Towards Precise Service Specification with UML and UML-RT

Ingolf H. Krüger

California Institute for Telecommunication and Information Technology
University of California, San Diego
La Jolla, CA, 92093-0114, USA

Abstract

The notion of service enjoys increasing popularity as a means for structuring complex distributed systems. The current trend towards “web services” is just one example of this increase in popularity; others occur in safety-critical systems (such as in the automotive or avionics domain), where precise specification, and correct implementation of requirements are essential. Surprisingly, however, no precise, mathematical foundation supporting the use of services as a pervasive abstraction across development phases exists to date. Most definitions of the term service refer only to syntactic lists of procedures or methods upon which clients can call. This is inadequate as a basis for more elaborate service specifications that include, for instance, Quality-of-Service properties. Consequently, the modeling notations as provided by UML/UML-RT need to be adapted to support services as first-class modeling elements. Here, we suggest a formal foundation for services; this foundation is based on our observation that services are mainly defined by the interplay of components in the system under consideration.

We discuss the limitations of UML and UML-RT with respect to service specification, and show how Message Sequence Charts (MSCs) – an integral part of the emerging UML 2.0 standard – can be semantically mapped to our service definition. This yields an intuitive, flexible notation for service specifications.

1 Introduction

The notion of service plays a fundamental role in the telecommunications domain. The enormous number of individual functions within telecommunication switches, for instance, is often organized according to the user-triggered services (such as “establish call”, “start call-forwarding”, to name just two examples) or maintenance/business services (such as “startup”, “update”, and “billing”) implemented by these functions.

Increasingly, the term service is adopted in domains other than traditional telecommunication; more and more systems are built by composing pieces of functionality that are available via the Internet or another form of communication network. The popularity of “web services” [29] is one indicator of this trend; the proliferation of software-enabled, networked control units in the automotive and avionics domains is another. In particular, the boundaries between traditional business systems, and technical, embedded systems vanish – and so do the differences in the specification techniques, development methods and tools used. Increasingly, for instance, safety-critical subsystems in cars (such as motor and airbag control) blend with comfort systems (such as seat controls and central locking system) – to enable seat adjustments only up to a certain speed, or to open doors automatically after a crash has been detected. Furthermore, the passengers bring in additional devices, including cell phones and personal digital assistants (PDAs); which opens the door for added-value services delivered by networking within and beyond car boundaries.
As a consequence, a precise understanding of what a service is, and what its defining characteristics are, is a prerequisite for specifying, correctly developing, and integrating systems of the envisioned kind. However, despite its importance, no notion of service exists today that adequately serves as the basis for system development in the entire range from requirements capture to implementation. In addition, none of the prominent modeling notations currently available addresses the service concept as a first-class modeling entity.

In this text, we focus on defining a precise notion of service, and on exploring the use of modeling techniques provided by the Unified Modeling Language (UML) and its cousins, as well as Message Sequence Charts (MSCs) for capturing and developing service specifications in an intuitive notation.

1.1 The Notion of Service

What distinguishes a service from, say, a method call upon an object in some programming language? What specification techniques and development methods do we need to develop services systematically? These questions need to be addressed in order to make progress towards a comprehensive, scalable, service-oriented development methodology.

The literature, especially in the telecommunications domain, provides many informal definitions for the term “service” (cf., among others, [11]). An interesting source for a number of such definitions is [13]; there, we find under the entry “(software) service” the following: “A set of functions provided by a (server) software or system to a client software or system, usually accessible through an application programming interface”. Other definitions, appearing in the context of middleware technologies, such as Jini[31], SOAP[29], .NET[23], or JXTA[15], typically capture only syntactic lists of operations upon which a client can call.

In our view, a service is defined by the interaction among the entities involved in establishing the service.

Understanding services as the key interaction patterns occurring during system execution goes well beyond the mostly syntax-driven specification of function names and datatypes as advocated in the mentioned middleware approaches. If a service is defined by its behavior, and the interactions ensuing from invoking the service, we immediately get a handle at important development aspects, including service composition, refinement, and Quality-of-Service specifications such as throughput, latency or security of information transmission, and – most importantly – correctness. Many of these quality attributes cannot be established locally; instead, they emerge from the interplay of several system components.

1.2 Services and Service-Oriented Software Architectures

The service notion introduced above is applicable across development phases, be it during requirements capture – where interactions occur among conceptual components – or during implementation – where interactions occur among technical or implementation components.

Because of this flexibility, our service notion fits particularly well with the emerging trend towards service-oriented software architectures for complex, distributed, and reactive systems. Here, a system consists of uniquely identifiable services, which are built on top of a layer of supporting infrastructure. Every service registers with a central directory; subsequently the registered service can be located and used by other services. The infrastructure also handles the communication between services. Examples of existing architectures following variations of this scheme are Parlay[11], Jini[31] and web services[29, 5].

Because these approaches address mainly the implementation and deployment of services, there is ample need for systematic development steps leading to the eventual implementation. The key point here is the shift of focus from the component providing a service to the service as a “first-class” implementation element. Correspondingly, we are interested in introducing services as first-class modeling elements.
1.3 Implications for Service-Oriented Modeling

A consequence of the above service definition – and the realities of system implementation on the basis of, say, web services – is that a central step in modeling service-oriented systems is the capturing and modeling of the interaction patterns among the components establishing a service.

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)[25] 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).

Message Sequence Charts (MSCs), 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.

None of these notations, however, supports an independent service concept, let alone one that is precise enough to enable development and validation of safety-critical systems. Up to now, hopes for seamless integration of the service concept into modeling notations and methodologies are in vain.

1.4 Contributions and Outline

In the following sections we address three main themes. First, in Sec. 2, we introduce a precise service notion on the basis of a formal system model. This service notion focuses mainly on the interplay between system components, but can easily be extended to cater for elaborate hierarchical system specifications, and even for representing detailed Quality-of-Service constraints. Second, in Sec. 3, we show how to make the relationship between MSCs and our service notion formally precise. Third, in Sections 4 and 5, we investigate in more detail to what extent the UML and UML-RT support service specifications in the precise sense of Sec. 2, and discuss steps required towards establishing a more thorough integration of services into these modeling notations. In Section 6 we present our conclusions, and discuss opportunities for future work.

As a result, we obtain both a mathematically founded notion of service, and – using MSCs – an intuitive description technique for it, which nicely embeds into the (upcoming version 2.0 of the) UML.

2 A Framework for Precise Service Specification

As we have outlined in Sec. 1, the notion of service still lacks a precise foundation. In this section we provide a first step in this direction, specifically geared towards our understanding of services as patterns of interaction.

2.1 System Model

We prepare our precise definition for services by first introducing the structural and behavioral model (the system model) on which we base our work. We pay special attention to providing a system model that enables interaction- and state-oriented behavior specifications in parallel. This is a prerequisite for a seamless integration of these two complementary architectural aspects; this integration is needed, for instance, to capture Quality-of-Service (QoS) specifications. Along the way we introduce the notation and concepts we need to describe the model.
2.1.1 System Structure

Structurally, a system consists of a set $P$ of components, objects, or processes, 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 $S \overset{\text{def}}{=} \Pi_{p \in P} S_p$. For simplicity, we represent messages by the set $M$ of message identifiers.

Fig. 1 shows a system structure diagram (SSD) describing the sets $P$ and $C$ in graphical notation for a simplified central locking system (CLS) for cars. We assume that the CLS consists of four components: a key sensor ($KS$), a left and a right lock motor ($LM$ and $RM$), and the controller ($Control$). Fig. 1 defines $P = \{LM, Control, KS, RM\}$ and $C = \{cl, lc, cr, rc, ec\}$.

![Figure 1: Simple SSD that defines the sets $P$ and $C$](image)

2.1.2 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. [1, 3]).

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. [4, 30]). Here, we only use and introduce a small fraction of this rich semantic model; for a thorough introduction to the topic, we refer the reader to [30, 4].

A stream is a finite or infinite sequence of messages. By $X^*$ and $X^\omega$ we denote the set of finite and infinite sequences over set $X$, respectively. $X^\omega \overset{\text{def}}{=} X^* \cup X^\infty$ denotes the set of streams over set $X$. Note that we may identify $X^*$ and $X^\infty$ with $\bigcup_{i \in \mathbb{N}} ([0,i] \rightarrow X)$ and $\mathbb{N} \rightarrow X$, respectively. This allows us, for $x \in X^\omega$ and $n \in \mathbb{N}$, to use function application to write $x.n$ for the $n$-th element of stream $x$.

We define $\tilde{C} \overset{\text{def}}{=} C \rightarrow M^*$ 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.

---

1In the remainder of this document, we use the terms components, objects, and processes interchangeably.
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 $S^\infty$.

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 $P((C \times S)^\infty)$. Any element $(\varphi_1, \varphi_2)$ of a system’s semantics consists of a valuation of the system’s channels ($\varphi_1 \in C^\infty$) and a description of the system state over time ($\varphi_2 \in S^\infty$). The existence of more than one element in the semantics of a system indicates nondeterminism.

### 2.2 Service Notion

Based on our observation that the key to understanding a service is to understand the interplay of the components involved in delivering the service, we define our service notion to be a projection of the overall system behavior on a certain period of time. More precisely, we define a set

$$Q \subseteq (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$$

to be a service (specification) with respect to the system model introduced in Sec. 2.1; here, $\mathbb{N}_\infty$ denotes the set $\mathbb{N} \cup \{\infty\}$, i.e. the set of natural numbers together with their supremum $\infty$.

Given a service $Q$, every element $(\varphi, t) \in Q$ describes one nondeterministic alternative of the system’s behavior until time $t$. This service notion captures in an abstract way what happens in the system under consideration until a certain time point; it refers to two major aspects of system behavior: component interaction and state change. Components are referred to only indirectly as the sources and destinations of channels, and as the locations for program state in this model.

As an example, consider the following service specification, where $c, d \in C$, such that $c \neq d$ are arbitrary channels:

$$Q_{rr} = \{ (\varphi, t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty :$$

$$\langle \forall t_1 \in [0, t] : \varphi_1.c.t_1 = \langle \text{request} \rangle : \langle \exists t_2 \in [t_1, t] : \varphi_1.d.t_2 = \langle \text{reply} \rangle \rangle \} \}$$

This specification describes a service where every message request on channel $c$ is followed by a message reply on channel $d$. In Sec. 3 we will show how such services can be specified more intuitively by means of MSCs.

The service notion we have defined above is quite abstract and general. For instance, in the service specification $Q_{rr}$ we do not constrain the behavior on any channel other than $c$ and $d$. In a sense, with a service as defined above we specify only what the system must satisfy at least. During requirements capture this “looseness” caters for incomplete specifications. As more requirements become available, we can easily eliminate undesirable behaviors from a service specification.

As an example, consider the specification of Quality-of-Service (QoS) constraints; we can formulate them as predicates on the interaction- or state-behavior patterns that constitute a service. Let, for instance, $Q$ be a service specification, $c, d \in \mathbb{N}$ be natural numbers, and $c \in C$ a channel. Then for all $(\varphi, t) \in Q$ the predicate$^3$:

$$\langle \forall t_1, t_2 : t_2 - t_1 \leq d : \# \{ t_3 \in [t_1, t_2] : \langle \exists c \in C : \langle m \rangle = \langle \varphi_1, c.t_3 \rangle \} \rangle \rangle \geq e$$

specifies that within at most $d$ time units at least $e$ instances of message $m$ occur in the system. This QoS constraint expresses a global liveness property for the service specification; it does not state, however, which component is responsible for implementing the liveness property. Another constraint easily formulated in this framework is a bound on the number of different states assumed during service execution; this is an example for specifying resource limitations. The following predicate specifies that component $p$ assumes at most $n$ different states while participating in service $Q$ (let $p \in P$, $t_1, n \in \mathbb{N}$):

$$\langle \forall (\varphi, t) \in Q : \# \{ s \in S_p : \varphi_2.t_1.p = s \wedge t_1 < t \} < n \rangle$$

$^2$For any $\varphi \in (\tilde{C} \times S)^\infty$ we define $\varphi_1 \in C^\infty$ to be $\varphi$’s projection onto its first component; similarly we define $\varphi_2 \in S^\infty$ to be $\varphi$’s projection onto its second component.

$^3$For any set $A$ by $\# A \in \mathbb{N}_\infty$ we denote the number of $A$’s elements.
We refer the reader to [18] for a more detailed discussion of the suitability of this service notion for composition, refinement, and hierarchical service specifications. In particular, our service notion enables composition of services displaying overlapping behavior; a corresponding join operator is defined in [16]. The system model can also be adapted to the specification of mobile, as well as real-time and hybrid systems. Using, for instance, $\mathbb{R}$ instead of $\mathbb{N}$ as the basis for our notion of time yields a very flexible model for specifying both real-time requirements and continuous system behavior in general (see, for instance, [2, 10] for the technical details).

Because of its generality, our system model also allows inclusion of resources other than “time” and “memory” into specifications. One way to achieve this is to assign more elaborate types to the channels connecting components (inducing, say, security or performance constraints); another is to model resources explicitly as components in the architecture of the system under consideration. This enables specification of and reasoning about meaningful, global QoS constraints.

3 MSCs as a Precise Description Technique for Services

In the following we establish a semantic mapping from MSCs to the precise service notion introduced in Section 2.2. This also serves as a basis for semantic support for the notion of service in the emerging UML 2.0 standard (cf. Sec. 4); our semantic approach – based on the notion of streams – blends nicely with the history-based action semantics for the UML[6]. In the interest of space we constrain ourselves to a significant subset of the notational elements mentioned in Sec. 4. For a comprehensive treatment of MSC syntax and semantics we refer the reader to [16]; further approaches to defining MSC semantics appear, among others, in [12, 21, 24, 3, 7, 19].

3.1 MSCs

MSCs [14, 24, 16] provide a rich graphical notation for capturing interaction patterns. MSCs have emerged in the context of SDL[8] as a means for specifying communication protocols in telecommunication systems. 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.

![Figure 2: Basic MSC for the “locking” service](image)

As an example, consider the specification of interactions within our imaginary CLS. The MSC of Fig. 2 depicts how these components collaborate to provide the locking service of the CLS.

Component Control receives message $ec\rightarrow lck$ from KS. Upon receipt of this message Control issues messages $cl\rightarrow down$ and $cr\rightarrow down$ to LM and RM, respectively. Each of the components
LM and RM acknowledges Control’s request by sending back a reply message ($lc\triangleright rdy$ and $rc\triangleright rdy$, respectively). The box labeled par (for “parallel”) indicates that the interactions between Control and LM, and Control and RM proceed independently.

Fig. 3 (a) shows the analogous interaction pattern for the unlocking service. In this specification we introduce labeled hexagons to indicate information about a component’s state while it partakes in an interaction pattern. Here, Control transits from state $LCKD$ (for “locked”) to $UNLD$ (for “unlocked”).

Syntactically, we have adopted a slightly modified version of MSC-96\[14\]; 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 Sec. 3.2.

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 (b), for instance, specifies that every system execution is an infinite sequence of steps, where each step consists of the locking or the unlocking of the car.

The MSCs of Figures 2 and 3 (a) represent a “global” view on the collaboration of the four components to establish the desired effect for the respective service. The HMSC $CLS$ of Fig. 3 (b) specifies how the locking and unlocking services compose to yield the “CLS” service. HMSCs are particularly interesting for abstract forms of service specification, because they refer to the components involved in the interaction pattern only indirectly.

### 3.2 Mapping MSCs to Services

Because of their focus on the global collaboration view, MSCs are particularly suited for service specifications; in this section we will establish a semantic link between the precise service notion we work with, and the intuitive notation of MSCs.

#### 3.2.1 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 $\alpha$ and $\beta$ we denote by $\alpha;\beta$ and $\alpha\sim\beta$ the sequencing and interleaving of $\alpha$’s and $\beta$’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 \texttt{false}, and \(\alpha\) otherwise. By \(\alpha \uparrow g\) we denote a “while” loop, repeating the interactions of \(\alpha\) while \(g\) evaluates to \texttt{true}; a special case is \(\alpha \uparrow \infty\), which denotes an infinite repetition of \(\alpha\).

An MSC definition associates a name with an interaction specification, written \texttt{msc X = \alpha}. By \(\langle \text{MSC} \rangle\) and \(\langle \text{MSCNAME} \rangle\) 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 \(\to Y\), where \(Y\) is the name of the MSC to be referenced.

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

\[
\texttt{msc locking} = \texttt{ec} \searrow \texttt{lck}; (\texttt{cl} \searrow \texttt{ldn}; \texttt{lc} \searrow \texttt{lmr}) \sim (\texttt{cr} \searrow \texttt{rdn}; \texttt{rc} \searrow \texttt{rmr})
\]

Similarly, the composite service depicted in Fig. 3 (b) translates into:

\[
\texttt{msc CLS} = (\to \texttt{locking} \mid \to \texttt{unlocking}) \uparrow \infty
\]

\subsection*{3.2.2 Denotational Semantics}

In this section we introduce the semantic mapping from the textual representation of MSCs into the semantic domain \((\bar{C} \times S)^\infty \times \mathbb{N}_\infty\). 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 Sec. 2.1. Put another way, we interpret an MSC as a constraint at the possible behaviors of the system under consideration. More precisely, with every \(\alpha \in \langle \text{MSC} \rangle\) and every \(u \in \mathbb{N}_\infty\) we associate a set \(\llbracket \alpha \rrbracket_u \in \mathcal{P}((\bar{C} \times S)^\infty \times \mathbb{N}_\infty)\); any element of \(\llbracket \alpha \rrbracket_u\) is a pair of the form \((\varphi, t) \in (\bar{C} \times S)^\infty \times \mathbb{N}_\infty\). The first constituent, \(\varphi\), of such a pair describes an infinite system behavior. \(u\) and the pair’s second constituent, \(t\), describe the time interval within which \(\alpha\) 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 \(\alpha\) makes no statement whatsoever about the interactions and state changes happening in the system (cf. Fig. 4).

\textbf{Definition 1 (Behavior “Beyond Infinity”)} To model that we cannot observe (or constrain) system behavior “beyond infinity” we define that for all \(\varphi \in (\bar{C} \times S)^\infty\), \(\alpha \in \langle \text{MSC} \rangle\), and \(t \in \mathbb{N}_\infty\) the following predicate holds:

\[
(\varphi, t) \in \llbracket \alpha \rrbracket_\infty
\]

We assume given a relation \(\text{MSCR} \subseteq (\langle \text{MSCNAME} \rangle \times \langle \text{MSC} \rangle)\), which associates MSC names with their interaction descriptions. We expect \(\text{MSCR}\) to be the result of parsing all of a given MSC document’s MSC definitions. For every MSC definition \texttt{msc X = \alpha} in the MSC document we assume the existence of an entry \((X, \alpha)\) in \(\text{MSCR}\). For simplicity we require the MSC term associated with an MSC name via \(\text{MSCR}\) to be unique.

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

\[
\llbracket \texttt{empty} \rrbracket_u \overset{\text{def}}{=} \{(\varphi, u) : \varphi \in (\bar{C} \times S)^\infty\}
\]

![Figure 4: MSC \(\alpha\) constrains system behavior \(\varphi\) only over time interval \([u, t]\)](image-url)
**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:

$$\text{any}_u \overset{\text{def}}{=} \{(\varphi, t) \in (\hat{C} \times S)^{\infty} \times \mathbb{N} : t \geq u\}$$

any subsumes all possible behavior, i.e. for all $$\alpha \in \langle \text{MSC} \rangle$$ we have:

$$[\alpha]_u \subseteq [\text{any}]_u$$

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:

$$[ch \triangledown m]_u \overset{\text{def}}{=} \{(\varphi, t') \in (\hat{C} \times S)^{\infty} \times \mathbb{N} : \min\{v : v > u \land m \in \pi_1(\varphi).v.ch\}\}$$

Because we disallow pairs $$(\varphi, \infty)$$ in $$[ch \triangledown m]_u$$ 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. The channel names used in message specifications, and the channel names appearing in an SSD (such as the one shown in Fig. 1) must be consistent, i.e. a message can occur only on a channel between two components if such a channel exists in the corresponding SSD.

**Sequential Composition** The semantics of the semicolon operator is sequential composition (strong sequencing in the terms of [14]): given two MSCs $$\alpha$$ and $$\beta$$ the MSC $$\alpha ; \beta$$ denotes that we can separate each system behavior in a prefix and a suffix such that $$\alpha$$ describes the prefix and $$\beta$$ describes the suffix:

$$[\alpha ; \beta]_u \overset{\text{def}}{=} \{(\varphi, t) \in (\hat{C} \times S)^{\infty} \times \mathbb{N} : \exists t' \in \mathbb{N} : (\varphi, t') \in [\alpha]_u \land (\varphi, t) \in [\beta]_{t'}\}$$

**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 $$[p_K] \in P(S)$$ 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 $$\alpha$$:

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

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 $$\alpha$$ 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 $$p \lor q \equiv \text{true}$$ we define:

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

For guards $$p$$ and $$q$$ with $$p \land q \equiv \text{true}$$ the alternative expresses a nondeterministic choice.
References  If an MSC named \( X \) exists in the given MSC document, i.e. there exists a pair \((X, \alpha) \in MSCR \) for some \( \alpha \in (MSC) \), then the semantics of a reference to \( X \) equals the semantics of \( \alpha \). Otherwise, i.e. if no adequate MSC definition exists, we associate the meaning of \texttt{any} with the reference:

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

To identify \texttt{any} with an unbound reference has the advantage that we can understand the binding of references as a form of refinement.

Interleaving  Intuitively, the semantics of interleaving MSCs \( \alpha \) and \( \beta \) “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 [16] for the details.

Loops  Here, we concentrate on guarded loops, where the validity of a guard determines further execution of the loop body; this loop construct suffices for the purposes of this text. 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:

\[
[\alpha \uparrow p]_u = \left[ (p : (\alpha ; \alpha \uparrow p)) \mid (\neg p) : \text{empty} \right]_u
\]

The fixpoint exists because of the monotonicity of its defining equation (with respect to set inclusion); see [16] for the rationale, as well as for other forms of loops (such as bounded finite and unbounded repetitions). On the basis of guarded repetition we can easily define the semantics of \( \alpha \)'s infinite repetition (written \( \alpha \uparrow \infty \)) as follows:

\[
[\alpha \uparrow \infty]_u \overset{\text{def}}{=} [\alpha \uparrow \text{true}]_u
\]

MSCs and Services  In a final step we link service specifications with MSCs; given an MSC \( \alpha \) we immediately obtain a service specification \( Q_\alpha \) as follows:

\[
Q_\alpha \overset{\text{def}}{=} [\alpha]_0
\]

This definition facilitates the use of MSCs as a graphical description technique for service specifications. By virtue of the above construction we also have at our disposal synthesis algorithms such as [17], which translate MSCs into corresponding automaton specifications for individual components. This provides a systematic handle at automatically deriving interface behavior for the components involved in a service.

4 Service Modeling with UML/UML-RT

Having the service notion introduced in Section 2.2, as well as MSCs as a corresponding graphical description technique available, we can now investigate to what extent modeling languages such as the UML[25] and UML-RT[26, 20] support service specifications.

Our service notion is based on the observation that a central element in the design of distributed, reactive systems is the interplay of the components participating in the execution of a certain task. Therefore, we place our emphasis on the expressiveness of the “candidate” notations with respect to interaction patterns.

4.1 UML (version < 2.0)  The UML versions below 2.0 already provide a plethora of description techniques for structural and behavioral system aspects. Class, object, component, and deployment diagrams focus mainly on system structure, whereas statecharts emphasize behavior, typically that of individual components.
Activity, sequence, collaboration, and use case diagrams also focus on behavior, but additionally reference structural elements (such as names of objects or classes).

Sequence and collaboration diagrams are the UML's primary description techniques for component interaction. Sequence diagrams are syntactically similar to MSCs as introduced in Sec. 3.1, but add notation for representing method calls and control flow. Collaboration diagrams resemble SSDs (cf. Fig. 1), slightly modified by labeling the channels with the messages they carry; sequence numbers prefix the messages to represent the order in which the messages occur.

At first sight, these description techniques seem to be good candidates for service specifications. However, despite their syntactic proximity to MSCs, sequence and collaboration diagrams are very limited in expressiveness both syntactically and semantically.

For instance, sequence diagrams are anonymous, which precludes referencing them in other parts of the specification. Moreover, their syntactic means for expressing alternatives and repetition are limited; expressing the independent sending of messages, or composing a specification from parts, which is straightforward in MSCs (cf. Figures 2 and 3), is a severe challenge for all but trivial examples using sequence diagrams. Although activity diagrams could play the role of HMSCs as "roadmaps" through a service specification, the limited referencing mechanisms of the UML hinder seamless integration of these diagram types.

The UML supports the concept of a role, i.e. an axis in a sequence diagram can represent a class rather than a concrete object; this is useful for representing abstract service specifications which can be instantiated to concrete objects. However, the binding of roles to concrete objects happens by subclassing, which results in cluttered class specifications if objects play multiple roles within the system.

Furthermore, the UML provides no conceptual notion of components (component diagrams and the corresponding interface notion specifically refer to implementation components), let alone a notion of hierarchy (especially for class diagrams). This leads to entangled interaction patterns unless a very disciplined approach to component-oriented development is pursued.

In summary, the major application for sequence and collaboration diagrams is the informal representation of short scenarios. The integration of services into a modeling approach based on the UML requires significant methodological support, as well as modifications to the notations themselves.

4.2 UML-RT

UML-RT is a derivative of ROOM[27] (Real-Time Object-Oriented Modeling) and the UML. UML-RT provides graphical description techniques for capturing hierarchical structural decomposition (via capsule and class diagrams), asynchronous point-to-point (p2p) component interactions (via sequence diagrams), and individual component behavior (via a variant of the UML’s statecharts).

In the following we concentrate briefly on capsule and sequence diagrams; these are the major models we work with for service specifications.

A capsule in UML-RT represents a potentially active component whose communication with its environment proceeds by means of asynchronous signal exchange via its ports. A port is an interface object defining the role of the capsule it belongs to within a communication protocol. Connectors establish p2p communication links between different ports, and define the protocol carried out on this link. A protocol in UML-RT consists of a set of signals sent and received along a connector; surprisingly, however, the ordering of these signals is not part of the protocol specification in UML-RT; UML-RT suggests the use of sequence diagrams for modeling protocols and protocol roles. As we have discussed in the context of the UML’s description techniques above, however, the expressiveness and applicability of sequence diagrams is very limited for service specifications.

Capsules can nest hierarchically to arbitrary depth; an enclosing capsule communicates with its sub-capsules also via ports and connectors just as it does with its environment. There is no means for accessing sub-capsules directly from the environment of their container.

The strictly hierarchical capsule concept, together with the notions of ports, connectors, and protocols improve upon the UML’s support for service-oriented approaches to system development.
However, there still exists a discontinuity in the step from capturing the interaction patterns defining a service, and the specification of capsules and p2p protocols implementing these interaction patterns.

4.3 UML 2.0

The upcoming version 2.0 of the UML[22] promises to combine the strengths of UML-RT’s hierarchical component model with an adapted form of MSCs. The interaction descriptions in the proposal include major changes to the former model for sequence and collaboration diagrams. Each diagram now is a named entity, and can be referenced from within others. Sequence diagrams now provide operators known from MSCs (including alternatives, parallel composition, and loops). In addition, the proposal contains operators for “critical regions” (interaction patterns which act as “atomic entities” to their context, thus preventing interleaving of the pattern’s events with those from the context), “negation” (to indicate interaction patterns which are forbidden), “assertions” (constraints, say, at the state space of the model), and several forms of sequencing (“strict” sequencing amounts to traditional sequential composition, whereas “weak” sequencing demands sequential composition only on individual lifelines; events occurring on different lifelines may be unordered unless an ordering is established by explicit communication).

However, the notion of service does not exist as an independent, abstract modeling element in UML 2.0. Some work in that direction does occur in the context of [28], where the notion of “scenario” plays a central role for QoS specifications. Our service notion of Sec. 2.2 can serve as the basis for a semantics definition for integrating the interaction concept of UML 2.0 and the scenario notion in [28].

Moreover, a prerequisite for meaningful QoS constraints is the ability to model resource access over time; we have given two examples in Sec. 2.2. The UML 2.0 proposal for interactions could be enhanced by means of “tags” that could be inserted to keep track, say, of message instances or occurrences of states in an execution; this would enable application of OCL specifications to represent QoS properties as we did in Sec. 2.2.

The integration and adaptation of MSCs into the UML goes a long way towards having a precise model for service specifications. However, there is also enormous potential for improvement in the 2.0 proposal; we address important steps towards this goal in the next section.

5 Seamless Integration of Services into the UML – Beyond the Status Quo

Despite the critical assessment in the preceding sections we conclude that the support for modeling interactions and services within UML 2.0 significantly surpasses its predecessors. In particular, because the revised sequence diagrams are directly derived from MSCs, the semantics mapping we have provided in Sec. 3.2 is a semantics basis for a significant subset of these diagrams. This immediately provides us with an intuitive and expressive notation for specifying services; it can easily be augmented to support, for instance, the QoS specifications we have discussed in 2.2. Because QoS constraints are typically of non-local nature, services are an ideal context for expressing such constraints. The UML’s sequence diagrams already have notation for specifying simple time constraints; this can be combined with our service semantics, and used as the basis for more elaborate QoS adornments in the sense of [28].

However, to fully unfold the potentials of an interaction-based service notion, we have to consider methodological aspects as well as notational and semantic ones.

A central challenge in the development process for service-oriented systems is to bridge the gap between requirements capture and implementation. During requirements capture, the components hosting a certain service are typically unknown – instead, we may have some knowledge about conceptual entities involved in establishing a service. During implementation we have to map the conceptually specified service onto an existing infrastructure.
The following aspects are essential to making progress towards a seamless integration of services into a UML-supported development process:

**Component Model for Services:** As we have noted in Sec. 4, the UML fails to provide services as a modeling concept. This forces the developer to think in terms of complete component behavior too early in the development process.

An alternative is to borrow from the component model as provided by UML-RT and UML 2.0, and to turn services into abstract components of their own right. This yields a “black-box” view of global system aspects. Intuitively, each of the services “locking” and “unlocking” of the CLS example (cf. Figures 2 and 3 (a)) then becomes a component of an abstract, service-oriented view of the system under consideration.

A key question then is what interfaces/ports these service components will expose. The example of the “CLS” service (cf. Fig. 3 (b)) shows that an interface for control flow might be required (establishing the order between the locking and unlocking services). Depending on how the system is cut into services, however, there might also be the need for dataflow between the services. Note that the services locking and unlocking are closed in the sense that no communication occurs with outside components. Imagine, in contrast, an open service specification in which the component “KS” is not referenced in the service specifications for locking and unlocking. Then each of the services would have to provide a port such that the “lck” and “unlck” messages could be relayed to component Control via this port.

An even more elaborate notion of service-component would not only encapsulate one interaction pattern; it would rather enable hierarchical decomposition to enable composite services. Then the introduction of a registry (to enable dynamic addition and removal of services) would be the logical next step. This would yield a faithful and scalable abstraction of today's service-oriented system implementations.

**Advanced Composition Operators:** A consequence of introducing services as components as described in the preceding paragraphs is that we have to devise new, powerful composition operators for services.

Consider again the CLS example; each of the services locking and unlocking refers to all of the four components KS, Control, LM, and RM. Thus, the service specifications for locking and unlocking are overlapping.

Composition operators for overlapping structure and behavior are almost entirely lacking even in UML 2.0. Activity diagrams come closest to what MSCs offer under the label HMSCs; neither of the two allows to specify coinciding messages in different interaction patterns. Because there are many ways of cutting an overall collaboration into services, the ability to include a single component (and its interaction patterns) into multiple service specifications is a prerequisite to obtaining comprehensible models.

**Layered Architectures:** To maintain the traceability between the service specification on the conceptual level and the implementation, we envision a seamless development process based on our interaction-centered service notion, yielding multiple layers of software architectures for services in an incremental fashion. The logical architecture consists of components representing the individual services, without associating concrete (physical or binary) components where the functionality is located. Establishing this association is a design step, and leads to the actual implementation architecture. In it, the concrete components performing the necessary steps for delivering the service are known; the actual “code” for establishing a particular service may be spread over several implementation components. In this incremental process there will be multiple layers of architectures involved in the design of a complex system, such that layer $i + 1$ will represent the implementation architecture for the logical architecture captured in layer $i$. In the development of each architectural layer the capturing or refinement of services is accompanied by building a domain model capturing the relevant system entities in that layer. This defines the “vocabulary” available for representing the services and their interactions.
**Refinement and Refactoring Techniques:** Key methodological tools for building a sequence of layered architectures, each representing a more concrete version of the service-oriented system under consideration, are structural and behavioral refinement, as well as “refactoring” techniques[9].

By virtue of the refinement notions we have introduced in [16] for MSCs, we immediately obtain corresponding support for systematic development for services. Structural refinement, for instance, is a systematic means for organizing services into a layered architecture without breaking the layering in subsequent development steps; UML 2.0 offers this under the label “part decomposition”. Similar refinement notions – not supported by the UML – allow the mapping from interaction patterns among conceptual components to their technical counterparts (interaction refinement), and the removal of nondeterminism from service specifications (property refinement).

In fact, property refinement amounts to simple set inclusion in our system model: if \( Q \subseteq T \) we call \( Q \) a property refinement of \( T \).

An example of an important refactoring step for service components as introduced above is to “externalize” a component referenced within a service specification. An example would be to turn the component \( KS \) referenced in the locking and unlocking services into an external component. As discussed above this induces introduction of a port playing the role of \( KS \) in the resulting service descriptions.

Systematic refinement techniques, combined with more pragmatic, implementation-oriented approaches such as refactoring for concrete applications, are particularly important to maintain the notion of service as a pervasive abstraction across development phases.

**6 Conclusions and Outlook**

One of the keys to systematic development of complex, reactive systems is to have a thorough understanding of the services the system provides. Here, we have introduced a precise, mathematical notion of service, based on the observation that a service is defined by the interplay among components required to establish a certain result. We have shown that our notion of service supports, in particular, Quality-of-Service specifications, such as resource constraints; such constraints are typically non-local in nature, and can only be formulated taking the interplay among several components into account.

Comprehensibility of their design is especially important in the domain of safety- and security-critical systems. Still, however, most of these systems are developed such that the final hardware architecture determines the structuring of software components right from the start of software design. As a consequence it is often difficult, if not impossible, to reason about the collaboration of the resulting designs and implementations. By focusing on the interplay of components to establish certain services we provide a level of modeling abstraction reducing the complexity of models for system behavior: each service involves “it’s” components only partially; therefore, less behavioral aspects have to be taken into account in reasoning about a service.

We have shown how to map the intuitive graphical notation of MSCs to our service notion. As a result we can use MSCs as a precise graphical description technique for the interaction patterns defining services. Because MSCs will find their way into the UML 2.0 standard in the form of extended sequence diagrams, this also enables using them as a graphical description technique for services in the context of the UML.

We have also alluded to the challenges arising in the seamless embedding of services into an overall development process for service-oriented systems within the UML. We have compared the various description techniques for component interaction offered by various versions of the UML with respect to their applicability for service specifications. It turns out that the UML, even in the forthcoming version 2.0, lacks an independent service concept as well as expressive composition operators for services. We have presented a “wish-list”, which indicates the direction in which the UML should evolve to fully embrace services as a pervasive modeling concept.

We conjecture that unless the UML adopts services as a central modeling element its corresponding tools will not mature towards true support for systematic software construction, especially in the context of critical systems design.
Adressing the items referred to in the “wish-list” provides ample opportunity for future work. Of particular importance are the availability of a hierarchical notion of service-component, with corresponding composition operators and systematic development steps, to yield a truly scalable software engineering approach based on services.

References


