Specifying Services with UML and UML-RT
Foundations, Challenges and Limitations

Ingolf H. Krüger

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

Abstract

The key to systematic development of complex, reactive systems is to have a thorough understanding of the services the system provides. A service, in our view, is defined by the interplay among components required to establish a certain result. Services shift attention from the details of individual components to a global view of the system. We give a formal definition for the notion of service, and discuss to what extent the modeling languages provided by UML and UML-RT support a service-oriented development approach.

1 Introduction

More and more software systems are formed by composing individual, even personalized software services running on various types of computing platforms in the range from large servers to electronic control units (ECUs) in cars to smart-card systems. The popularity of today’s client/server-based Internet applications, set-top boxes, cellular phones, portable digital assistants, and even handheld computer games is only a first indicator of the potentials that highly distributed, interacting, service-oriented software systems have to offer.

But what exactly is a service, what distinguishes it from, say, a method call upon an object in some programming language, and what do we need to develop services systematically? Are prominent modeling languages such as the UML ready for the specification of services?

1.1 The notion of service

The literature provides many informal definitions for the term “service”, inspired mainly by applications in the telecommunications domain (cf., among others, [7]). In [8], for instance, 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”. Similar definitions appear in the context of middleware technologies, such as Jini[25], SOAP[23], .NET[18], or JXTA[12]; they do recognize services as a central element in system implementation. However, their notion of service typically consists only of (syntactic) lists of operations upon which a client can call. The order in which certain calls are to be performed – the protocol for accessing the service – is not considered at all.

We believe, however, that a key element in describing services precisely is the interaction among the entities involved in establishing the service.

To support this observation we give an example from the automotive domain. In some of today’s luxury cars the user has access to no less than 700 different functions, ranging from fine tuning of the airconditioning system to probing tire pressure. The individual functions (such as “move seat forward”, “move seat backward”) compose to more elaborate services (such as “recall driver’s seat position”). Increasingly, services brought into the car by the passengers along with their cellular phones, laptop computers, or personal digital assistants (PDAs), interact with systems onboard the vehicle; consider, as an example, cell phone-based navigation systems interacting with the onboard display of the luxury car mentioned earlier.

This example shows that in developing the mentioned services we have to have a profound understanding of the interactions within the system under consideration. This goes well beyond the mostly syntax-driven specification of function names and datatypes as advocated in the mentioned middleware approaches. The example also shows another important trend: more and more, safety relevant and convenience features converge – yet, we don’t want the software for the airconditioning, or a multiplayer cell-phone game to interfere maliciously with the motor management or airbag control in a car. This calls for a precise, mathematically founded service notion enabling systematic analysis and design of component interaction.

1.2 Systematic Service Development

The complexity of designing and managing the interplay between multiple individual services is significant. While technologies such as XML, SOAP, and .NET certainly provide an implementation platform for certain services, a more fundamental, conceptual approach for capturing the requirements of and developing the specification for services is needed. Proving the correctness, and predicting the resource demands are examples of difficult problems already for all but the most trivial “stand-alone” systems; interaction among and coordination of services aggravates the situation further.

Typically, there exists a significant gap between the capturing of requirements, and the following phases of design and implementation for software systems. The different services a system provides to its environment are usually spread out over a number of components within the implementation of the
system. Traditionally, client/server systems were built around a central server (platform), with a clear distinction between clients and servers; increasingly, this separation of concerns vanishes, yielding systems where every component plays both the client and the server role over time.

A crucial question then is how to trace requirements captured and modeled early in the development process down to their counterparts in the implementation. The lack of this traceability, in turn, makes it difficult to ensure important quality attributes such as throughput, latency or security of information transmission (often referred to as “Quality-of-Service” attributes), and – most importantly – correctness. Many of these quality attributes cannot be established locally; instead, they emerge from the interplay of several system components.

This makes the case for studying and designing these properties based on the collaboration of the components participating in establishing a service; as a result we can reason on a global, instead of only on a local basis. The mapping of the global interaction protocols and properties to the behavior of individual components then becomes a step of system design, leading from more abstract descriptions of services to more concrete specifications of components implementing the service.

Most prominent software development approaches and modeling languages, however, place their focus on the construction of individual software components, instead of on component collaboration. An example from the area of object-oriented development is the Unified Modeling Language (UML)[19]. Its syntactic means 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). Corresponding tool environments, for instance, provide almost no support for the transition from an interaction specification to component specifications implementing these interactions.

Consequently, development processes based on the UML (cf., for instance, [13,11]) place an emphasis on the development of statecharts – the coordination aspect is typically viewed only as part of the informal initial requirements capturing, or as a means of documentation. This is particularly true, also, for methodologies emphasizing use cases (cf., for instance, [10]); use cases describe interactions between a user and the system under development in an informal manner. They are mainly employed during very early development stages to capture the core functionality of the system as a whole. A seamless transition from use cases to system implementation is neither intended, nor automatically feasible.

Therefore, such development approaches and their corresponding tools need to be reconsidered in view of the service notion and iterative development approach we assume in this text.
1.3 Contributions and Outline

In the following sections we address two challenges. In Section 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. Based on this service notion we investigate in Section 3 to what degree industrially accepted modeling languages (namely UML and its “real-time” companion UML-RT) support the specification of services; we also point at potentials for their improvement. In Section 4 we present our conclusions, and discuss opportunities for future work.

2 Towards a Precise Service Notion

As we have outlined in Section 1, the notion of service still lacks a precise foundation. In this section we provide a first step towards such a foundation, 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\(^1\), and a set \( C \) of named channels. Each channel \( ch \in C \) is directed from its source to its destination component; we assume that channel names are unique. Channels connect components that communicate with one another; they also connect components with the environment. Communication proceeds by message exchange over these channels.

With every \( p \in P \) we associate a unique set of states, i.e. a component state space, \( S_p \). We define the state space of the system as \( S \equiv \Pi_{p \in P} S_p \). For simplicity, we represent messages by the set \( M \) of message identifiers. Table 1 summarizes these structural elements.

---

\(^1\) In the remainder of this document, we use the terms components, objects, and processes interchangeably.
Table 1

<table>
<thead>
<tr>
<th>Entity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>set of system components</td>
</tr>
<tr>
<td>(C)</td>
<td>set of directed channels</td>
</tr>
<tr>
<td>(S_p)</td>
<td>state of component (p \in P)</td>
</tr>
<tr>
<td>(S)</td>
<td>system state ((S \overset{\text{def}}{=} \prod_{p \in P} S_p))</td>
</tr>
<tr>
<td>(M)</td>
<td>set of message identifiers</td>
</tr>
</tbody>
</table>

Structural elements of the system model

Figure 1 shows a system structure diagram (SSD), describing the sets \(P\) and \(C\) in graphical notation; it defines \(P = \{LM, Control, RM\}\) and \(C = \{cl, lc, cr, rc, ec\}\).

![Fig. 1. Simple SSD that defines the sets \(P\) and \(C\)](image)

The systems we consider here are fixed in the sense that neither set \(P\), nor set \(C\) changes over time. In Section 2.3 we will discuss how this model can accommodate even systems whose structure dynamically changes.

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. [4,24]). It serves particularly well for property-oriented component specifications, as well as for the definition of refinement notions and for the verification of corresponding refinement relationships between specifications (cf. [14]). 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 [24,4].

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. $X^\omega \overset{\text{def}}{=} X^* \cup X^\omega$ denotes the set of streams over set $X$. Note that we may identify $X^*$ and $X^\omega$ 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 $\tilde{C}^\omega$ as an infinite valuation of all channels. This models that at each point in time a component can send multiple messages on a single channel.

With timed streams over message sequences we have a model for the communication among components over time. Similarly we can define a succession of system states over time as an element of set $S^\omega$.

With these preliminaries in place, we can now define the semantics of a system with channel set $C$, state space $S$, and message set $M$ as an element of $\mathcal{P}((\tilde{C} \times S)^\omega)$. Any element $(\varphi_1, \varphi_2)$ of a system’s semantics consists of a valuation of the system’s channels ($\varphi_1 \in \tilde{C}^\omega$) and a description of the system state over time ($\varphi_2 \in S^\omega$). The existence of more than one element in the semantics of a system indicates nondeterminism.

Table 2 lists the semantic entities for modeling system behavior.

### Table 2

Behavioral elements of the system model

<table>
<thead>
<tr>
<th>Entity</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{C}$</td>
<td>channel valuation ($\tilde{C} \overset{\text{def}}{=} C \rightarrow M^*$)</td>
</tr>
<tr>
<td>$\tilde{C}^\omega$</td>
<td>overall channel history</td>
</tr>
<tr>
<td>$S^\omega$</td>
<td>state history</td>
</tr>
<tr>
<td>$(\tilde{C} \times S)^\omega$</td>
<td>combined channel and state history</td>
</tr>
<tr>
<td>$\mathcal{P}((\tilde{C} \times S)^\omega)$</td>
<td>semantics domain for system behaviors</td>
</tr>
</tbody>
</table>

#### 2.2 Service Notion

Based on the system model introduced above we now define our notion of service formally. This definition serves two purposes. First, it gives us a handle at being precise of what we expect of a service both syntactically and
semantically. Second, it serves as the basis for determining how well UML and UML-RT are suited for describing services – relative to our definition.

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 Section 2.1.

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 \( c \in C \) is an arbitrary channel\(^2\):

\[ Q_t = \{ (\varphi, \infty) : \varphi \in (\tilde{C} \times S)^\infty \wedge \#\{ t \in \mathbb{N} : \langle \text{tick} \rangle = \varphi_1.c.t \} = \infty \} \]

This specification describes a service where message \( \text{tick} \) occurs infinitely often on channel \( c \); this could, for instance, model a time service on channel \( c \). In Section 2.4 we will show how such services can be specified in an intuitive, graphical notation.

2.3 Discussion

The service notion we have defined above is quite abstract and general. For instance, in the service specification \( Q_t \), we do not constrain the behavior on any channel other than \( c \). In a sense, with a service as defined above we specify only what the system must satisfy at least. Because of its “looseness” this service notion readily supports central aspects of practical specifications, which we briefly address in the following:

- **Service composition**: As we have outlined in Section 1 a key element in service-oriented system development is the composition of more elaborate services from existing ones. In the semantic framework we have introduced here we can easily express sequential and parallel composition, finite and infinite repetition, as well as interruption or preemption of services. The model also accommodating the joining of overlapping services, i.e. services

\[^2\text{For any set } A \text{ by } \#A \in \mathbb{N}_\infty \text{ we denote the number of } A\text{'s elements. For any } \varphi \in (\tilde{C} \times S)^\infty \text{ we define } \varphi_1 \in \tilde{C}^\infty \text{ to be } \varphi\text{'s projection onto its first component; similarly we define } \varphi_2 \in S^\infty \text{ to be } \varphi\text{'s projection onto its second component}\]
sharing part of an execution sequence. The composition operators defined in [14] for interaction patterns can be carried over directly to the service notion we have introduced here.

- **Service refinement**: Notions of refinement, such as behavioral and structural refinement, are crucial for a seamless, scalable integration of services into an overall development process. We define, for instance $Q_1$ to be a (behavioral) refinement of $Q_2$, if $Q_1 \subseteq Q_2$ holds. Behavioral refinement reduces the amount of nondeterminism in a service specification. We refer the reader to [14] for details about this and other notions of refinement, including tools for adjusting the granularity of messages and for hierarchical system decomposition.

- **Quality-of-Service specifications**: QoS constraints can be formulated as predicates on the interaction- or state-behavior patterns that constitute a service. Let, for instance, $Q$ be a service specification, $e, d \in \mathbb{N}$ be natural numbers, and $c \in C$ a channel. Then for all $(\varphi, t) \in Q$ the predicate:

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

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 example specifies that component $p$ assumes at most $n$ different states while participating in service $Q$ (let $p \in P, t_1, n \in \mathbb{N}$):

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

- **Real-Time services and hybrid systems**: The notion of time introduced in Section 2.1 serves as a model for causality; its usefulness for specifying real-time constraints is limited – especially in connection with refinement. However, using $\mathbb{R}$ 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, [2,6] for the technical details).

- **Hierarchy**: Hierarchical systems can be modeled by mapping the component structure onto the set of component names; this way, the name of a component reflects its position in the component hierarchy. By imposing the component structure on the set $P$ of component names we can easily enforce a strict notion of hierarchy in our system model. To enforce a simple form of “layered architecture” [5], for instance, we simply need to formulate a predicate describing that components on different layers of the hierarchy may not share channels.
• **Interfaces and projections:** We can add further structure to our service specifications by stating precisely which channels (or components) are involved, and which of the messages exchanged are relevant in establishing the service; this leads to a notion of *service interface*. Mathematically, this corresponds to projections of the overall system model onto a sub-model defined by the relevant messages, components, and channels.

• **Mobility:** Our system model is based on the static structure formed by the sets $C$, $S$, and $P$. We can characterize mobile systems as those whose structure changes over time; this includes the addition or deletion of components and channels, and also the relocation of a component from one position in the component hierarchy to another. By introducing a state automaton, whose states represent the current values for the mentioned sets, and whose transitions describe the modifications to these sets, we immediately get a handle at dealing with mobility; the service notion will then be defined relative to the automaton describing system evolution.

Thus, our formal framework covers a broad spectrum of important modeling requirements for services; in particular, it directly reflects our goal of placing interaction in the center of concern of service specification. In the next section we will see how we can specify the interaction patterns on which our service notion is based in an intuitive, graphical notation.

### 2.4 Graphical Modeling of Services with MSCs

Message Sequence Charts (MSCs) [9,14] are a graphical notation for capturing component collaboration. Typically, an 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 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 to bottom) within an MSC defines possible sequences of interactions among the depicted components.

As an example, consider the specification of a central locking system (CLS). The CLS we consider here consists of four components: a key sensor ($KS$), a left and a right lock motor ($LM$ and $RM$), and the controller ($Control$). Figure 1 shows the SSD for this system. The controller receives message $ecc>lck$ or $ecc>unlock$ from the key sensor when the operator locks or unlocks the car, respectively. Upon receipt of either message the controller initiates the locking and unlocking by issuing appropriate messages ($clr>down/ clr>down$ or $clr>up/ clr>up$) to both motors. Each of the motors acknowledges the controller’s request by sending a reply message ($le>rdy$ and $re>rdy$) to the controller.

We capture these informally described services by means of MSCs as shown

---

<sup>3</sup> For any $c \in C$ and $m \in M$ we denote by $c> m$ the sending of message $m$ on channel $c$.  

---
Fig. 2. Specification of the “locking” and “unlocking” services in Figures 2 (a) and (b). The boxes labeled “par” indicate that the interactions depicted above the dashed line can occur in parallel with the interactions below that line. Syntactically, we have adopted a slightly modified version of MSC-96[9]; our message arrows carry an indicator for the channel on which a message is sent in addition to the message itself, as suggested by our system model.

MSC-96 provides a rich set of operators for composing MSCs; particularly appealing in the context of service specification are High-Level MSCs (HMSCs). Intuitively, 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 Figure 3, 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.

Fig. 3. HMSC for the CLS

The MSCs of Figure 2 represent a “global” view on the collaboration of the four components to establish the desired effect for the respective service. The HMSC CLS of Figure 3 specifies how the locking and unlocking services compose to yield the “CLS” service.

In [14] we have introduced a semantic mapping that associates with every MSC a subset of $(\hat{C} \times S)^\infty \times N_\infty$. As a consequence we can use MSCs directly
to describe services as defined in Section 2.2.

3 Service Specification in UML and UML-RT

Having the service notion introduced in Section 2.2, as well as MSCs as a “candidate” description technique (cf. Section 2.4) available, we can now investigate to what extent modeling languages such as the UML[19] and UML-RT[20,17] support service specifications. We discuss the notations of UML and UML-RT that most closely relate to our service specifications, together with their limitations, in Section 3.1. In Section 3.2 we mention steps towards a more seamless integration of the notion of service into the framework of UML and UML-RT.

3.1 Modeling Notations in UML/UML-RT and their Limitations

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 notations provided by UML and UML-RT with respect to interaction patterns.

3.1.1 UML

The UML provides 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 Section 2.4, but add notation for representing method calls and control flow. Collaboration diagrams resemble SSDs (cf. Figure 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.

### 3.1.2 UML-RT

UML-RT is a derivative of ROOM[21] (Real-Time Object-Oriented Modeling) and the UML (Unified Modeling Language). 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, 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 di-
rectly 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.

3.2 Towards Methodological Service Development

An obvious cure for some of the mentioned limitations of UML and UML-RT is the adoption of a more powerful sequence diagram notation; the notation we have described briefly in Section 2.4 already goes beyond sequence diagrams in the composition operators it offers, its referencing mechanism, and its semantic foundation. It can also be extended to cover control flow specifications as needed for interaction patterns based on procedure calls.

To mitigate the discontinuity from captured interaction patterns for services and their “implementation” via ports and protocols we have developed techniques for inferring state-based component behavior from MSCs (cf. [15,14,16]); these techniques can also be used to generate state-automata for each individual port of a UML-RT model. This bridges the gap between services and their p2p realization in UML-RT. Scalability of this approach is ensured by the strict hierarchical structuring of capsule specifications; this leads to service specifications of manageable size.

More ambitiously, we envision a seamless development process based on an 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.

We illustrate the separation between logical and technical architecture as sketched in Figure 4 again by means of the CLS example. Logically, the system might be structured into infrastructure services and a power locking management service. For an implementation there might be fixed, physical
components like a door controller, the door locks, as well as a particular communication bus, available. The services specified logically (represented by the large shaded oval in Figure 4) will have to be distributed over the components of the implementation architecture (small shaded ovals in Figure 4).

UML-RT provides helpful concepts for representing components, as well as layered architectures as needed in the envisioned approach. The systematic, traceable handling of the design step shown in Figure 4, however, is a topic of ongoing research.

In [22] the modeling of scheduling constraints, performance requirements and similar QoS specifications is achieved by annotating an extended notion of scenario; this yields a more seamless integration of services into the modeling notations of the UML. By combining the resulting “type system” for service properties with a systematic approach to architecture transformations we expect a first step towards a thorough methodology for service-oriented system design to emerge.

4 Conclusions and Outlook

The notion of service receives an increasing amount of attention in both academia and industry. In this text we have suggested 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 have demonstrated that our service notion supports simple QoS specifications, and have discussed its viability in important methodological aspects such as service composition and refinement. MSCs provide an intuitive graphical representation for our service notion. Based on these preliminaries we have discussed the description techniques provided by UML and UML-RT and their relationship to our service notion.

Within the UML, the expressiveness and integration of interaction specifi-
cations with other description techniques displays much potential for improvement. UML-RT improves upon the UML by introducing a clear, hierarchic component model with an emphasis on the protocols for exchanging signals between components. However, because currently UML-RT favors point-to-point communication protocols, a gap between the specification of services and their implementation in the UML-RT framework remains.

As a step towards a more seamless integration of services into UML-RT we have suggested to increase the expressiveness of the UML’s sequence diagrams by means of notational elements known from MSCs, to employ automatic synthesis techniques yielding state models for ports directly from MSC specifications, and to adopt an incremental and iterative, service-centered development process, yielding increasingly concrete layers of software architectures.

More research, however, is needed to provide a thorough methodological basis for service-oriented software and systems engineering. In particular, the extent to which the formal notion of services introduced here is scalable and adequate for capturing relevant aspects of services needs to be investigated further. Moreover, the mapping between services specified on adjacent layers of architectural abstraction needs careful consideration. Furthermore, the overall practicality of the service-centered development process we envision needs to be validated by means of case studies.

Acknowledgments

The author is grateful to Manfred Broy, Bernd Finkbeiner, Bill Griswold, Bill Howden, Leonid Kof, Ramesh Rao, Bernhard Schätz, and Alexander Wisspeintner for interesting discussions and valuable input on the topics presented here.

References


