of interaction modeling, including overlapping interactions, progress/liveness specifications, and preemption. Such support is needed particularly in the context of *service-oriented* specifications both in the business information and embedded systems domain. In this text, we introduce extensions to the "standard" MSC notation addressing these deficits, and provide a semantic foundation for these extensions.

1 Introduction

The complexity of software-enabled systems continues to rise. Driven by the wired and wireless incarnations of the Internet, the networking of formerly monolithic devices and their software components increases. More and more systems emerge as a collaboration between peer networking nodes, each offering software functions to its environment, and utilizing the functions offered by others.

As a consequence, the focus of concern in requirements capture, design, and deployment of software solutions shifts from individual computation nodes to their interaction. This shift of concern is exemplified by the current trend towards "web services". Here, software functions (called *services*) are published as individual entities at well-known addresses on the Internet; these functions can be consumed by others, yielding possibly complex, composite services.

To model and implement such interaction-based services systematically, expressive description techniques and methodological foundation for component interaction are essential.

Typical software development approaches and modeling languages, however, place their focus on the construction of individual software components, instead of on component collaboration. The Unified Modeling Language (UML)[16] is a typical example (at least before version 2.0, which is still under discussion). Its syntactic means – and corresponding tool support – for specifying state-based behavior of individual components (statechart diagrams) are far better developed than the corresponding notations for interaction patterns (activity, sequence and collaboration diagrams).

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 387402, 2003.

Message Sequence Charts (MSCs)[7,8], on the other hand, have been widely accepted as a valuable means of visualizing and specifying asynchronous component interaction. Their potentials in this regard have earned them entrance into the current suggestion for the UML's 2.0 standard, where they are adapted to support modeling of a wide range of communication concepts.

Both of these notations, however, provide only very basic support for interaction specifications. In this text, we show how to extend the expressiveness of MSCs and sequence diagrams to include:

- Overlapping interaction patterns
- Liveness/progress properties
- Preemption specifications.

In the following paragraphs we describe each of these extensions in more detail. Although there are many other areas for improvement we could address – including proper handling of data in sequence diagrams, and hierarchical refinement for messages and components, to mention just two examples – for reasons of brevity we focus on the three extensions listed below.

1.1 Overlapping Interaction Patterns

We call sequences of interactions in which some communication partners and some of the messages they exchange coincide *overlapping*; overlapping interaction patterns are extremely important in service-oriented specifications. Each individual service only represents a *partial* view on the collaborations within the system under consideration. To get the *overall* picture for one implementation component, say, all the different services in which a single component is involved need to be joined. This requires an adequate composition operator for making the relationship between different service specifications precise.

1.2 Trigger Composition

Similarly important is availability of an operator for specifying liveness/progress in MSCs. Most sequence diagram dialects provide means for indicating alternative interaction patterns; they fail, however, to offer notation for indicating which alternatives should be selected to make progress towards a desirable goal. Consequently, liveness properties can only be described as a side remark or in another modeling language, such as an state-automaton-based approach. Without proper support for liveness, sequence diagrams cannot mature beyond scenario specifications. Specifically, we are interested in working with abstract progress properties, such as "if a certain interaction pattern has occurred in the system, then another one is inevitable", or "one interaction pattern *triggers* another".

1.3 Preemption

Preemption is a fundamental concept especially in technical and embedded systems development. Although the very roots of MSCs are in telecommunication

systems, where preemption scenarios abound – think of specifying a telephone call, where at any time either party can hang up –, no support for preemption exists in either MSCs or sequence diagrams.

1.4 Contributions and Outline

Our contributions in this text are twofold. First, in Sect. 2, we provide a motivating "toy" example, illustrating the usability and applicability of the concepts just outlined. This example serves also to introduce the extensions to the MSC syntax we use, and how they integrate with the "standard" notation. Although we have chosen to stay close to the MSC-96 syntax, the example illustrates that our suggestions could also be transferred to other interaction-based specification techniques, such as the UML's sequence diagrams.

Second, in Sect. 3 we provide a formal semantics for MSCs including support for the new composition operators that address overlapping scenarios, triggered and preemptive collaborations. The basis for this semantics is a precise system model for component interaction, based on streams. There have certainly been other attempts at semantics definition for MSCs before; the combination of explicit concepts for overlapping, trigger composition, and preemption of MSCs is, however, not treated consequently in these approaches. In addition, our approach has the advantage of supporting systematic MSC refinement as well as component synthesis from MSCs – for reasons of brevity we have to refer the reader to [9] for more details on these issues.

We discuss related work in Sect. 4, as well as our conclusions and future work in Sect. 5.

2 Example: The Abracadabra-Protocol

To illustrate the applicability of the suggested operators, we model a simplified version of the Abracadabra communication protocol[1,2] using MSCs[7,15,9]. To describe this protocol we assume given a system consisting of two distinct components X and Y; we assume further that these two components communicate via messages sent along channels xy (from X to Y), and yx (from Y to X). Figure 1 shows this component structure in graphical form.

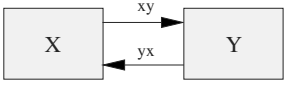


Fig. 1. System Structure Diagram (SSD) for the Abracadabra-protocol

The symmetric Abracadabra-protocol describes a scheme that allows any of the two components to establish a connection to the other component, send data messages once a connection exists, and tear down an existing connection it

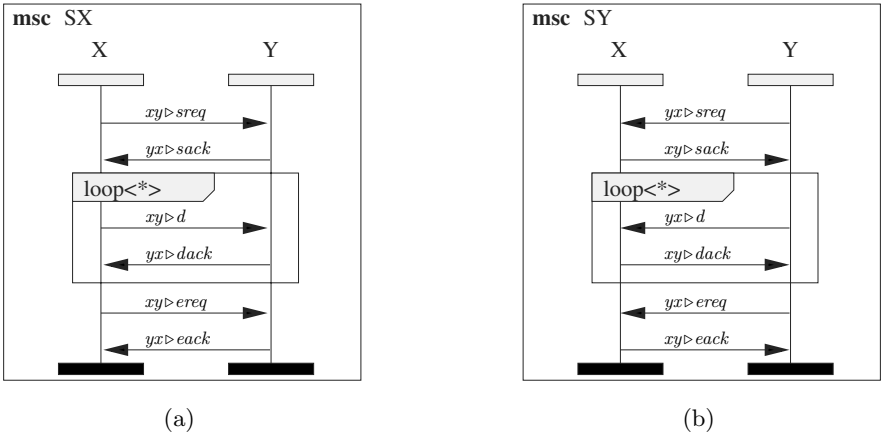
has initiated. If both components try to establish a connection simultaneously, the system is in conflict. Then, both components tear down their "attempted" connections to resolve the conflict.

Section 3 contains the formal definitions corresponding to the concepts and operators introduced informally here.

2.1 Message Sequence Charts

MSCs provide a rich graphical notation for capturing interaction patterns. MSCs have emerged in the context of SDL[6] as a means for specifying communication protocols in telecommunication systems. They have also found their way into the new UML 2.0 standard[14], which significantly improves the role of interaction models within the UML.

MSCs come in two flavors: Basic and High-Level MSCs (HMSCs). A basic MSC consists of a set of axes, each labeled with the name of a component. An axis represents a certain segment of the behavior displayed by its corresponding component. Arrows in basic MSCs denote communication. An arrow starts at the axis of the sender; the axis at which the head of the arrow ends designates the recipient. Intuitively, the order in which the arrows occur (from top to bottom) within an MSC defines possible sequences of interactions among the depicted components.



(a) (b)

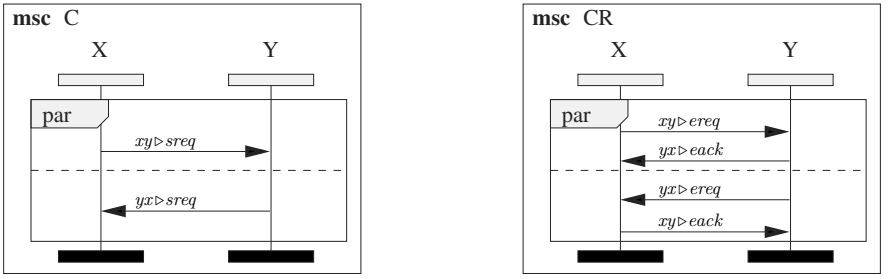
Fig. 2. MSCs for successful transmission

As an example, consider the MSC of Fig. 2a. It depicts how X and Y interact to establish successful data transmission. X initiates the interaction by sending message $xy{\triangleright}sreq$ ("sending requested") to Y. Upon receipt of Y's reply $yx{\triangleright}sack$ ("sending acknowledged"), X sends an arbitrary, finite number of $xy{\triangleright}d$ messages. Each data message is acknowledged individually by Y; X waits for a $yx{\triangleright}dack$ message from Y before sending the next data message. Graphically,

repetition is indicated by the loop box enclosing the recurring messages. To close the transmission X sends message $xy \triangleright ereq$ ("end requested") to Y; Y acknowledges transmission termination by means of a $yx \triangleright eack$ ("end acknowledged") message. Figure 2b shows the symmetric case, where Y is the initiator.

Syntactically, we have adopted a slightly modified version of MSC-96[7]; our message arrows carry an indicator for the channel on which a message is sent in addition to the message itself; we will come back to this in Sect. 3.2. Moreover, we use unbounded loops, which are not available in MSC-96.

Conflict in the ABRACADABRA protocol occurs if both X and Y try to establish a connection simultaneously. The MSC in Fig. 3a captures this case by means of causally unrelated messages, using the "parallel box" syntax of MSC-96. Conflict resolution is handled by mutual exchange of messages *ereq* and *eack*



(a) (b)

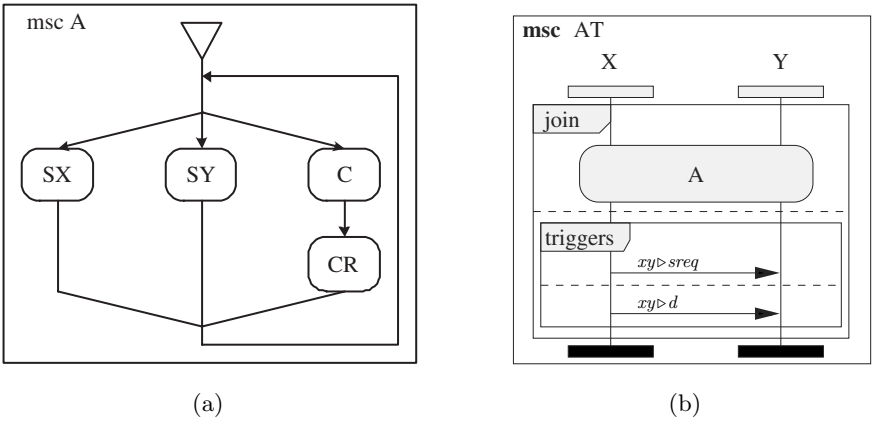
Fig. 3. MSCs for conflicts and their resolution

by X and Y. Again, there need not be a specific order between the *ereq* and *eack* messages with different origins (cf. Fig. 3b).

An HMSC is a graph whose nodes are references to other (H)MSCs. The semantics of an HMSC is obtained by following paths through the graph and composing the interaction patterns referred to in the nodes along the way. The HMSC of Fig. 4a, for instance, specifies that every system execution is an infinite sequence of steps, where each step's behavior is described by one of the following: MSC SX (successful communication initiated by X), MSC SY (successful communication initiated by Y), or MSC C (conflict) followed by MSC CR (conflict resolution). Multiple vertices emanating from a single node in the HMSC graph indicate nondeterministic choice.

2.2 Introducing Progress/Liveness

So far, we have left open whether a send request by either component will, eventually, result in an established connection. The use of nondeterministic choice in the definition of MSC A allows an infinite sequence of steps consisting only of conflict and conflict resolution.



(a) (b)

Fig. 4. HMSC for the ABRACADABRA-protocol, and ABRACADABRA with progress property

We introduce a new composition operator, called "trigger composition", to cast the progress/liveness property (cf. [4,13]) that a message $xy \triangleright sreq$ must lead to subsequent data exchange, i.e. occurrence of at least one $xy \triangleright d$ message.

Informally speaking, we write $\alpha \mapsto \beta$ to indicate that whenever the interaction pattern specified by MSC α has occurred in the system under consideration, it is eventually followed by an occurrence of the interaction pattern specified by MSC β.

Having trigger composition available, we can describe the progress property mentioned above as $xy \triangleright sreq \mapsto xy \triangleright d$; see Fig. 4b for the graphical syntax we use for trigger composition.

2.3 Joining Overlapping MSCs

By now, we have two separate MSCs describing system behaviors. On the one hand we have MSC A, which describes the major interaction patterns of the ABRACADABRA protocol. On the other hand we have the MSC $xy \triangleright sreq \mapsto xy \triangleright d$, which describes a progress property relating occurrences of messages $xy \triangleright sreq$ and $xy \triangleright d$.

Our next step is to compose these two MSCs such that the resulting MSC contains only paths through the ABRACADABRA protocol that fulfill the progress property. To that end, we introduce the "join" composition operator for MSCs. The join $\alpha \otimes \beta$ of two MSCs α and β describes behaviors complying to both MSCs such that identical messages occurring in both MSCs are identified.

The join operator in Fig. 4b "binds" the messages occurring in the trigger composition $xy \triangleright sreq \mapsto xy \triangleright d$ to those in A. The semantics of the joint MSC is the subset of A's semantics where every $xy \triangleright sreq$ message is followed by an $xy \triangleright d$ message eventually. Put another way, no element of the semantics of the joint

MSC has only conflicts or successful transmissions initiated by Y, if X issues message $xy \triangleright sreq$ at least once.

2.4 Introducing Preemption

To demonstrate an application of preemption we extend the informal protocol specification given above as follows: we modify the system structure from Fig. 1 by connecting component X to the "environment" (represented by component *ENV*) through channel ex (cf. Fig. 5).

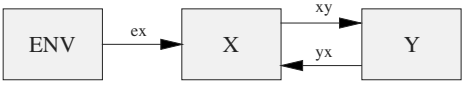
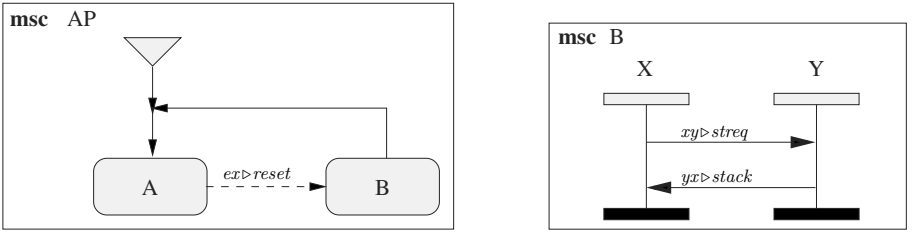


Fig. 5. SSD for the ABRACADABRA-protocol with preemption

By sending message *reset* along channel ex the environment can force X to stop any communication it may currently be involved in, and to restart the whole protocol afresh. Upon receipt of message *reset*, component X sends message *streq* ("stop requested") to Y; Y replies by sending message *stack* ("stop acknowledged") to X.

The HMSC AP from Fig. 6a models this behavior by means of a preemption arrow labeled with the preemptive message $ex \triangleright reset$. MSC B (cf. Fig. 6b) shows the handling of the preemption. Recall that MSC A (cf. Fig. 4a) captures the overall behavior of the ABRACADABRA-protocol (without preemption).



(a) (b)

Fig. 6. Preemption and its handling

Without the preemption construct we would have to rewrite the MSCs A, SX, SY, C, and CR completely to accommodate the external reset request. The resulting MSCs would lose their intuitive appeal almost entirely, because the one exceptional case would dominate the whole specification.

3 A Formal Framework for Precise MSC Specifications

In this section we introduce the formal framework for the semantics definition of MSCs. We use this framework, in particular, to describe the semantics of the operators for the join, trigger, and preemptive composition of MSCs.

3.1 System Model

We prepare our precise semantics definition for MSCs by first introducing the structural and behavioral model (the *system model*) on which we base our work. Along the way we introduce the notation and concepts we need to describe the model.

System Structure. Structurally, a system consists of a set P of components, objects, or processes[1], and a set C of named channels. Each channel $ch \in C$ is directed from its source to its destination component; we assume that channel names are unique. Channels connect components that communicate with one another; they also connect components with the environment. Communication proceeds by message exchange over these channels.

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 $S \stackrel{\mathsf{def}}{=} \Pi_{p \in P} S_p$. For simplicity, we represent messages by the set M of message identifiers.

System Behavior. Now we turn to the dynamic aspects of the system model. We assume that the system components communicate among each other 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 it triggers; more precisely, the delay establishes a strict causality between an output and its triggering input (cf. [3,2]).

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 and relations on streams are an extremely powerful specification mechanism for distributed, interactive systems (cf. [3,17]). Here, we only use and introduce a small

[1] In the remainder of this document, we use the terms components, objects, and processes interchangeably.

fraction of this rich semantic model; for a thorough introduction to the topic, we refer the reader to [17,3].

A stream is a finite or infinite sequence of messages. By X^* and X^∞ we denote the set of finite and infinite sequences over set X, respectively. $X^\Box \stackrel{\mathsf{def}}{=} X^* \cup X^\infty$ denotes the set of streams over set X. We identify X^* and X^∞ with $\bigcup_{i\in\mathbb{N}}([0,i] \to X)$ and $\mathbb{N} \to X$, respectively, and use function application to write $x.n$ for the n-th element of stream x (for $x \in X^\Box$ and $n \in \mathbb{N}$).

We define $\tilde{C} \stackrel{\mathsf{def}}{=} C \to M^*$ as a channel valuation that assigns a sequence of messages to each channel; we obtain the timed stream tuple $\tilde{C}^\infty$ as an infinite valuation of all channels. This models that at each point in time a component can send multiple messages on a single channel.

With timed streams over message sequences 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 an element of $\mathcal{P}((\tilde{C} \times S)^\infty)$. For notational convenience we denote for $\varphi \in (\tilde{C} \times S)^\infty$ by $\pi_1(\varphi)$ and $\pi_2(\varphi)$ the projection of φ onto the corresponding infinite channel and state valuations, respectively; thus, we have $\pi_1(\varphi) \in \tilde{C}^\infty$ and $\pi_2(\varphi) \in S^\infty$. The existence of more than one element in the semantics of a system indicates nondeterminism.

3.2 MSC Semantics

In the following we establish a semantic mapping from MSCs to the formal framework introduced above. In the interest of space we constrain ourselves to a significant subset of the notational elements contained in MSC-96 and UML 2.0, yet provide the extensions for overlapping scenarios (join operator), progress (trigger composition), and preemption. For a comprehensive treatment of MSC syntax and semantics we refer the reader to [9]; further approaches to defining MSC semantics are discussed in Sect. 4.

Preliminaries. To facilitate the semantics definition we use a simplified textual syntax for MSCs. The base constructors for MSCs are **empty**, $c \triangleright m$ and **any**, denoting the absence of interaction (empty MSC), the sending of message m on channel c, and arbitrary interactions, respectively. Given two MSCs α and β we denote by $\alpha\,;\,\beta$ and $\alpha \sim \beta$ the sequencing and interleaving of α's and β's interaction patterns, respectively. If g represents a predicate on the state space of the system under consideration, then we call $g : \alpha$ a guarded MSC; intuitively it equals **empty** if g evaluates to false, and α otherwise. By $\alpha \otimes \beta$ we denote the join of MSCs α and β. The join of two MSCs corresponds to the interleaving of the interaction sequences they represent with the exception that common messages on common channels synchronize. The trigger composition, written $\alpha \mapsto \beta$ of two MSCs expresses the property that whenever the interactions specified by α have occurred the interactions specified by β are inevitable. We write $\alpha \stackrel{ch\Box m}{\rightarrow} \beta$

to denote preemption. If message $ch \triangleright m$ occurs during the behavior represented by α, then this behavior is preempted and continued by the behavior that β represents. Intuitively, we can think of β as the preemption/exception handler, and of $ch \triangleright m$ as the exception being thrown. To express the restarting of α upon occurrence of preemptive message $ch \triangleright m$, we write $\alpha\Uparrow_{ch \triangleright m}$. $\alpha\uparrow_g$ denotes a "while" loop, repeating the interactions of α while g evaluates to true; a special case is $\alpha\uparrow_\infty$, which denotes an infinite repetition of α. We define $\alpha^0 \stackrel{\text{def}}{=} \textbf{empty}$, and $\alpha^{i+1} \stackrel{\text{def}}{=} (\alpha\,;\,\alpha^i)$ for $i \in \mathbb{N}$. $\alpha\uparrow_*$ denotes unbounded finite repetition of α.

An MSC definition associates a name with an interaction specification, written **msc** X $= \alpha$. By ⟨MSC⟩ and ⟨MSCNAME⟩ we denote the set of all syntactically correct MSCs, and MSC names, respectively. An MSC document consists of a set of MSC definitions (assuming unique names for MSCs within a document). To reference one MSC from within another we use the syntax $\rightarrow Y$, where Y is the name of the MSC to be referenced.

Example: As an example for the representation of MSCs in the syntax introduced above we consider again the service depicted in Fig. 2a. In our textual syntax the scenario is expressed as follows:

$$\textbf{msc}\ \mathrm{SX} =$$
$$xy \triangleright sreq\ ;\ yx \triangleright sack\ ;\ (xy \triangleright d\ ;\ yx \triangleright d)\uparrow_*\ ;\ xy \triangleright ereq\ ;\ yx \triangleright eack$$

The textual MSC definition corresponding to the graphical representation in Fig. 6a is

$$\textbf{msc}\ AP = ((\rightarrow A) \overset{ex \square reset}{\rightarrow} (\rightarrow B))\uparrow_{<\infty>}$$

Denotational Semantics. In this section we introduce the semantic mapping from the textual representation of MSCs into the semantic domain $(\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. 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 Sect. 3.1. Put another way, we interpret an MSC as a constraint on the possible behaviors of the system under consideration. More precisely, with every $\alpha \in$ ⟨MSC⟩ and every $u \in \mathbb{N}_\infty$ 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. Intuitively, u corresponds to the "starting time" of the behavior represented by the MSC; t indicates the time point when this behavior has finished. Hence, outside the time interval specified by u and t the MSC α makes no statement whatsoever about the interactions and state changes happening in the system. To model that we cannot observe (or constrain) system behavior "beyond infinity" we define that for all $\varphi \in (\tilde{C} \times S)^\infty$, $\alpha \in$ ⟨MSC⟩, and $t \in \mathbb{N}_\infty$ the following predicate holds: $(\varphi\, t\,) \in [\![\alpha]\!]_\infty$.

We assume given a relation $MSCR \subseteq$ ⟨MSCNAME⟩ × ⟨MSC⟩, which associates MSC names with their interaction descriptions. We expect $MSCR$ to be

the result of parsing all of a given MSC document's MSC definitions. For every MSC definition **msc** $X = \alpha$ in the MSC document we assume the existence of an entry (X, α) in *MSCR*. For simplicity we require the MSC term associated with an MSC name via *MSCR* to be unique.

Empty MSC. For any time $u \in \mathbb{N}_\infty$ **empty** describes arbitrary system behavior that starts and ends at time u. Formally, we define the semantics of **empty** as follows:

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

Arbitrary Interactions. MSC **any** describes completely arbitrary system behavior; there is neither a constraint on the allowed interactions and state changes, nor a bound on the time until the system displays arbitrary behavior:

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

any has no direct graphical representation; we use it to resolve unbound MSC references (see below).

Single Message. An MSC that represents the occurrence of message m on channel ch constrains the system behavior until the minimum time such that this occurrence has happened:

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

Because we disallow pairs $(\varphi\ \infty)$ in $[\![ch \triangleright m]\!]_u$ we require the message to occur eventually (within finite time). This corresponds with the typical intuition we associate with MSCs: the depicted messages do occur within finite time.

We add the channel identifier explicitly to the label of a message arrow in the graphical representation; this is useful in situations where a component has more than one communication path to another component.

Sequential Composition. The semantics of the semicolon operator is sequential composition (*strong sequencing* in the terms of [7]): given two MSCs α and β the MSC $\alpha\,;\,\beta$ denotes that we can separate each system behavior in a prefix and a suffix such that α describes the prefix and β describes the suffix:

$$[\![\alpha\,;\,\beta]\!]_u \stackrel{\mathsf{def}}{=} \{(\varphi\, t\,) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \langle \exists t' \in \mathbb{N}_\infty :: (\varphi\, t'\,) \in [\![\alpha]\!]_u \ \wedge\ (\varphi\, t\,) \in [\![\beta]\!]_{t'} \rangle\}$$

Guarded MSC. Let $K \subseteq P$ be a set of instance identifiers. By p_K we denote a predicate over the state spaces of the instances in K. Let $[\![p_K]\!] \in \mathcal{P}(S)$ denote the set of states in which p_K holds. Then we define the semantics of the guarded MSC $p_K : \alpha$ as the set of behaviors whose state projection fulfills p_K at time u, and whose interactions proceed as described by MSC α:

$$[\![p_K : \alpha]\!]_u \stackrel{\text{def}}{=} \{(\varphi\, t\,) \in [\![\alpha]\!]_u : \pi_2(\varphi).u \in [\![p_K]\!]\}$$

We require p_K to hold only at instant u. This allows arbitrary state changes from time u on. In particular, at no other point within the time interval covered by α can we assume that p_K still holds.

Alternative. An alternative denotes the union of the semantics of its two operand MSCs. The operands must be guarded MSCs; the disjunction of their guards must yield true. Thus, for $\alpha = p : \alpha'$, $\beta = q : \beta'$ with $\alpha', \beta' \in \langle \text{MSC} \rangle$, and guards p, q with $p \vee q \equiv \text{true}$ we define:

$$[\![\alpha \mid \beta]\!]_u \stackrel{\text{def}}{=} [\![\alpha]\!]_u \cup [\![\beta]\!]_u$$

For guards p and q with $p \wedge q \equiv \text{true}$ the alternative expresses a nondeterministic choice.

References. If an MSC named X exists in the given MSC document, i.e. there exists a pair $(X, \alpha) \in \mathit{MSCR}$ for some $\alpha \in \langle \text{MSC} \rangle$, then the semantics of a reference to X equals the semantics of α. Otherwise, i.e. if no adequate MSC definition exists, we associate the meaning of **any** with the reference:

$$[\![\rightarrow X]\!]_u \stackrel{\text{def}}{=} \begin{cases} [\![\alpha]\!]_u & \text{if } (X, \alpha) \in \mathit{MSCR} \\ [\![\mathbf{any}]\!]_u & \text{else} \end{cases}$$

To identify **any** with an unbound reference has the advantage that we can understand the binding of references as a form of property refinement (cf. [9]).

Interleaving. Intuitively, the semantics of interleaving MSCs α and β "merges" elements $(\varphi\, t\,) \in [\![\alpha]\!]_u$ with elements $(\psi, t\,') \in [\![\beta]\!]_u$. Formal modeling of this merge is straightforward, albeit more technically involved; we refer the reader to [9] for the details.

Join. The join $\alpha \otimes \beta$ of two operand MSCs α and β is similar to their interleaving with the exception that the join identifies common messages, i.e. messages on the same channels with identical labels in both operands. MSC-96 does not offer an operator with a similar semantics.

$$\begin{gathered}
[\![\alpha \otimes \beta]\!]_u \stackrel{\text{def}}{=} \{(\varphi\, t\,) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
\langle \exists t_1, t_2 :: (\varphi\, t_{\,1}) \in [\![\alpha]\!]_u \wedge (\varphi\, t_{\,2}) \in [\![\beta]\!]_u \wedge t = \max(t_1, t_2) \rangle \\
\wedge \langle \forall X \in (\mathit{msgs}.\alpha \cap \mathit{msgs}.\beta)^*, \psi \in (\tilde{C} \times S)^\infty, \mathit{ch} \in C, t' \in [u, t] \cap \mathbb{N} :: \\
((X \neq \langle\rangle) \wedge (\pi_1(\psi).t'.\mathit{ch} = \pi_1(\varphi).t'.\mathit{ch} \setminus X)) \\
\Rightarrow \langle \forall t'' \in \mathbb{N} :: (\psi, t\,'') \notin [\![\alpha]\!]_u \wedge (\psi, t\,'') \notin [\![\beta]\!]_u \rangle \rangle\}
\end{gathered}$$

In this definition we use the notation $m \backslash n$ as a shorthand for the stream obtained from m by dropping all elements that do also appear in n. By $msgs.\gamma$ we denote the set of message labels occurring in MSC γ.

The second outer conjunct of this definition ensures that we *cannot* reconstruct the behaviors of α and β independently from the behaviors of their join, if the two MSCs have messages in common. This distinguishes the join clearly from the interleaving of α and β, if $msgs.\alpha \cap msgs.\beta \neq \emptyset$ holds. In this state of affairs, we call α and β *non-orthogonal* (or *overlapping*); if $msgs.\alpha \cap msgs.\beta = \emptyset$ holds, then we call α and β *orthogonal* (or *non-overlapping*).

The definition of the join operator is quite restrictive; for example, consider the two MSCs $\alpha = c{\triangleright}m\,;\ c{\triangleright}n$ and $\beta = c{\triangleright}n\,;\ c{\triangleright}m$, which define two different orderings of the messages $c{\triangleright}m$ and $c{\triangleright}n$. It is easy to see that we have $[\![\alpha \otimes \beta]\!]_u = \emptyset$.

Preemption. The semantics of MSC $\alpha \overset{ch\square m}{\rightarrow} \beta$ is equivalent to the one of α as long as message $ch{\triangleright}m$ has not occurred. From the moment in time at which $ch{\triangleright}m$ occurs, the MSC immediately switches its semantics to the one given by β:

$$
\begin{aligned}
[\![\alpha \overset{ch\square m}{\rightarrow} \beta]\!]_u \overset{\mathsf{def}}{=}\ & \{(\varphi\, t\,) \in [\![\alpha]\!]_u : \langle \forall v \in \mathbb{N} : u \leq v \leq t : m \notin \pi_1(\varphi).v.ch \rangle\} \\
& \cup \{(\varphi\, t\,) \in (\tilde{C} \times S)^{\infty} \times \mathbb{N}_{\infty} : \\
& \quad \langle \exists v : v \in \mathbb{N} : \\
& \qquad v = \min\{t' : t' > u \ \wedge\ m \in \pi_1(\varphi).t'.ch\} \\
& \qquad \wedge\ (\varphi\, v\ -1) \in [\![\alpha]\!]_u^{v-1} \\
& \qquad \wedge\ (\varphi\, t\,) \in [\![\beta]\!]_v \rangle\}
\end{aligned}
$$

Here we use the set $[\![\alpha]\!]_u^v \subseteq (\tilde{C} \times S)^{\infty} \times \mathbb{N}_{\infty}$ for any $\alpha \in \langle \mathrm{MSC} \rangle$, which is similar to $[\![\alpha]\!]_u$ except that each element of $[\![\alpha]\!]_u^v$ constrains the system behavior until time $v \in \mathbb{N}_{\infty}$:

$$
\begin{aligned}
[\![\alpha]\!]_u^v \overset{\mathsf{def}}{=} \{&(\varphi\, v\,) \in (\tilde{C} \times S)^{\infty} \times \mathbb{N}_{\infty} : \\
&\langle \exists (\psi, t\,) : (\psi, t\,) \in [\![\alpha]\!]_u : \varphi|_{[u\text{‹}v]} = \psi|_{[u\text{‹}v]} \ \wedge\ v \leq t \rangle\}
\end{aligned}
$$

Preemptive Loop. The definition of MSC preemption above does not capture the restarting of an interaction in case of the occurrence of a certain message. To handle this case we define the notion of preemptive loop. The intuitive semantics of the preemptive loop of α for a given message $ch{\triangleright}m$ is that whenever $ch{\triangleright}m$ occurs α gets interrupted and then the interaction sequence proceeds as specified by α, again with the possibility for preemption. More precisely, we define $[\![\alpha{\Uparrow}_{ch\square m}]\!]_u$ to equal the greatest fixpoint (with respect to set inclusion) of the following equation:

$$
[\![\alpha{\Uparrow}_{ch\square m}]\!]_u = [\![\alpha \overset{ch\square m}{\rightarrow} (\alpha{\Uparrow}_{ch\square m})]\!]_u
$$

The fixpoint exists due to the monotonicity of its defining equation (with respect to set inclusion).

Trigger Composition. By means of the trigger composition operator we can express a temporal relationship between two MSCs α and β; whenever an interaction sequence corresponding to α has occurred in the system we specify, then the occurrence of an interaction sequence corresponding to β is inevitable:

$$\begin{aligned}[\![\alpha \mapsto \beta]\!]_u \stackrel{\text{def}}{=} \{(\varphi\, t\,) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\ \langle \forall t', t'' : \infty > t'' \geq t' \geq u : \\ (\varphi\, t\,'') \in [\![\alpha]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\varphi\, t\,) \in [\![\beta]\!]_{t'''} \rangle \rangle \}\end{aligned}$$

Loops. The semantics of a guarded loop, i.e. a loop of the form $\alpha\uparrow_p$, where p represents a guarding predicate, is the greatest fixpoint (with respect to set inclusion) of the following equation:

$$[\![\alpha\uparrow_p]\!]_u = [\![(p : (\alpha\,;\, \alpha\uparrow_p)) \mid ((\neg p) : \mathbf{empty})]\!]_u$$

The fixpoint exists because of the monotonicity of its defining equation (with respect to set inclusion); see [9] for the rationale, as well as for other forms of loops (such as bounded finite repetition). On the basis of guarded repetition we can easily define the semantics of α's infinite repetition (written $\alpha\uparrow_\infty$) as follows:

$$[\![\alpha\uparrow_\infty]\!]_u \stackrel{\text{def}}{=} [\![\alpha\uparrow_{\text{true}}]\!]_u$$

For unbounded repetition we define $[\![\alpha\uparrow_*]\!]_u \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} [\![\alpha^i]\!]_u$.

4 Related Work

Suggestions in the literature for MSC dialects abound; [9] contains an extensive list of references. [7,8] defines the standard syntax and semantics for MSC-96. The 2.0 version of the UML[14] has a new interaction model based on MSCs; previous versions adopted a much less powerful notation. LSCs [5] distinguish several interpretations for MSCs (similar to our discussion in [9,10,11]), which allow, in particular, the definition of liveness properties and "anti-scenarios".

None of the above, however, provides operators for treating overlapping scenarios explicitly. By providing the semantic foundation for overlapping using the join operator, we have taken a step towards the independent description of collaborations defining services. Furthermore, although LSCs support complex liveness specifications, we believe that the notion of trigger composition we have borrowed from UNITY's "leadsto" operator (cf. [4]), provides a more accessible way of capturing abstract progress properties. In particular, the combination of trigger composition and join enables separation of concerns in MSC specifications – in the sense of aspect-oriented specification and programming. We refer the reader to [9] for a detailed treatment of further advantages of our model, such as support for MSC refinement and synthesis of component implementations.

5 Conclusions and Outlook

The advent of web services and service-oriented system design in general [12] puts an increasing emphasis on component interaction as the central development aspect. MSCs and similar notations provide a widely adopted means for capturing such interaction patterns in the form of scenarios. MSCs, however, fall short regarding support for important modeling aspects, including overlapping interaction patterns, progress/liveness specifications, and preemption specifications.

In this text, using the ABRACADABRA protocol as an example, we have illustrated the need for corresponding extensions to the MSC notation. In the context of service-oriented system modeling, for instance, specification tools for overlapping interaction patterns are indispensable; the overall system emerges typically as the composition of multiple services such that some components participate in multiple services simultaneously – the corresponding interaction patterns overlap. Similarly, we have shown how to integrate dedicated support for preemption into MSCs as a collaboration-oriented specification notation. Last, but not least, we have also integrated progress specifications into the MSC notation, allowing the developer to express abstract properties such as "whenever one interaction pattern has occurred in the system under consideration, then another interaction pattern is inevitable".

We have also introduced a comprehensive, yet concise mathematical framework for defining the semantics of the extensions we have described. Because our formal model is based on the work in [9], we have at our disposal powerful refinement and synthesis techniques for MSCs even including the extensions described in this text.

Areas for further work include the extension of the treatment of overlapping interaction patterns in the direction of aspect-oriented specifications. Furthermore, the composition operators introduced here need to be substantiated by corresponding tool support; implementation of a corresponding tool prototype is underway.

Acknowledgments. The author is grateful for the reviewers' insightful comments and suggestions. This work was supported by the California Institute for Telecommunications and Information Technology (CAL-(IT)^2).

References

1. Manfred Broy. Some algebraic and functional hocuspocus with ABRACADABRA. Technical Report MIP-8717, Fakultät für Mathematik und Informatik, Universität Passau, 1987. also in: Information and Software Technology 32, 1990, pp. 686–696.
2. Manfred Broy and Ingolf Krüger. Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts. In J. Staples, M. G. Hinchey, and Shaoying Liu, editors, *Formal Engineering Methods (ICFEM'98)*, pages 2–15. IEEE Computer Society, 1998.

3. Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement.* Springer New York, 2001. ISBN 0-387-95073-7.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design. A Foundation.* Addison Wesley, 1988.
5. Werner Damm and David 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.
6. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL. Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1998.
7. ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.
8. ITU-TS. Recommendation Z.120 : Annex B. Geneva, 1998.
9. Ingolf Krüger. *Distributed System Design with Message Sequence Charts.* PhD thesis, Technische Universität München, 2000.
10. Ingolf Krüger. Notational and Methodical Issues in Forward Engineering with MSCs. In Tarja Systä, editor, *Proceedings of OOPSLA 2000 Workshop: Scenario-based round trip engineering.* Tampere University of Technology, Software Systems Laboratory, Report 20, 2000.
11. Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.
12. Ingolf H. Krüger. Specifying Services with UML and UML-RT. Foundations, Challenges and Limitations. *Electronic Notes in Theoretical Computer Science*, 65(7), 2002.
13. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
14. U2 Partners. Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02: Unified Modeling Language 2.0 Proposal. Version 0.671 (draft). available at `http://www.u2-partners.org/artifacts.htm`, 2002.
15. Michel Adriaan Reniers. *Message Sequence Chart. Syntax and Semantics.* PhD thesis, Eindhoven University of Technology, 1999.
16. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1999.
17. Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.