FROM MSCS TO STATECHARTS

Ingolf Krüger, Radu Grosu,
Peter Scholz, Manfred Broy[1]

We present a first step towards a seamless integration of MSCs into the system development process. In particular, we show how scenario-based system requirements, captured in the early system analysis phase using MSCs, are translated into state-based description techniques like Statecharts. To this end, we sketch a schematic integration of MSCs and Statecharts.

1. Introduction

Message Sequence Charts (MSCs) have gained wide acceptance for scenario-based specifications of component behavior (see, for instance, [IT96, BMR+96, Rat97, BHKS97]). In particular, they have proven useful in the requirements capture phase of the software development process. However, up to now, most development methods do not assign a precise meaning to this graphical description technique. Moreover, MSCs are often used only for documentation purposes, and are not seamlessly integrated into the development process. Therefore, a lot of their potentials are usually left unexploited. Besides the interaction oriented system view provided by MSCs, state-based specification languages, such as Statecharts, are of particular importance in later development stages. For Statecharts, concepts for refinement, verification, and code generation have recently been developed. Aiming at a seamless development process, an integration of both description techniques seems necessary and promising. As a step towards this goal we show how to derive Statecharts from MSCs schematically. To that end, we discuss various interpretations of MSCs with respect to their applicability within the development process we target at, and show how to translate a subset of MSC specifications into Statecharts models.

1.1. MSCs and Development Processes

Typical software development processes of today [V97, Boe86, Boe88, Roy70] suggest to develop the intended product not in one single step, but in successive devel-

[1] Institut für Informatik, Technische Universität München, Arcisstr. 21, D-80290 München, Germany; email: {kruegeri,grosu,scholzp,broy}@in.tum.de

opment phases. Four common phases across different development processes are requirements capture, design, validation, and implementation.

MSCs have gained wide acceptance particularly during requirements capture 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. In later phases, the most common use of MSCs today is for documentation or explanatory purposes. 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. As a first step towards achieving such an integration, we consider the combination of MSCs and state-based specification techniques, such as Statecharts. The latter are mainly employed during design, and, increasingly often, during validation and implementation of distributed embedded systems. In particular, we aim at deriving Statecharts from a set of given MSCs such that the former exhibit the same interaction behavior as the latter. The resulting Statecharts can then be refined towards an implementation.

1.2. A closer look at MSCs

One of the major obstacles at increasing the integration of MSCs into the overall software development process is that their semantics with respect to other system views, such as the component distribution or state transition views, is either left unspecified, or restricted to a purely scenario-based specification style. The potential for confusion arising in both cases manifests itself in the slightly adapted saying “Everybody understands MSCs, but everybody understands them differently”.

Traditionally, the graphical syntax of most MSC dialects provides the following basic symbols: labeled axes, representing part of a component's existence, and labeled arrows, representing message exchange, directed from the sender to the recipient of a message. For simple scenarios these symbols already suffice. Frequently, additional constructs that increase the expressiveness of the notation are added. Examples of such constructs are sequential and parallel composition, choice, repetition, and even hierarchic decomposition (see, for instance, the graphical syntaxes discussed in [IT96, Rat97, BMR+96]). Most of these turn out to be essential when transiting from scenarios to more elaborate system descriptions. To perform this transition, we have to state precisely what we mean when we draw an MSC. To show that there is indeed a lot of room for discussion and, therefore, (mis)interpretation, we raise the following questions. First, what does an MSC mean with respect to the set of components within the system under development? Answering this question results in three different interpretations: the system consists exactly, at most, or at least of the components referenced in the MSC. Second, how “complete” is the information we obtain from a given MSC with respect to the components involved and the messages they exchange? The interaction between the components referenced in the MSC is determined either completely, i.e. there do not occur other message sequences between the components, or partially, i.e. other message sequences are explicitly allowed. Third, at what time instants do successive message exchanges occur? This addresses a problem that is of

particular importance when designing reactive systems: may, for instance, two successive message exchanges occur during the same system step? Considering the interpretations obtained by answering these questions yields a classification of possible MSC semantics. Each element of this classification can be investigated for its usefulness in enabling a methodical transition from MSCs to state-based description techniques. To base this discussion on more solid grounds we formalize our notion of systems and interaction.

1.3. Statecharts

Statecharts are a graphical description technique for the state-based, behavioral specification of control-oriented systems [Har87]. The language was developed for the description of reactive systems. Statecharts combine the operational notions of Mealy machines (sequential automata) with graphical structuring mechanisms to concisely describe large state spaces.

The main concept of Statecharts is that of sequential automata. These automata consist of states and transitions. An automaton's state denotes a section of a complex state of a reactive system. Transitions connect consecutive system states. A transition is labeled with a pair, consisting of the condition that must be fulfilled in order to trigger the transition and an action that specifies more detailed system behavior to be executed when the transition is performed.

To enable description of practically relevant systems, these automata can be composed to larger specifications using two principal structuring mechanisms: parallel composition and hierarchic decomposition. Here, the basic assumption is that automata composed in parallel proceed in lock-step with respect to a common clock. Without any further assumptions, these automata do not interact at all. However, if specified by the user, they also can exchange messages in order to mutually influence each other's behavior. Reactive systems may have complex system states. Hence, single automaton states may not be appropriate for a detailed description of these systems, and more elaborate techniques are needed. Statecharts allow the designer to structure single states of sequential automata by hierarchic decomposition. The Statechart that describes the system's behavior in a specific state in more detail is simply (graphically) inserted into this state. The behavior of a hierarchically decomposed Statechart is comparable with the one of procedure calls in an imperative programming language.

However, a specification formalism is of limited value without its integration into an appropriate system development process. What we need is to give guidelines how to develop a concrete implementation or realization from an abstract system specification. This transition from an abstract specification to a concrete implementation covers a number of different development steps such as specification, verification, efficient compilation, and refinement. For the Statecharts dialect μ-Charts, such a development process has been presented in [Sch98a].

1.4. Integrating MSCs and Statecharts

In the development process using Statecharts as sketched above, we factored out the requirements engineering phase. Hence, there is still a gap between this phase and the

state-based description of a reactive system with Statecharts. This gap can be filled by applying techniques that are based on MSCs. In contrast to Statecharts, these provide a possibility for event- or scenario-based development of a system in its early design phase. In order to get a seamless development process that also covers requirements engineering, it is interesting to develop an appropriate MSC-like description technique that is based on the same semantic model as Statecharts and therefore can be automatically translated to an equivalent Statechart specification. Furthermore, these MSC-like techniques can serve to represent counter-examples and witnesses from model checking results, or simply as a front end for the input language of an appropriate model checker. Finally, MSCs can be used to restrict communication between Mealy machines combined in parallel to only those signal interactions that are free of causality errors [Sch98b], which is one of the major problems when specifying larger systems with Statecharts.

Integrating MSCs and Statecharts to form a seamless development process, a number of interesting questions have to be solved. First of all, both languages have to be unified on the semantic level. To this end, we define their semantics by means of a relational stream semantics. The stream semantics of one component, specified either as an MSC or as a Statechart reflects its, potentially non-deterministic, input/output behavior. Dealing with a non-deterministic semantics has the advantage that we can treat abstract specifications and concrete, implementation-level descriptions with the same mathematical machinery. Furthermore, this approach makes it possible to allow mixed MSC and Statechart specifications. This is a prerequisite to capture the behavior also of those systems under development, where different system parts reside in different development phases. Of course, a common, homogenous semantic model further eases specification of complex systems: not only can components in different development phases be described with different techniques, but also those in identical phases. Thus, it is possible that some system parts are merely specified in MSCs and others just in Statecharts.

2. Translating MSCs into Statecharts

In this section we discuss a general procedure for deriving state-based component descriptions from a set of MSCs. To that end, we first sketch the underlying semantic framework that we use to achieve a smooth integration of Statecharts and MSCs (Section 2.1). We give an overview of our translation scheme in Section 2.2, and present an example of its application in Section 2.3.

2.1. Semantic Model

As we have mentioned already in Section 1, there is a lot of freedom in interpreting MSCs. Here, we introduce a simple, yet expressive formal framework, on top of which we can both discuss possible MSC semantics, and establish an integration of MSCs and Statecharts.

The systems we are interested in consist of a set of components, and a set of directed channels. Components that have to communicate must be connected by channels. Each component operates by reading input on its input channels, calculating its

output, and writing the output to its output channels. Furthermore, we assume that the system is driven by some global, discrete clock. This is a quite natural assumption considering the current hardware technology and it allows us to determine exactly at what time unit a message occurs on a given channel. We call the sequence of messages passed along a channel over time a channel history. Our basic assumption here is that communication happens message-asynchronously: the sender of a message is never blocked by the receiver of the message. A simple strategy to assure this kind of communication is buffering messages.

Given a system specification *Sys* and an MSC *msc*, we can now define how the two relate. Each such relationship defines one possible interpretation of *msc*. There are two interesting questions concerning the semantics of any given MSC: what sequences of messages does it represent, and when – with respect to a complete execution of the system under design – do they occur. We assume that we know the set of sequences defined by an MSC so we can focus on answering the second question. Suppose the system *Sys* is characterized by a set of channel histories V and that *msc*, msc_0 and msc_1 are MSCs. We distinguish the following interpretations of the MSCs with respect to *Sys*:

exact:	the interaction sequences specified by *msc* occur exactly once in all $v \in V$, and no other interactions are possible;
existential:	there are elements $v \in V$ that exhibit the interaction sequences specified by *msc*;
universal:	all elements $v \in V$ exhibit the interaction sequences specified by *msc*;
trigger condition:	we say that msc_0 triggers msc_1, if, after an interaction sequence as specified by msc_0 has occurred in a valuation $v \in V$, an interaction sequence as specified by msc_1 will also occur in v within a finite amount of time;
negation:	using logical negation, we obtain the dual interpretations of the ones given above.

Each of these interpretations has its place within the software development process. During requirements capture each of them is of value and in use. The exact interpretation is the one we use for the transition from MSCs to Statecharts. For testing, and documentation the most prominent use of MSCs today is under the existential, and universal interpretation, together with their negations. The bottom line of this discussion is that in whatever phase of the development process we employ MSCs, we should state clearly what interpretation we have in mind.

2.2. Roadmap for the translation procedure

In the previous section we have explained the semantic model for the systems we are interested in, together with a number of possible interpretations for MSCs in that model. Here, we select the exact interpretation in a slightly modified form as the basis for our translation procedure from MSCs to Statecharts.

The basis for the translation is the procedure explained in detail in [BGK98]. 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 occurring in the system under development. The procedure we suggest makes use of information on the component's control/data-state provided by the designer, for instance, via the condition symbols found in the MSC standard [IT96, IT98]. Each condition will be translated into a corresponding automaton state. We assume further that we try to obtain an automaton for exactly one of the components, say c, occurring in the MSCs. The second input we expect, besides the set of MSCs, is the name of the starting state for c's automaton. The procedure for obtaining that automaton consists of five successive phases: projection, normalization, transformation into an MSC-Automaton, transformation into an automaton, and optimization.

2.2.1. Projection

During the first phase we project each of the given MSCs onto the component c, i.e. we remove all other instance axes, as well as message arrows that neither start, nor end at c.

2.2.2. Normalization

Then, in the second phase, we normalize the projected MSCs. An MSC in our normal form consists of exactly two condition symbols and a (possibly empty) sequence of message arrows in between. To normalize an MSC that does not start or end with a condition symbol, we add the one representing the given initial state. We split MSCs that contain more than two conditions at any of the intermediate ones. Thus, we replace each such MSC by two new ones; the first of these is the original MSC up to (and including) the condition at which we split, the second one is the original MSC from that condition on.

2.2.3. Transformation into an MSC-Automaton

To obtain what we call an MSC-Automaton, we now identify the condition labels occurring in the resulting MSCs with states of the automaton for component c. Furthermore, we create a transition from state s_0 to state s_1 labeled by MSC M if and only if there is a normalized MSC M whose start and end conditions are labeled with s_0 and s_1, respectively. As the initial condition for this automaton we use the one provided as an input to the procedure. The result of this phase is an automaton that provides a high-level view on the states of the component we are interested in.

2.2.4. Transformation into an Automaton

In the fourth phase of our procedure we expand the transitions that are labeled with MSCs by intermediate states and transitions. We replace all transitions that are labeled with an MSC containing only conditions by an ε-transition. Let M be any other MSC label. It defines $k > 0$ communication actions that we denote by (d_1, n_1, c_1), ..., (d_k, n_k, c_k), recorded in the order of the corresponding arrows of M. Here d_i, n_i, and c_i represent the direction of the message exchange ("!" for an outgoing, and "?" for an incoming arrow), the message label, and the name of the corresponding sender or receiver, respectively (for $1 \le i \le k$). For $k = 1$ we replace the transition labeled with M by one labeled with (d_1, n_1, c_1). For $k > 1$ we add k-1 new states, labeled by s_1 through s_{k-1}, as well as k new transitions to the automaton. The first transition starts in the state corresponding to the start condition of M, ends in s_1, and its label is (d_1, n_1, c_1). The k-

th transition, labeled with ($d_{k,}$ n_k, c_k), starts in the new state s_{k-1}, and ends in the state corresponding to the final condition of M . All other transitions start in state s_{i-1}, end in state s_i, and their label is (d_i, $n_{i,}$ c_i), for $1 < i < k$. After applying these steps to all transitions labeled with MSCs we obtain a nondeterministic automaton with input/output actions.

2.2.5. Optimization

To optimize the resulting automaton we may now use standard algorithms from automata theory; examples of such algorithms are the elimination of ε-transitions, the subset-construction, and the Myhill-Nerode construction for minimization (cf. [HU93]). Other optimizations are motivated by the semantics of the underlying communication system. For Statecharts we might, for instance, condense transition sequences of the form "?, !, !, ..., !"[2] into a single transition labeled with "?/!!...!". The symbol "/" separates input and output actions that occur during the same time instant [Sch98a]. This corresponds to identifying the occurrence of a sequence of outputs with their triggering input.

The next section provides an example application of the procedure we have outlined here.

2.3. Example

As the running example we use a simplified specification of a central locking system (CLS) for cars. This example is inspired by a case study from automotive industry. The principal structure of the CLS is sketched in Figure 1; it gives an overview of the locking system of a two-door car.

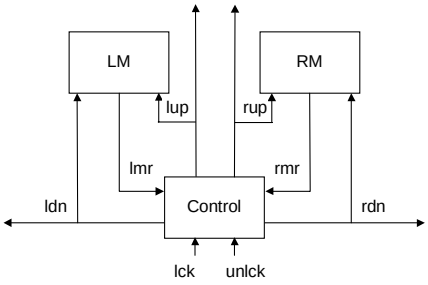

Figure 1: Central locking system – structure and signals

2.3.1. Informal Description

The CLS consists essentially of three main parts: the Control and the two door motors LM (short for "left motor") and RM (short for "right motor"). These parts are composed in parallel. In this system structure, we also describe the signal flow between different components by arrows, which connect output and input interfaces of possibly communicating components.

[2]Here we consider the direction of the communication only.

The default configuration of the system is that all doors are unlocked (UNLD) and both motors are OFF. The Control is the central component of the central locking system.

The driver can unlock or lock the doors either from outside by turning the key or from inside by pressing a button. Locking and unlocking the doors leads to complex signal interactions. Both actions generate the external signals lck (for "locking") or unlck (for "unlocking"). The Control generates the internal signals ldn and rdn and changes to its locking configuration LCKD.

Instantaneously, influenced by ldn and rdn, respectively, both motors begin to lock the doors by entering their DWN configurations. As the speeds of the motors depend on external influences like their temperature, each motor either needs one or two time units to finish the lowering process. Only when both motors have sent their ready messages lmr (for "left motor ready") and rmr ("right motor ready"), the Control enters its LCKD configuration.

2.3.2. Scenario-based View

The MSCs of Figure 2 illustrate most of the requirements stated informally in the previous paragraphs; for reasons of brevity we have omitted the handling of simultaneity and of the possibly different speeds of components.

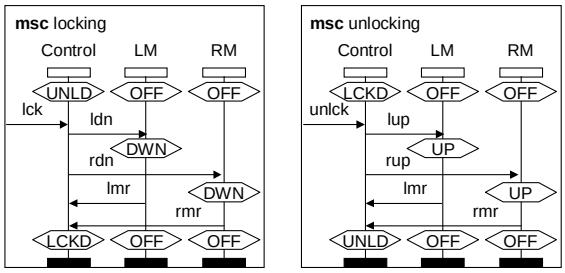

Figure 2: Central locking system – scenarios for locking and unlocking

Consider, for instance, MSC locking, which depicts the behavior of the Control, LM, and RM during the locking operation of the CLS. The angular boxes in MSCs denote the configurations or states that the corresponding component is in, according to the informal description above. The depicted interactions also proceed along the lines of the informal requirements capture. If the Control is in state UNLD and receives the lck signal, it emits the lup and rup messages to the two motors. After it has received both the lmr, and the rmr signals it changes to the LCKD state. The translation of the other components' behavior, as well as the translation of the second scenario (see MSC unlocking) are equally direct.

2.3.3. Translation into Statecharts

In this section we apply the translation procedure as described in Section 2.2 to the MSCs depicted in Figure 2. We illustrate the results of the individual phases of the procedure using the Control component.

In the first step we project the two MSCs onto the axes that represent Control. The resulting MSCs (see Figure 3, left) are already in normal form. Note that we would

have to split the projections of LM and RM at the intermediate UP and DWN conditions to obtain normal forms for these components. We derive the MSC-automaton for Control as it appears in Figure 3, right, because MSCs LC and UC take that component from state UNLD to LCKD, and from LCKD to UNLD, respectively.

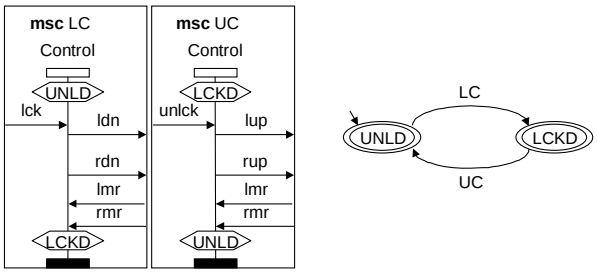

Figure 3: Projected MSCs for component Control and corresponding MSC-Automaton

By expanding the MSC-labeled transitions in the fourth phase of the procedure, we obtain the automaton depicted in Figure 4. Note that we have omitted the receiver in the transition labels because in the broadcasting communication model of Statecharts this information is redundant.

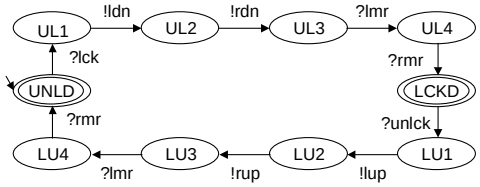

Figure 4: Automaton for component Control

Using the optimization suggested in Section 2.2, and combining the result with the result of deriving also the automata for LM and RM, we obtain the Statechart depicted in Figure 5.

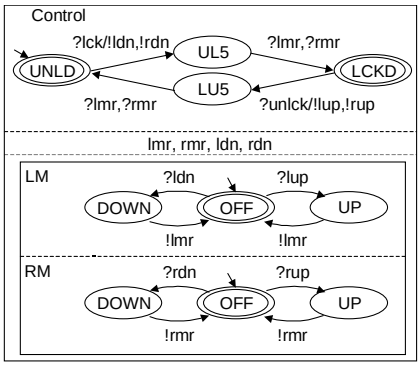

Figure 5: Overall Statechart for component Control and both motors

Having translated the set of MSCs into a Statechart, we may further refine the system under development towards the final implementation as shown in [Sch98b].

3. Conclusion

In this paper we showed how to translate scenario-based system requirements into state-based system designs.

Although we presented the translation in a time-synchronous communication setting, it is actually independent from the underlying communication mechanism. The reason is that we associate a separate automaton to each component occurring in the MSCs. Communication is captured by parallel composition of automata.

The composition can be adapted to the communication style chosen for MSCs. If this communication is time-synchronous, one has to define a time-synchronous parallel composition of automata (as done in this paper). If the communication is time-asynchronous, one has to define a time-asynchronous parallel composition. Similarly, one can consider a communication mechanism using one, two or several communication buffers. In all these cases, our translation remains the same. One only needs to change the automata composition.

Acknowledgments

Our work was partially supported with funds of the Deutsche Forschungsgemeinschaft under the Leibniz program, by Siemens-Nixdorf within project SysLab, and by the Bayerische Forschungsstiftung. We are grateful to the participants of DIPES'98 for interesting discussions about this subject.

References

[BGK98] Manfred Broy, Radu Grosu, and Ingolf Krüger. Deutsche Patentanmeldung, Aktenzeichen 198 37 871.8, 1998.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. *Using extended event traces to describe communication in software architectures*. In Asia-Pacific Software Engineering Conference and International Computer Science Conference, Hong Kong. IEEE Computer Society, 1997.

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

[Boe86] B. Boehm. *A spiral model of software development and enhancement*. ACM SIGSOFT, pages 14 – 24, August 1986.

[Boe88] B. Boehm. *A spiral model of software development and enhancement*. IEEE Computer, pages 61 – 72, May 1988.

[Har87] D. Harel. Statecharts: *A Visual Formalism for Complex Systems*. Science of Computer Programming, 8:231–274, 1987.

[HU93] J. E. Hopcroft and J. D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison Wesley, 2. korrigierter Nachdruck, 1993.

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

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

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

[Roy70] W.W. Royce. *Managing the development of large software systems*. IEEE WESCON, pages 1 – 9, August 1970.

[Sch98a] P. Scholz. *A Refinement Calculus for Statecharts*. In Proceedings of the ETAPS/FASE'98, Lisbon (Portugal), March 30 - April 03, 1998, volume 1382 of Lecture Notes in Computer Science. Springer, 1998.

[Sch98b] P. Scholz. *Design of Reactive Systems and their Distributed Implementation with Statecharts*. Ph.D. thesis, Technische Universität München, 1998.

[V97] Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell. Allgemeiner Umdruck Nr. 250/1. Juni 1997, BWB IT I5.