Modeling and Synthesis with MSC Extensions for Broadcasting, Overlapping, Preemptive, and Triggered Collaborations

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

Abstract

While Message Sequence Charts (MSCs) and related notations are valuable in representing basic point-to-point (p2p) communication, they lack adequate support for important aspects of interaction modeling, including broadcasting, preemption, progress/liveness specifications, and overlapping interactions. Such support, however, is needed particularly in the context of service-oriented specifications. In this text, we explore extensions to the “standard” MSC notation addressing these deficits, and provide a rough sketch of corresponding implementation strategies in the context of the CTAS avionics case study.

1. Introduction

Message Sequence Charts (MSCs)[4, 5] are a valuable description technique for visualizing and specifying point-to-point (p2p), asynchronous component interaction. Their potentials in this regard have earned them entrance into the current suggestion for the UML’s 2.0 standard [11] for sequence diagrams (SDs), where they are adapted to support modeling of a wide range of communication concepts.

With the advent of “web services” and corresponding service-oriented approaches for requirements capture, design and implementation of component interaction and its associated description techniques become the main focus of the development process. Development concerns cutting across component boundaries, such as component interaction, are crucial for managing complexity and traceability; this is also recognized on the level of implementation, where aspect-oriented programming is rapidly gaining attention in both academia and industry.

Therefore, strong notational support for capturing collaboration aspects is an essential ingredient for component- and service-oriented software development. However, the currently available notations for component collaboration – including MSCs and SDs – cover only a limited scope of the interaction specifications needed in complex distributed and reactive systems. In this text, we show how to extend the expressiveness of MSCs and sequence diagrams to include broadcasting, preemption specifications, overlapping interaction patterns, and liveness/progress properties. In the following paragraphs we describe each of these extensions in more detail.

Broadcasting  Broadcasting is the central communication paradigm in a wide range of application domains, including automotive, avionics and wireless communications. Neither MSCs nor SDs offer notation for broadcasting, resulting either in awkward graphical specifications, or in abandonment of these description techniques for scenarios in the mentioned application domains.

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, no support for preemption exists in MSCs and SDs. A typical example for the utility of preemption is specification of a telephone call, where at any time either party can hang up. Again, modeling this state of affairs with MSCs or SDs is syntactically awkward at best.

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 a 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”.

**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.

**Contributions and Outline** Our contributions in this text are twofold. First, in Sect. 2, we provide a motivating example, illustrating the utility 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.

Second, in Sect. 3, we briefly discuss the transition from MSC specifications including the concepts introduced here to automaton-based implementations. Basing our approach on hierarchical component models allows us to keep the individual component interfaces and corresponding interaction protocols manageable in size; this enables use of automatic synthesis algorithms, such as [8], in a scalable way.

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

**2. Example: The CTAS Case Study**

To illustrate the applicability of the suggested operators, we model part of the Center TRACON Automation System (CTAS) using MSCs enriched by the extensions introduced above. CTAS is a set of tools and processes for air traffic management at large airports. According to [12] a key element of CTAS is the continual distribution of accurate weather information to client processes. This information is received by a central communications manager process, and then forwarded to all client processes that have successfully registered with the communications manager. Examples for clients are processes for airplane route analysis, aircraft and weather panels, as well as graphical user interfaces for simulation purposes.

For reasons of brevity we limit ourselves to modeling only a modified subset of the requirements stated in [12]; this subset, however, allows us to showcase all of the MSC extensions discussed in Sect. 1. In particular, we consider the use cases 2.8.5 and 2.8.10-15 for a static variant of CTAS, consisting of five distinct components: the communications manager CM, an aircraft panel AP, a weather panel WP, the planview GUI PGUI for simulation, and the route analysis component RA; we assume further that these components communicate via messages sent along directed channels ca (from CM to AP), ac (from AP to CM), cw, wc, cp, pc, cr, and rc. Fig. 1 shows this component structure in graphical form. We do not model dynamic client addition and removal.

![Figure 1. System Structure Diagram (SSD) for the Center TRACON Automation System (CTAS)](image)

MSCs provide a rich graphical notation for capturing interaction patterns. MSCs have emerged in the context of SDL[2] as a means for specifying communication protocols in telecommunication systems. In the following, we give an informal account of the semantics of MSCs and the new concepts introduced here; we refer the reader to [6, 9] for a precise semantics definition.

**Basic MSCs** 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.

As an example, consider the MSC of Fig. 2 (a). It depicts how CM and AP interact to initialize AP as a “weather-aware client” (part of use case 2.8.5). CM initiates the interaction by sending message ca • CTAS_GET_NEW_WTHR to AP. AP replies by sending message ac • yes to CM. Syntactically, we have adopted a slightly modified version of MSC-96[4]; our message arrows carry an indicator for the channel on which a message is sent in addition to the message itself; the channel and message names appear before and after the • symbol, respectively.
Fig. 2 (a) also shows the use of state markers, graphically denoted by hexagons labeled with an identifier for the current state of the component under consideration. We use state markers in this case study to model the state transitions of CM. The corresponding phrases in the requirements document are “The CM should perform the following actions when the Weather Cycle status is ‘X’”, where ‘X’ is the name of the initial state of CM in the use case; the final state is identified by phrases such as “it should set the Weather Cycle status to ‘Y’”, where ‘Y’ is the name of the final state of CM in the use case.

The MSC in Fig. 2 (b) captures an interaction pattern where CM receives yes messages from all components attached to it on the respective channels. These replies are causally unrelated; we indicate this state of affairs by means of the “par-box” syntax of MSC-96. In the regions of the par-box, which are separated by dashed lines, we depict the interaction sequences (consisting of individual messages in our example) occurring mutually independently.

To specify that a component performs some local activity (such as state changes through assignments), MSC-96 provides the concept of actions. Their graphical representation is a labeled rectangle attached to the instance performing the action. Semantically, an action represents a local event of its corresponding instance. Fig. 2 (c) shows an example: CM performs the local action labeled set sockets(updating), which we use to abbreviate the socket assignment referred to in use case 2.8.10. Similarly, the action labeled pu actions in Fig. 3 (c) abbreviates the sequence of assignments mentioned in steps b)-f) of use case 2.8.15.

**Broadcasting** Because MSCs do not offer the notion of broadcasting, specification of requirements such as “it should send a ‘Z’ message to all weather-aware clients”, which pervade almost all use cases in the CTAS requirements document, typically results in extensive use of par-boxes – to indicate independent sending of the same message to the respective recipient. To avoid such awkward notational crutches we introduce dedicated syntax for broadcasting as depicted in Fig. 2 (c). We use a solid line, labeled with the name of the message, together with outlined and filled circles to indicate broadcast messages; outlined and filled circles indicate sender and receivers of the message, respectively. Semantically, we can interpret this notation as an abbreviation for a par-box with one region per recipient of the broadcast message; each region holds precisely one message from the sender (via the appropriate channel) to one of the receivers, such that the message reaches every receiver.

**High-Level MSCs (HMSCs)** 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. 3 (a), for instance, specifies that every system execution is an infinite repetition of the sequence of MSCs consisting of pre updating followed by updating, which, in turn, is followed by post updating. The meaning of the labeled dashed arrow will become clear in the next paragraph.

**Introducing Preemption** The sequence of steps given by the HMSC, together with the MSCs pre updating, updating, post updating of Fig. 2 (c), Fig. 3 (b), and (c), respectively, represents the “main path” through CM’s weather update cycle. This path is described in use cases 2.8.10-12 and 2.8.14; it assumes that all weather-aware clients respond “yes” to CM’s CTAS GET NEW WTHR message. Use case 2.8.13 describes an alternate path to be taken if at least one client’s response is “no”: in this case the behavior as specified in MSC failed updating is to be executed (cf. Fig. 4 (a)). Modeling this alternate path with the means provided by MSC-96 naively is quite complex, as the requirement containing the phrase “any of” implies a combinatorial explosion of possible scenarios. To avoid this syntactic clutter we use the dedicated notation for preemption we have introduced in [6]. In an HMSC we denote preemption by means of a dashed arrow leading from the interaction pat-
tern to be preempted to the interaction pattern handling the preemption. The label of the dashed arrow indicates the message triggering the preemption; by \{c_0, \ldots, c_n\} \cdot m we denote that any of the messages \(c_0 \cdot m\) through \(c_n \cdot m\) will trigger the preemption. Using this notation we can model use case 2.8.13 as an “exception” of the use case 2.8.12 in a rather transparent way.

For the purpose of illustration we deviate from the CTAS requirements document, and restart the weather update cycle after handling the exception instead of following use case 2.8.16.

Progress/Liveness The HMSC \(\text{CTAS\_update}\) of Fig. 3 (a) leaves open whether an update cycle eventually reaches the post_updating interaction pattern. The use of preemption and repetition in the definition of MSC \(\text{CTAS\_update}\) allows an infinite sequence of steps consisting only of preempted update attempts.

We introduce a new composition operator, called “trigger composition”, to cast the progress/liveness property that every occurrence of an interaction pattern \(A\) inevitably leads to subsequent occurrence of another interaction pattern \(B\). In our example we use trigger composition to indicate that every message \(\text{CTAS\_GET\_NEW\_WEATHER}\) is followed by a \(\text{CM\_GROUND\_WIND\_SETTING}\); this implies that there are no system runs consisting entirely of failed update attempts. Fig. 4 (b) shows the graphical syntax we use for trigger composition.

Trigger composition allows us to specify potentially complex temporal dependencies of interaction patterns in a transparent way. In our simplified CTAS model, for instance, the trigger composition of Fig. 4 (b) does not make a statement about how the progress condition should be established; this is left as a design decision for later stages of the development process.

Joining Overlapping MScs By now, we have two separate MScs describing system behaviors. On the one hand we have MSc \(\text{CTAS\_update}\), which describes the major interaction patterns of the CTAS part we have modeled. On the other hand we have the MSc \(\text{CTAS\_trigger}\), which describes a progress property relating occurrences of messages \(\text{CTAS\_GET\_NEW\_WTHR}\) and \(\text{CM\_GROUND\_WIND\_SETTING}\).

Our next step is to compose these two MScs such that the resulting MSc contains only paths through \(\text{CTAS\_update}\) that fulfill the progress property. To that end, we introduce the “join” composition operator for MScs. The join of two MScs describes behaviors complying to both MScs such that identical messages occurring in both MScs are identified.

The join operator in Fig. 4 (c) “binds” the messages occurring in the trigger composition to those in \(\text{CTAS\_update}\). The semantics of the joint MSc is the subset of \(\text{CTAS\_update}\)’s semantics where every \(\text{CTAS\_GET\_NEW\_WTHR}\) message is followed by a \(\text{CM\_GROUND\_WIND\_SETTING}\) message eventually.

3. Component Synthesis

In the interest of space we give only a brief account of the key steps for synthesizing component prototypes from the MScs we have elicited from the CTAS requirements document. We refer the reader to [8] for details on the algorithm we have developed for state automaton synthesis from basic MScs. In [6] we have shown extensions for HMSCs, preemption, join, trigger composition, and several other MSc operators. [10] contains a detailed discussion of an extension of the algorithm in the context of broadcasting. There, the underlying idea is to introduce an explicit, hierarchical communication bus into the system’s software architecture; the broadcasting of messages is delegated from the individual components to the communication bus. This shifts
the burden of providing a scalable approach from the synthesis algorithm to the software architecture of the system under consideration. In this section we apply this principle also to the handling of preemption specifications; in addition, we discuss the relationship of trigger composition and join specifications to the synthesis task.

**Basic Algorithm** For sufficiently detailed MSC specifications we may try to exploit the collaboration information contained in them, and automatically derive implementations for the components participating in an interaction pattern. Over the past few years several different approaches and algorithms have been developed for this purpose (cf. [6] for an extensive list of references). In our approach we derive an automaton for an individual component specification from a given set of MSCs by successively applying four transformation steps: 1. projection of the given MSCs onto the component of interest, 2. normalization of the MSCs, i.e. adding missing start and end labels (state markers), and splitting MSCs with more than two labels at an intermediate label, 3. transformation into an automaton by identifying the MSCs as transition paths, and by adding intermediate states accordingly, and 4. optimization of the resulting automata. This synthesis algorithm works fully automatically for causal MSCs [3], and can handle choice, repetition, and concurrency/interleaving [6]. Because the algorithm is based on syntactic manipulation of the given MSCs it is oblivious to the underlying MSC semantics – as long as the semantics of the target component model matches the one used for the MSCs serving as input to the algorithm.

**Broadcasting** Our basic synthesis algorithm deals only with p2p message exchange. One implementation strategy for broadcasting that allows us to reuse the algorithm without modifications is to treat every broadcasting message as an abbreviation of the concurrent sending of the same message to all target components. After this syntactic substitution the original algorithm can be invoked, providing a correct implementation. This approach, however, has the disadvantage that each broadcast message induces a combinatorial explosion of the number of states in the sender’s state machine – if the concurrency is resolved by means of nondeterministic choice. Another approach would be to directly select an automaton model providing support for broadcasting, such as statecharts. This is unattractive, however, in the early stages of system development, as it commits the developer to a design- and implementation oriented form of specification right from the start. Our solution is to make the broadcasting explicit in the target system architecture. We change the system structure from Fig. 1 to the one depicted in Fig. 5; the newly added component Broadcaster receives broadcasting messages from CM, and forwards them to all other components. As a consequence in the state automaton for CM we only need one transition for sending a broadcasting message. In particular, we can again reuse the basic synthesis algorithm without the problem of incurring cluttered automata for the senders of broadcasting messages. A second advantage of this approach is that it directly supports hierarchical component refinement, and thus scales nicely. In [10] we have introduced an architectural pattern for broadcasting architectures, targeted at component models based on p2p communication, such as the one we alluded to in Sect. 2, as well as the channel- and port-based models of ROOM [13] and UML-RT [14]. In such hierarchical component models the explicit broadcaster component can also be introduced as a sub-component of CM, thus hiding it from CM’s environment.

**Preemption** In dealing with preemption a similar situation as described for broadcasting arises. In the CTAS case study, for instance, instead of cluttering CM’s automaton by having it react to all possible combinations of incoming “yes” and “no” messages from its clients we propose to add an arbitrating system component receiving all replies, and forwarding the overall result “all yes” or “some no” to CM (cf. Fig. 5). Again, this dramatically reduces the state model for CM while adding only little complexity to the system’s structure. If desired this arbiter can also be hidden inside of CM as part of a hierarchical component model.
Fig. 5 also shows a sketch of the automaton resulting from application of the basic synthesis algorithm after the changes in system structure (and corresponding changes to the MSCs, mirroring the added components and modified channels) are carried out. The transition labeled “1”, for instance, corresponds to sending the message $\text{cb} \cdot \text{CTAS.GET.NEW.WTHR}$ from $\text{CM}$ to $\text{Broadcaster}$, from where it will be forwarded to the clients. The transition labeled “5” corresponds to receiving message $\text{arc} \cdot \text{some_no}$ by $\text{CM}$ from $\text{Arbiter}$ – after the arbiter has determined that at least one client has responded with a “no” message; this results in handling of the exception.

**Trigger Composition** Trigger compositions express global liveness properties that in general cannot be established on a local component level. In this sense, trigger compositions specified using MSCs express proof obligations that can be fed, for instance, into model checking tools. The typical implementation strategy for detecting the absence of liveness at runtime is the use of timer components and to have components react to timeout events, which requires no changes to the basic synthesis algorithm.

**Join** The join of two MSCs corresponds to building the cross-product of the resulting automata resulting from application of the basic synthesis algorithm to each of the operand MSCs. For the details of this construction and its implications we refer the reader to [6].

### 4. Related Work

Suggestions in the literature for MSC dialects abound; [6] contains an extensive list of references. [4, 5] defines the standard syntax and semantics for MSC-96. The 2.0 version of the UML[11] has a new interaction model based on MSCs; previous versions adopted a much less powerful notation. LSCs [1] distinguish several interpretations for MSCs (similar to our discussion in [6, 7, 8]), which allow, in particular, the definition of liveness properties and “anti-scenarios”.

None of the above, however, provides operators for treating broadcasting and overlapping scenarios explicitly. By providing the 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 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 [6] 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

MSCs and similar notations provide a widely adopted means for capturing interaction patterns in the form of scenarios. MSCs, however, fall short regarding support for important modeling aspects, including broadcasting, preemption specifications, progress/liveness specifications, and overlapping interaction patterns.

By means of use cases from the CTAS case study we have demonstrated the utility of MSC operators addressing these deficits; we have also alluded to their semantic embedding. In addition, we have discussed extensions to our basic algorithm for component synthesis from MSC specifications. By slight modifications of the system architecture, the task of synthesizing state machines is significantly simplified and more scalable, despite a more expressive MSC syntax.

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; this will enable investigation of the tradeoff between the added expressiveness and the practical utility of the extended MSC notation.
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