

Structured Formalization of Service-Oriented Specifications

Markus Herrmannsdoerfer*, Sabine Rittmann*, Bernhard Schätz
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany

E-mail: {herrmama, rittmann, schaeetz}@in.tum.de

Abstract

We present and formalize concepts for the structured specification of system behavior based on services. Services are patterns of interaction between reactive components, thus supporting especially the development of distributed systems. We introduce a core set of concepts for the specification of basic services as well as for the combination of those to complex functionality. The result is an expressive mechanism for structured behavioral specifications. While basically independent from specific notations, we demonstrate the application of those concepts using (High-level) Message Sequence Charts and Mode Diagrams for the specification of services and their combination. We illustrate the approach using a simple telephone system as a running example.

1 Introduction

Today we face a trend toward increasingly complex computer-based systems. Hereby, the complexity stems from different sources. For example, today's systems usually provide a great variety of sophisticated and intermingled user functionality. Furthermore, they are distributed over heterogeneous hardware platforms. In order to handle this complexity, new techniques for functionality descriptions are needed, providing a set of concepts with precisely defined semantics in order to avoid misunderstandings and ease tool-supported formal reasoning.

The aim of such formally founded concepts, as introduced in the following, is to support a model-based, domain-independent development process using precisely defined notational techniques. The main idea is to extend model-based development to the analysis phase in order to specify the usage behavior consistently. The center pieces

of the presented approach are *services* which are defined as interaction patterns between system entities. Thus, services describe *partial pieces of the system behavior*.¹ By combining the single services, we obtain more comprehensive behaviors up to the overall system behavior.

As services are *partial behaviors* (as defined in [1]) focusing on the interaction between components, sequence diagrams are a suitable notational technique for their specification. Therefore, we suggest to specify services by means of Message Sequence Charts [5]. The combination of services can be specified by High-level Message Sequence Charts [5] providing a sequence-oriented view onto the system, and so-called Mode Diagrams providing a more state-oriented view onto the system.

1.1 Contributions

In this paper we make the following contributions: We introduce fundamental concepts for the modular specification and combination of the system functionality.² The main idea is to first specify atomic observations and then to combine them to more comprehensive functionalities. A set of sequences of observations makes up the system functionality. We show that three basic concepts are powerful enough to specify the system behavior: *atomic observations*, *combination operators (conjunction and disjunction)*, and *abstraction*.

We demonstrate the application of these fundamental concepts by formalizing specific notational techniques, namely an MSC dialect with HMSCs [5] and Mode Diagrams. To that end, we show how the basic syntax constructs are mapped to the basic concepts. Due to this mapping, we observe that the same concepts occur in different notational techniques and on different levels of granularity. Furthermore, we show how easy syntactic extensions ("syntactic sugar") can be introduced.

¹Please note that in our context the definition of the term service differs from the usage of the term in the field of service-oriented computing.

²Preparatory work on this topic can be found in [10].

*This work is partially funded by the Deutsche Forschungsgemeinschaft (DFG) under grant number BR 887/19-2

The basic concepts can be directly translated into trace sets using *weak second order monadic structures with one successor* (WS1S), which is the formalization supported by the model checker *Mona* [7]. Thus, consistency and refinement checks can be performed on the service specifications.

1.2 Related Work

The approach presented here provides a formalization for the modular description of functionalities or features, supporting the compositional construction of complex behavior from elementary functions using a set of basic composition forms.

Similar to our approach, Jackson and Zave suggest to construct complex systems – in the form of virtual architectures for telecommunication systems – by specifying modular behaviors in features and combining them [6].³ However, features are only combined sequentially in a pipe-and-filter architecture. While providing a useful paradigm for the design of systems in the telecommunication domain, especially for the specification and analysis more powerful ways to combine modular behaviors are needed, as provided by our approach.

In [9], Krüger et al. also define behavior by interaction patterns (services) and specify it by (H)MSCs, which – transformed into an automata dialect (cf. [8]) – can be model checked, etc. Similarly, Damm and Harel specify system functionality by means of a Live Sequence Charts (LSCs), extending MSCs, e.g., to distinguish between possible and necessary behavior, and also providing model checking support [3]. However, these approaches rather focus on the formalization of specific notational techniques – (H)MSCs and LCSs, resp. – while here fundamental concepts underlying sequence-based functional specifications are identified and formalized, suitable for a whole class of notational techniques.

Broy et al. introduce formal theories for structured behavioral specifications based on streams, also focusing on concepts rather than specific notations and using traces to specify system behavior [1, 2]. Specifications can be combined to describe more complex behavior; components – total behaviors used in [2] – and services – partial behaviors used in [1] – can be decomposed hierarchically. However, the authors do not distinguish between data flow (describing the values observed and controlled by a function) and control flow (describing the activation and deactivation of functions), thus lacking some forms of composition presented here.

As mentioned above, Schätz introduces a similar formalization for modular specifications of functionality with control and data flow interfaces, also providing model checking

³In addition, the features are added to a base specification, namely the plain old telephone system (POTS).

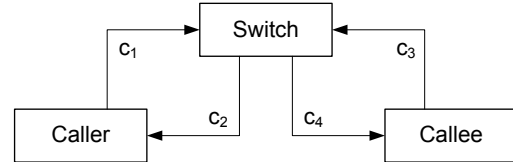


Figure 1. Component architecture of the running example

support based on *Mona* [10]. However, unlike here, elementary functions are described in a state transition manner not suited for the formalization of MSC-like notations.

1.3 Outline

The paper is structured as follows: In Section 2, we give the textual requirements and the graphical specification of a small running example. To that end, we make use of an MSC dialect with HMSCs and Mode Diagrams whose syntax is informally described. In Section 3, we introduce the fundamental concepts for the structured, service-oriented specification of the system behavior. We show its application by formalizing basic (H)MSC constructs (Section 4) and syntactic extensions. In Section 5 we shortly describe how the fundamental concepts serve well as a basis for model checking. Finally, we summarize our approach and provide an outlook on interesting future research topics in Section 6.

2 Running Example: Simple Telephone System

In order to explain the syntax and semantics of our notational techniques, we use a running example: a simplified version of a telephone system. In this section, we state the functional requirements textually. Furthermore, we provide a specification of the system behavior by means of services. We specify the basic services with an MSC dialect, and their interplay with HMSCs and Mode Diagrams. The syntax of these notational techniques is explained informally. In Section 4, we will define the semantics of the notational techniques formally.

We only consider telephone calls between one Caller and one Callee. They are connected with each other through channels via a Switch. The Caller initiates the call and the Callee answers it. In the following, we assume a component architecture as shown in Figure 1. Figure 2 shows the behavior of our running example as a Mode Diagram. We consider a telephone system which provides the possibility to subscribe and unsubscribe to an answering machine (AM). Depending on whether the AM is activated or deacti-

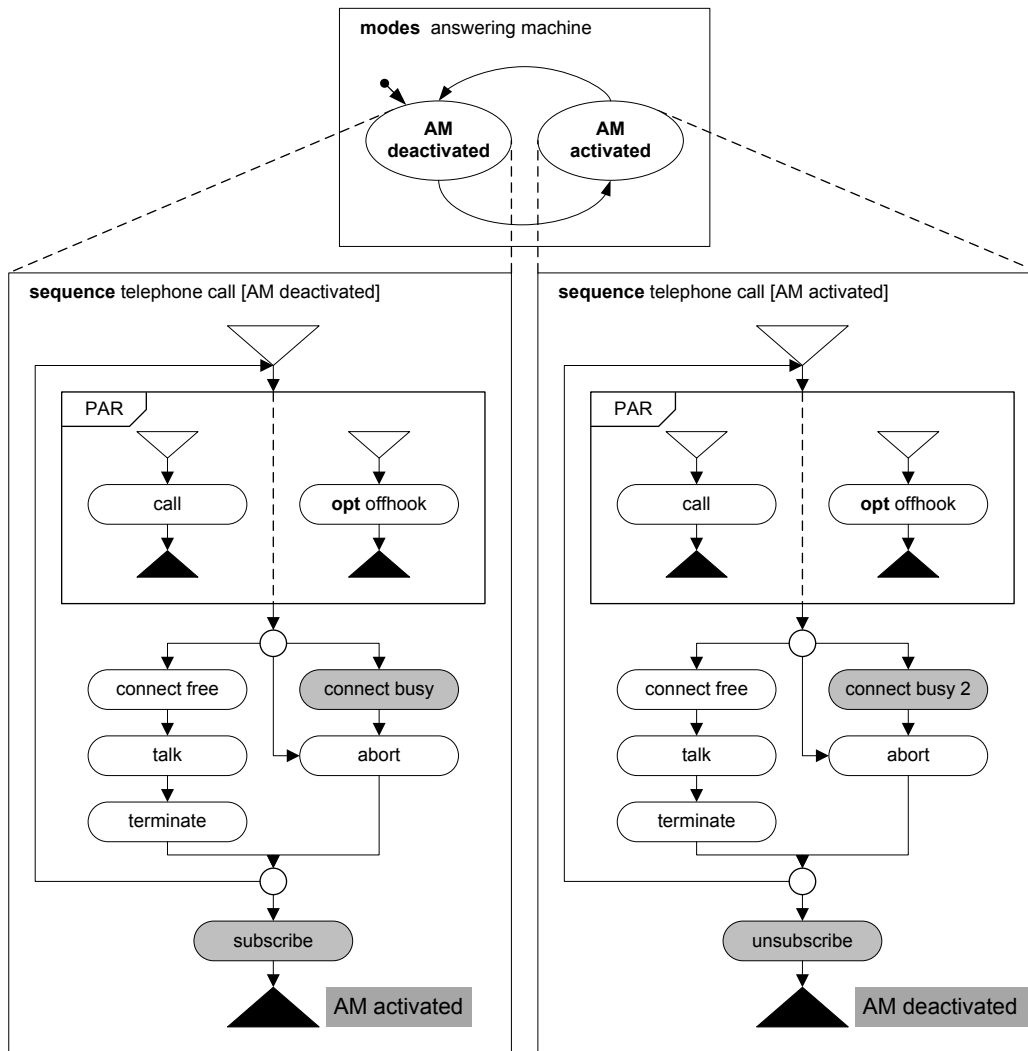


Figure 2. Typical procedure of a telephone call with and without an answering machine (Mode Diagram and HMSCs)

vated, the system can be seen to be in a different *mode*.⁴ For each mode we get another behavior, i. e. a possible procedure of a telephone call, which is specified by means of an HMSC respectively. MSC references contain service specifications (see later, Figure 3). By using Mode Diagrams, we have a state-oriented view onto the system; HMSCs provide a sequence-oriented view.

In both modes we observe the following behavior⁵: We assume that both telephones are not busy, when the system is set up and omit the specification of the initialization of the system. The Caller initiates a call (service call). In par-

⁴The approach make use of modes to structure the specification of the system behavior.

⁵The differences between the HMSCs of both modes are highlighted in Figure 2.

allel, the Callee might hook off the telephone independently from the Caller's action and is then busy (service offhook). Depending on the status of the Callee, namely if they are busy or not, we face the following possibilities specified by the alternative paths through the HMSC of Figure 2. If the Callee is not busy, a connection between the Caller and the Callee can be established (services connect free, talk, and terminate). If the Callee is busy and the AM is not activated (HMSC within the mode AM deactivated in Figure 2), no connection can be established (service connect busy) and the Caller has to abort the telephone call (service abort). If the Callee is busy and the AM is activated (HMSC within the mode AM activated), a message can be recorded (service connect busy 2). Another possibility in both scenarios is that the Caller changes their mind and immediately

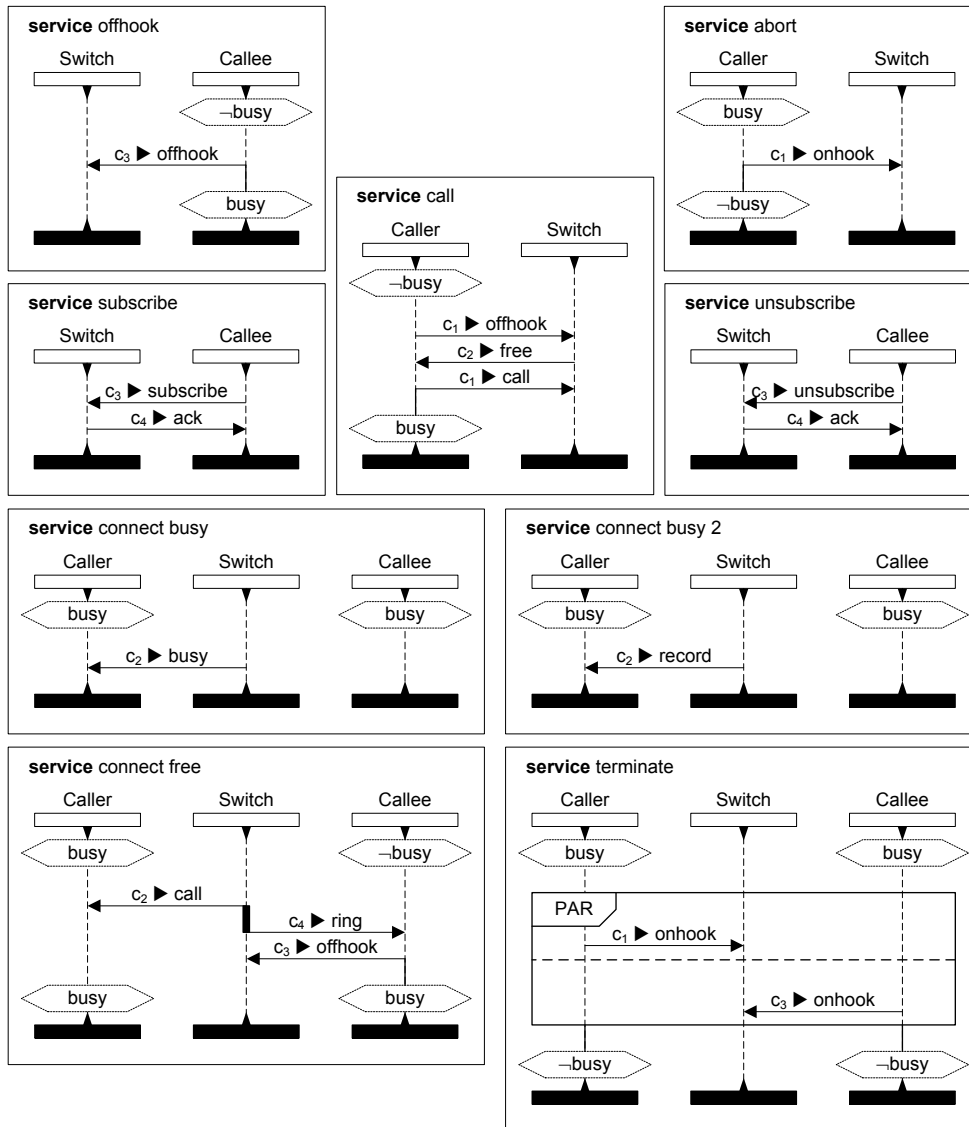


Figure 3. Specification of telephone services (MSCs)

aborts the telephone call (abort). At the end of a telephone call, the activation of the answering machine feature may be changed (services unsubscribe and subscribe).

Figure 3 contains the MSC specifications of the services which are referenced in the HMSCs of Figure 2.⁶ Here we provide a short and informal description of our MSC syntax. Axes represent (distributed) components. Hexagons mark conditions representing the states of the components. Arrows denote communication between components and are labeled by $c_1 \triangleright$ offhook indicating that message offhook is sent on channel c_1 . We only allow at most one message to be sent on a channel in one time tick. Furthermore, we

⁶Due to space constraints, the specification of the service talk is omitted in Figure 3.

assume a global, discrete clock which divides the time line in equidistant time intervals. Axes can be formatted differently: *Solid lines* between conditions, between conditions and messages, or between messages indicate that exactly one time tick elapses in between. *Dashed lines* specify that at least one time tick can occur in between. *Bold lines* represent the parallel occurrence of conditions or messages within the same time tick. However, both rectangular start and end shapes of an axis do not have any meaning.

For example, consider the service call of Figure 3: At the beginning, the Caller fulfills the condition \neg busy. After an arbitrarily large amount of time (represented by the dashed line), the Caller might hook off the telephone (offhook). Again, after at least one time tick, the Switch responds the

free message. The Caller then dials the number (call). Exactly one time tick after having dialed the number, the Caller fulfills the busy condition. In the service connect free (see also Figure 3), the bold line specifies that the messages call and ring are sent in parallel, i. e. within the same time tick. Note that the ordering of the messages is not important as they are sent in parallel.

3 Basic Concepts

Since services are intended for the modular specification of functionality, a formalization similar to [4] is used to introduce a set Fun of functional descriptions as well as its interpretation as semantical basis. However, in contrast to the former, this approach focuses on the formalization of observation sequences rather than transition systems; therefore it supports the description of functions with their partially defined behavior, especially allowing the introduction of new partiality by simultaneous combination as defined in Section 3.5. In the following, Fun corresponds to the set of function terms, starting from basic functions and using operators to form more complex descriptions.

3.1 Basics

The structural aspects of a function are defined by its variables Var – used to transfer signal values between the function and its environment and to represent local control states – as well as its control locations Loc – used to transfer execution control between the function and its environment. To describe the behavior of a function, we use the concepts

State: A state $s \in \overline{Var} = Var \rightarrow Val$ maps variables to their current values.

Observation: An observation is a triple $(a, p_1 \circ s_1 \circ s_2 \circ \dots \circ s_n \circ p_2, b)$ consisting of a finite sequence $s_1 \circ s_2 \circ \dots \circ s_n$ of states corresponding to an execution starting at location a (at which the precondition p_1 holds), changing variables according to $s_1 \circ s_2 \circ \dots \circ s_n$, and ending at location b (at which the postcondition p_2 holds). Since in the following only continuous functions are introduced, a restriction to finite observations is sufficient.

Behavior: The behavior of a function is the set Obs of all its observations.

For a state $s : Var \rightarrow Val$ with $Var' \subseteq Var$ we use notation $s \upharpoonright Var'$ for restrictions $(s \upharpoonright Var')(v) = s(v)$ for all $v \in Var'$. This restriction is extended to sequences of states through point-wise application. For sequences r and t we use notation $r \circ t$ to describe the concatenation of r and t .

3.2 Basic Functions

The most basic function describes the observation of a single state. When entered through its start control location at which a certain precondition holds, it defines the values of its variables and terminates by exiting via its end control location. At the end control location, a postcondition is fulfilled. The left-hand side of Figure 4 shows such a basic function with variables c_1 and c_2 , start control location $start$, and end control location end . Its behavior is described by a labeled transition from $start$ to end , connected by labeled arrows representing the precondition $\neg busy$ and postcondition $busy$, and with a label consisting of the state $c_1 = offhook \wedge c_2 = nil$. This label states that variable c_1 carries signal $offhook$ while variable c_2 carries signal nil . The interface of this function is defined by $Var = \{c_1, c_2\}$, and its locations by $Loc = \{start, end\}$. As shown in Figure 4, the control flow interface (i.e., the set of interface locations) is described by circles connected to it; the data flow interface (i.e., the set of interface variables) is not explicitly modeled. The transition itself is described by an oval linked to the corresponding control locations with directed arrows representing the precondition and postcondition, respectively.

Abstracting from a concrete graphical representation, a basic function is described as the structure $(start, pre, act, post, end)$ with start control location $start$, end control location end , precondition label pre , postcondition label $post$, and action label act . Labels pre , act , and $post$ correspond to predicates with $pre, act, post : \overline{Var} \rightarrow \mathbb{B}$. The behavior of the basic function is the set of observations $(start, p_1 \circ s \circ p_2, end)$ with $pre(p_1) \wedge act(s) \wedge post(p_2)$.⁷

Consequently, the behavior of the above basic function is the set consisting of all observations $(start, p_1 \circ s \circ p_2, end)$ such that $s(c_1) = offhook$ as well as $s(c_2) = nil$ and $p_1(busy) = false$ and either $p_2(busy) = true$ or $p_2(busy) = false$.⁸ Note that, as shown in Figure 4 (middle sub-function of the second function, location mid), start and end locations need not be disjoint.

3.3 Disjunctive Combination

The behavior of a disjunctive combination of two functions corresponds to the behavior of either function. The second-left function in Figure 4 shows the disjunctive combination of three basic functions with interface locations

⁷Here, for reasons of simplicity a joint variable space for pre- and post-conditions as well as actions is used; a modularization in channel and state variables is straight-forward.

⁸In contrast to [10], no prefix closure of observations is used, leading to observations with explicit start point and end points, respectively, which is sufficient for the general use of MSCs, like e.g., in testing.

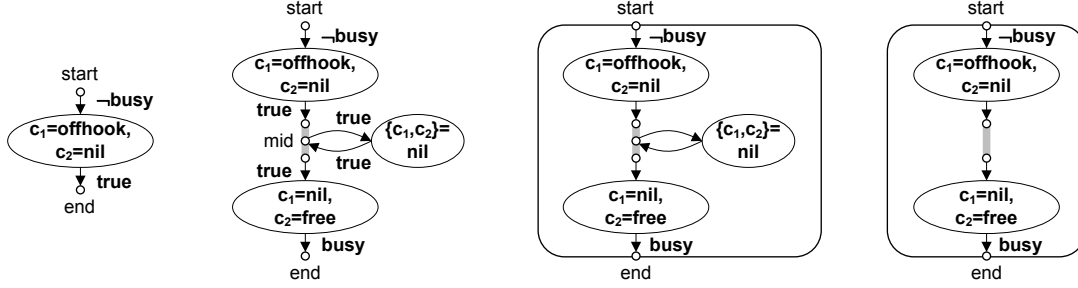


Figure 4. Basic Observations, Disjunctive Composition, Abstraction, and Conjunctive Composition

start and mid, mid and mid, as well as mid and end, respectively. It shares all the structural aspects of either function, and thus uses variables c_1 and c_2 , as well as control locations start, mid, and end. Formally, the disjunctive combination of two functions A and B results in a function described by $A + B$ that

- uses the variables of either function: $Var_{A+B} = Var_A \cup Var_B$
- accesses their control locations: $Loc_{A+B} = Loc_A \cup Loc_B$
- exhibits the behavior of either function: $(a, t, b) \in Obs_{A+B}$ if $(a, t \uparrow Var_A, b) \in Obs_A$ or $(a, t \uparrow Var_B, b) \in Obs_B$.

Intuitively, the combined function offers observations that can be entered and exited via one of its sub-functions. If the sub-functions share a common entry location, observations of either function starting at that entry location are possible; similarly, if they share a common exit location, observations ending at that common exit location are possible. Obviously, functions $A + B$ and $B + A$ are equivalent in the sense of having the same interface and behavior.

3.4 Abstraction

Abstraction allows to hide a location of a function, rendering the location inaccessible from the outside. At the same time, when reaching a hidden location the function does immediately continue its execution along a transition linked to the hidden location. In the second-right function in Figure 4, control location mid is hidden in the functionality described by its left neighbor. Formally, by hiding a location l from a function A we obtain a function described by $A \setminus l$ that

- uses the variables of A : $Var_{A \setminus l} = Var_A$
- accesses the control locations of A excluding l : $Loc_{A \setminus l} = Loc_A \setminus \{l\}$

- exhibits the behavior of A if entered/exited through locations excluding l and continuing execution at l : $(a, p_1 \circ s_1 \circ s_2 \circ \dots \circ s_n \circ p_{n+1}, b) \in Obs_{A \setminus l}$ if $(a, p_1 \circ s_1 \circ p_2, l), (l, p_n \circ s_n \circ p_{n+1}, b) \in Obs_A$ and $(l, p_i \circ s_i \circ p_{i+1}, l) \in Obs_A$ for $i = 2, \dots, n - 1$.

Obviously, $(S \setminus a) \setminus b$ and $(S \setminus b) \setminus a$ are equivalent in the sense of exhibiting the same interface and behavior. We write $A \setminus \{a, b\}$ for $(A \setminus a) \setminus b$.

3.5 Conjunctive Combination

Besides disjunctive combination, functions can be combined using conjunctive combination. The behavior of a conjunctive combination of two functions corresponds to the joint behavior of both functions. The right-most function in Figure 4 shows a function that, when composed conjunctively with the second-right function, results in the function itself. Formally, the simultaneous combination of two functions A and B results in a function described by $A \times B$ that

- uses the variables of each function: $Var_{A \times B} = Var_A \cup Var_B$
- accesses their *shared* control locations: $Loc_{A \times B} = Loc_A \cap Loc_B$
- exhibits the combined behavior of each function: $(a, t, b) \in Obs_{A \times B}$ if $(a, t \uparrow Var_A, b) \in Obs_A$ and $(a, t \uparrow Var_B, b) \in Obs_B$.

Intuitively, the combined functions offers observations that can be offered by both functions. Obviously, $A \times B$ and $B \times A$ are equivalent in the sense of exhibiting the same interface and behavior.

4 Formalization of MSCs and HMSCs

In this section, we provide a formalization of MSCs and HMSCs by using only the basic concepts from Section 3.

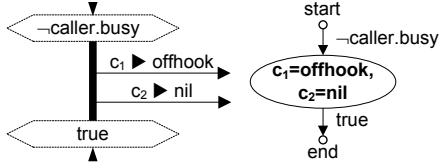


Figure 5. Basic elements and formalization by a function

For the formalization we use an axis-oriented scheme: For each axis within an MSC, we give its mapping onto a function (see Section 4.1); then the functions of all axes are combined to obtain the formalization of the whole MSC (see Section 4.2). Finally, we combine the MSC functions as specified by the HMSCs and Mode Diagrams (see Section 4.3).

A system consists of a set of logical components. These components are connected via directed channels, on which messages are asynchronously exchanged. Furthermore, a global system clock partitions the time into discrete ticks. The state space of a component consists of a number of variables, e. g. the boolean variable *caller.busy* for the component *Caller*. Each channel is formalized by a variable whose data type is the set of messages possibly sent over the channel. Note that the implicit consequence is that at most one message can be exchanged on a channel per time tick.

4.1 Mapping of an MSC Axis to a Function

We show how the MSC syntax constructs used in Section 2 are mapped onto functions using a simple precondition-action-postcondition scheme, and how those are combined to formalize an axis.

Basic Elements. We make use of basic elements as the building blocks of MSCs which are formalized by basic functions $(a, p_1 \circ s \circ p_2, b)$. They consist of the following parts: a start control location, a precondition, an action (e. g. the sending/receiving of a message), a postcondition, and an end control location. The start and end control locations are later used to glue together single behaviors. Figure 5 shows a basic element and its formalization by a basic function.

A condition part – either before or after the action – of a basic MSC (represented by a hexagon) is formalized by a suitable pre- or postcondition of the corresponding basic function $(start, pre, act, post, end)$. In case of the MSC in Figure 5, the precondition $pre \equiv \neg caller.busy$ corresponds to the condition before, while the postcondition $post \equiv true$ corresponds to the condition after the action. The action part of a basic MSC (represented by a bold solid life line

with communication arrows) is formalized by a suitable action of the corresponding basic function. In case of the MSC in Figure 5, the action $act \equiv c_1 = offhook \wedge c_2 = nil$ corresponds to the messages *offhook* and *nil* communicated over channels c_1 and c_2 .

In the following, for a more compact description, if the precondition (postcondition) in such a triple is true, it is not displayed in the graphical representation of the function. Similarly, for communications on channels with value *nil*, the corresponding sending/receiving arrows are not displayed in the graphical representation. By using this condensed representation leaving out one or more parts of the basic elements, as well as the combinations introduced in the following, the MSCs described in the previous section can be obtained.

Combination of Elements. To compose more complex MSCs, we now formalize the combination of two functions with start locations a and c as well as end locations c and b , resp., which may either be basic elements (a, p_1, s_1, p_2, c) and (c, p_3, s_2, p_4, b) consisting of messages or conditions as described before, or combinations themselves.

As mentioned above, an axis of a component can be formatted in a different way to specify the relative timing between the occurrences of messages and conditions in a more flexible manner. Solid lines indicate exactly one time tick, and dashed lines at least one time tick between messages and conditions. Both solid and dashed lines are formalized by disjunction of the corresponding functions and the abstraction of the connected locations. For solid lines, the functions are simply combined disjunctively, unifying control locations c , and then hiding c by means of abstraction. Note that if the postcondition of the preceding sequence p_2 and the precondition of the succeeding behavior p_3 are contradictory, the behavior of the composed function does not contain an observation. For dashed lines, we introduce an additional basic function $(c, true, Nil, true, c)$. The action part *Nil* indicates that no message is exchanged on any channel. Again, all three functions are combined disjunctively and location c is hidden.

Thus, formalization of a single axis leads to a composed function with a single start and end location, identifying the begin and the end of the corresponding MSC axis. Figure 6 illustrates the formalization of all three axis of the MSC connect free and contains an example for each kind of relative timing.

4.2 MSC Behaviors as a Combination of Axes

To formalize an MSC composed of several axes, we combine the functions of all axes to obtain the behavior of the overall MSC. By using the same start control locations

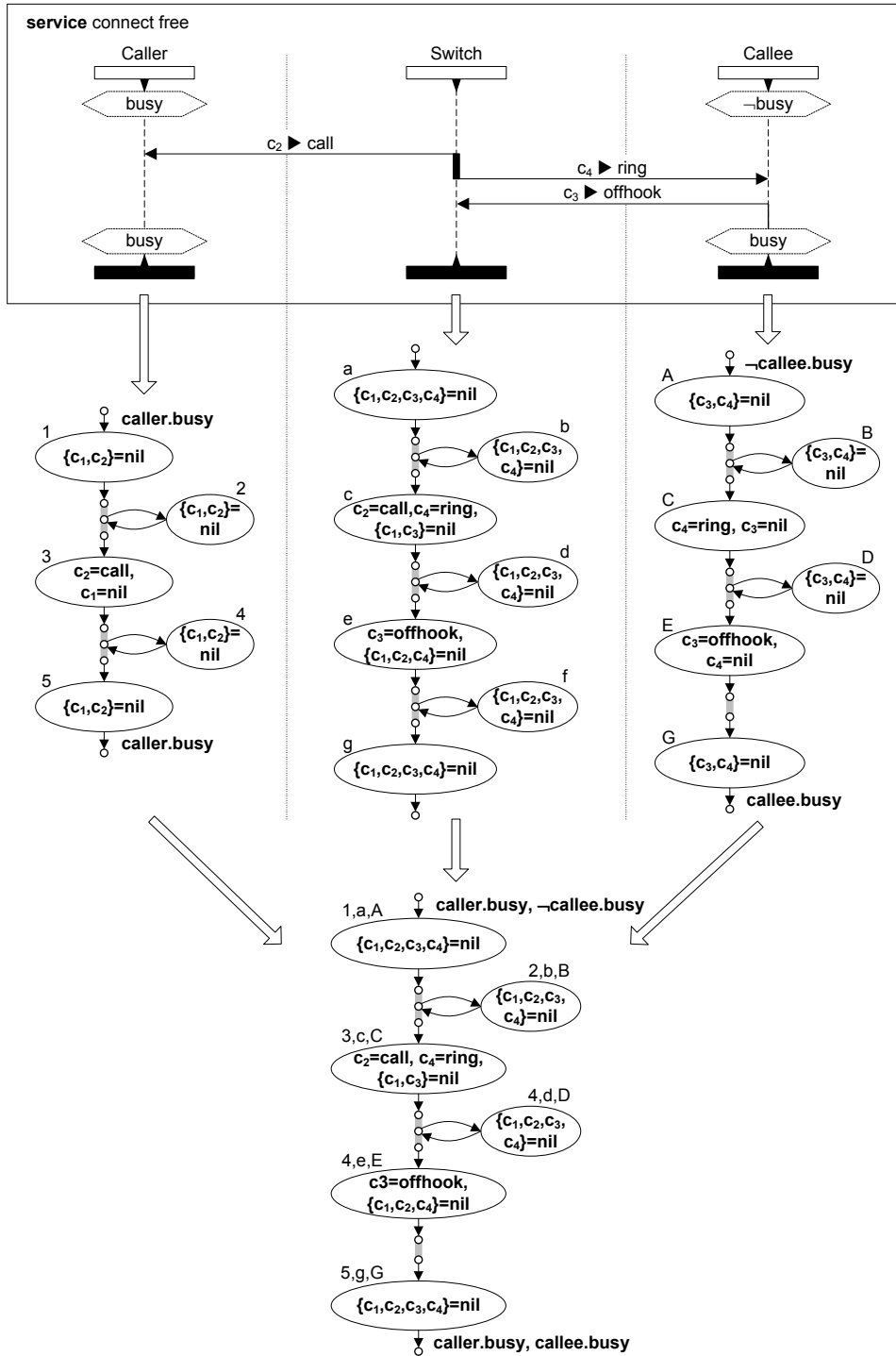


Figure 6. Formalization of the service connect free

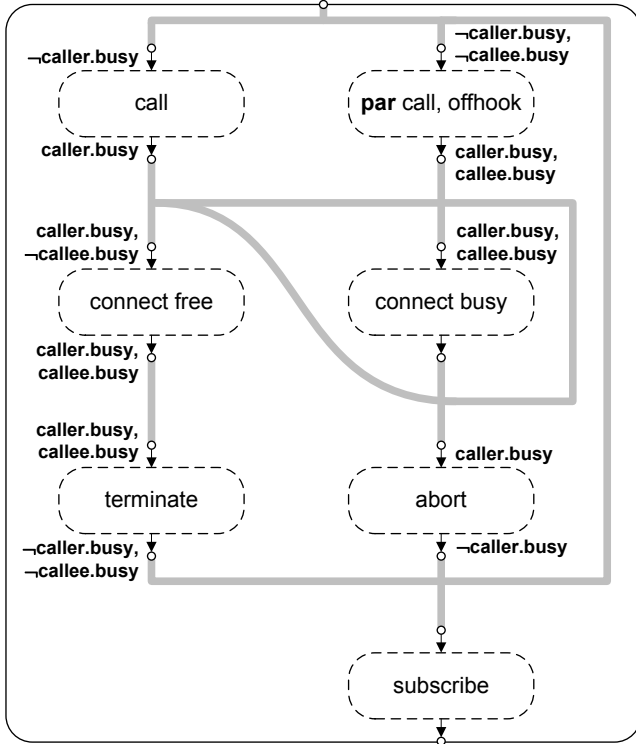


Figure 7. Sequence-oriented combination of services

as well as all end control locations for all axes, the conjunctive combination of the corresponding functions provides the intended behavior, describing the parallel execution of all axes.

As shown in Figure 6, semantically, conjunctive combination is achieved by creating the product of all the axes' functions, and then eliminating the following states and corresponding transitions: product states in which the predicates contradict each other, states which are not reachable from the initial states, and states from which the final states cannot be reached. As a result, the axes synchronize due to common message exchange. Figure 6 illustrates the combination for the service connect free. The states are numbered and referenced in the product automaton.

4.3 Combination of MSCs by HMSCs

In this step, we formalize the combination of MSCs as specified by means of HMSCs. HMSCs – as shown in Figure 2 – allow the sequential and alternative combination of MSCs.

Sequential Combination. Within an HMSC, two MSCs can be sequenced by connecting them by an arrow. Anal-

ogously to the case of the combined functions representing an MSC axis, these two MSCs can be mapped to functions with start locations a and c and end locations c and b , resp. Again, the sequential combination of these MSCs can be formalized by disjunction and abstraction. Thus, the functions are combined by disjunction via shared location c ; by hiding c , the sequential execution of both functions is achieved. Figure 7 shows an example sequence of MSCs connect free and terminate.⁹

Alternative Combination. As shown in Figure 2, within an HMSC, alternative combination of MSCs can be achieved by providing fork and join points between MSCs. Again, disjunctive combination can be used to formalize this form of construction. Here, the MSCs correspond to functions with a joint start location a and a joint end location b . An example for an alternative combination is shown in Figure 7, containing, among other alternative combinations, the combination of MSC sequence connect free as well as terminate and MSC abort.

Hierarchical combination. An HMSC allows to encapsulate a number of MSCs and can be referenced from other MSCs. In order to formalize hierarchical combination, abstraction is required. An example is shown in Figure 7, where the HMSC within the mode AM deactivated is abstracted to a function with one start and end control location.

4.4 Formalization of Extended MSC Constructs

Until now, we have defined a number of basic MSC constructs. The set of constructs can be easily extended by additional syntax constructs by defining a syntactic mapping to the basic ones.

For example, Mode Diagrams allow to partition the system behavior into high-level modes, as is illustrated by Figure 2. However, they can be easily mapped to HMSCs: modes can be transformed to MSC references and the indicator for an initial mode can be transformed to an initial HMSC node.

Another example are parallel boxes. Parallel boxes can be used within MSCs and HMSCs to denote concurrent execution, as is illustrated by Figure 2 and Figure 3. They can be formalized by an alternative combination of all possible orderings of states. Note that some of these messages may also occur within the same time tick.

5 Application

⁹Dotted boxes serve as placeholders for complete MSC functions.

As stated in Section 1, services are intended to support the modular construction of complex functionalities in the development process. The descriptive form of general functional descriptions introduced here – both on the semantical as well as on the syntactical level – eases the combination and reuse of functions and the reasoning about the overall functionality, thus providing a suitable approach to service specification. To apply service specifications in a development process, typical properties of special interest are, e.g., the detection of *partiality* and the establishment of *implementation relations* of service specifications.

To effectively use the implementation relation in a sound development process, (automatic) support for the verification of the implementation relation between two services is necessary. Since the behavior of services is defined by (possibly infinite) sets of finite traces, and the implementation relation is defined by the inclusion relation over those sets, a trace-based formalism is best-suited.

Therefore, here WS1S (weak second order monadic structure with one successor function) is used to implement automatic proof support. This formalism is, e.g., supported by the model checker *Mona* [7]. Using WS1S, functions are specified by predicates over sets of traces. As described in detail in [11], the operators introduced in Section 3 can be directly implemented in WS1S, allowing a compositional construction of the corresponding trace sets. Thus, the implementation relation between two function predicates can be formalized as a subset relation on trace sets; its validity can be verified using *Mona*. Besides proving the refinement relation between two functions, *Mona* can generate a counter-example for functions violating the refinement relation.

6 Summary and Outlook

In this paper we presented fundamental concepts for the structured, service-oriented specification of system behaviors. We introduced basic observations as well as combination and abstraction mechanisms. These fundamental concepts cover a large range of specifications of system functionality. We demonstrated the application of these concepts by formalizing basic syntax constructs of (H)MSCs and Mode Diagrams. Furthermore, we showed how syntactic extensions can be easily formalized by reducing them to other constructs. The formalization of the syntax constructs of (H)MSCs and Mode Diagrams show that the same fundamental concepts occur in different notational techniques (e.g. alternative combination in HMSCs and Mode Diagrams) and on different levels of abstraction (e.g. parallel combination in HMSCs and MSCs). Furthermore, the presented fundamental concepts can be easily transformed into sets of traces which serve as internal representation of the model checker *Mona* for automated verification.

Since services target the early development phases, a methodical development process requires their integration with later-stage description techniques. Here, e.g., [10] provides a similar formalization for state transition systems using different basic observations, thus allowing to relate MSCs to automata-based descriptions. Besides this basic integration, however, methodical issues must be investigated including the supported transition from services to logical components or the deduction of test cases, as well as the scalability of this approach for industrial-size applications.

References

- [1] M. Broy, I. Krüger, and M. Meisinger. A Formal Model of Services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1), 2007.
- [2] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces and Refinement*. Springer New York, 2001. ISBN 0-387-95073-7.
- [3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, pages 293–312. Kluwer Academic Publishers, 1999.
- [4] T. A. Henzinger. Masaccio: A Formal Model for Embedded Components. In *Proceeding of the First International IFIP Conference of Theoretical Computer Science*, pages 549–563. Springer, 2000. LNCS 1872.
- [5] ITU-TS. Recommendation Z.120: Message Sequence Chart (MSC). Geneva, 1996.
- [6] M. Jackson and P. Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, XXIV(10):831–847, October 1998.
- [7] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, 2001.
- [8] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.
- [9] I. H. Krüger, J. Ahluwalia, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, and S. Rittmann. Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS)*, 2004.
- [10] B. Schätz. Modular Functional Descriptions. In *Proceedings 4th International Workshop on Formal Aspects fo Component Software (FACS'07)*, Nice, 2007. Electronic Notes in Theoretical Computer Science.
- [11] B. Schätz and C. Pfaller. Integrating Component Tests to System Tests. In *5th International Workshop on Formal Aspects of Component Software*, 2008. To appear in: Electronic Notes on Theoretical Computer Science.