Applying Service-Oriented Development to Complex System: a BART case study

Ingolf H. Krüger[1], Michael Meisinger[2], and Massimiliano Menarini[1]

[1] Computer Science and Engineering Department
University of California, San Diego
La Jolla, CA 92093-0404, USA,
{ikrueger,mmenarin}@cs.ucsd.edu,
[2] Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany,
meisinge@in.tum.de

Abstract. Complex distributed systems with control parts are difficult to develop and maintain. One reason of the complexity is the high degree of interaction and parallelism in these systems. Systematic, architecture-centric approaches are required to model, implement and verify such systems. To manage complexity, we apply a service-oriented development process, yielding manageable and flexible architecture specifications. We specify interaction patterns defining services using an extended Message Sequence Chart notation. We model a portion of the BART system as a case study, demonstrating the applicability of our methodology to this domain of complex, distributed, reactive systems. Our approach allows us to separate the problem of orchestrating the interactions between distributed components and developing the control algorithms for the various control tasks. We provide a brief overview of service-oriented development and service-oriented architectures, as well as a detailed description of our results for the BART case study.

1 Introduction

Distributed, reactive systems are notoriously difficult to develop – especially when they are interaction- *and* control-intensive. The Bay Area Rapid Transit (BART) system with its Advanced Automatic Train Control (AATC) as controlling software is a telling example; another such area, which is increasingly recognized across academia and industry as a challenging application area for advanced software technologies is the automotive domain with its mix of safety-critical and comfort functions. The shift from monolithic to highly networked, heterogeneous, interactive systems, occurring across application domains, has led to a dramatic increase in both development and system complexity. At the same time the demands for safety, reliability, and other quality attributes have increased.

The major challenge in developing such systems is to manage the complexity induced by the distribution and interaction of the corresponding components. Model-based development techniques and notations have emerged as an approach to dealing with this complexity, in particular during the analysis, specification and design phases of the development process; popular examples are UML, SysML, ROOM and SDL. Each of these examples proposes managing the complexity of software development by separating the two major modeling concerns: system structure and system behavior.

In application domains such as process control, automotive, avionics, telecommunications and networking, the logical and physical component distribution has introduced the additional challenge of modeling, analysis and handling of cross-cutting concerns such as security and Quality-of-Service. Because system functions are scattered across modeling and implementation entities, the cross-cutting concerns in the system are increasingly difficult to trace and to ensure during all steps of the development process.

Service-Oriented Development (SOD) and Architectures (SOA) have been suggested as an approach to system development and architecture that helps address both system complexity *and* cross-cutting concerns, including the mentioned quality properties. Because services typically emerge from the interplay of multiple system components, SOD places particular focus on the interaction between components and system-wide functions.

1.1 Service-Oriented Architectures and Development

The center of concern in model-based design has so far mostly been individual components rather than their interplay. In contrast, service-oriented design emphasizes the interaction among components by using the notion of service to decouple abstract behavior from implementation architectures supporting it. The term "service" is used in multiple different meanings and on multiple different levels of abstraction throughout the Software and Systems Engineering community. Web Services currently receive a lot of attention from both academia and industry. Figure 1 shows a typical "layout" of applications composed as a set of (web) services. Often such systems consist of at least two distinct layers: a domain layer and a service layer. The domain layer houses all domain objects and their associated logic. The service layer acts as a façade to the underlying domain objects - in effect, offering an interface that shields the domain objects from client software. Typically, services in this sense coordinate workflows among the domain objects; they may also call, and thus depend on, other services. Some of the services, say Service 1 and Service 2 in our example, may reside on the same physical machine, whereas others, such as Service n may be accessible remotely via the Internet.

The layout shown in Figure 1 is prototypical not only of the typical situation we find for applications structured in terms of web services, but also for other domains where complex, often distributed applications are expected to offer externally accessible interfaces. Abstracting from the domain-specific details we observe that services often encapsulate the coordination of sets of domain objects

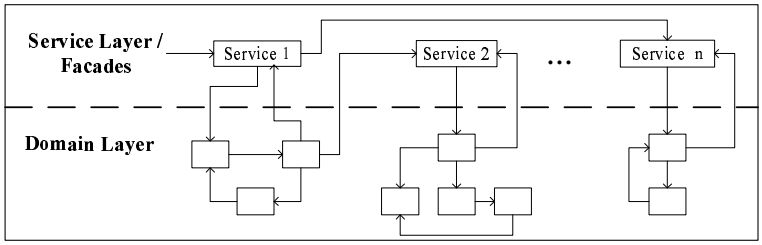


Fig. 1. Service-Oriented Architectures

to implement "use cases". We focus on the coordination aspect of each use case and define services as *partial* interaction specifications.

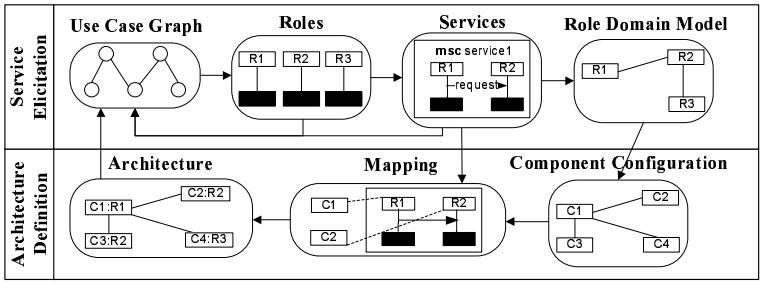


Fig. 2. Service-Oriented Development Process

Our approach to service-oriented development rests on the observation that services orchestrate a set of entities, each of which makes a *partial* contribution to the execution of the service. Whereas in traditional, component-oriented development approaches, component interplay is often treated as an afterthought, we place the orchestration aspect of services in the center of the development process from the outset. We have developed a two-phase, iterative development process as shown in Figure 2[15, 13]. In the following, we first give a brief overview of this process as we have applied it, among others, to the development of service-oriented automotive software; then we describe the extensions we introduce in this paper to deal with complex, control-intensive systems.

Phase (1), Service Elicitation, consists of defining the set of services of interest - we call this set the service repository. Phase (2), Architecture Definition, consists of mapping the services to component configurations to define deployments of the architecture.

In phase (1) we identify the relevant use cases and their relationships in the form of a use case graph. This gives us a relatively high-level, scenario-based view on the system. From the use cases we derive sets of roles and services as interaction patterns among roles. Roles describe the contribution of an entity to

a particular service independently of what concrete implementation component will deliver this contribution. An object or component of the implementation may play multiple roles at the same time. The relationships between the roles, including aggregations and multiplicities, develop into a role domain model. Together with a data domain model, indicating the types of data being manipulated by the system under consideration, the role domain model and the service specification are the foundation for the abstract core of the service-oriented architecture.

In phase (2) the role domain model is refined into a component configuration, onto which the set of services is mapped to yield an architectural configuration. These architectural configurations can be readily implemented and evaluated as target architectures for the system under consideration.

This process is iterative both within the two phases, and across: Role and service elicitation feeds back into the definition of the use case graph; architectures can be refined and refactored to yield new architectural configurations, which may lead to further refinement of the use cases.

The process of transferring house-ownership between two parties (also known as the *escrow* process) is a good example to better illustrate the utility of roles, services and components. Typically, the escrow process involves a number of players, including the seller, the buyer, a mortgage company, multiple real-estate agents, notary-publics, house inspectors, insurance agents and an escrow and title company. The process itself is precisely defined; the various actions of negotiating the price, signing an offer document, provisioning the money, providing proof of insurance, etc. are partially ordered, culminating in the transfer of title if all actions are performed within the required time and ordering – the process can be described properly without mentioning of any *concrete* players, such as a specific buyer, seller or bank. Instead, we can define the *escrow service* as the proper interplay among the set of players (which we call roles). An instance of the service emerges by mapping the roles to concrete players (which we call components). The service captures the deployment-independent aspects of the system under consideration; a concrete deployment (mapping of roles to components) defines an architecture configuration.

Following the process presented above allows us to specify system-wide services separately and map them subsequently to a given deployment architecture. Integration-complexity is addressed early in the development process by focusing on component interactions as the defining element of services. In the following section we show how we can also address control complexity in our apporach.

1.2 Contribution

To be successful in applying SOD and SOA to complex distributed systems with control challenges, software engineers need a thorough understanding of how to identify services and a corresponding architecture systematically, how to specify the services and architecture, how to implement, validate and verify the resulting specifications, and how to address the control requirements. In this paper, we present our approach to SOD/SOA based on a clear understanding of services as partial interaction patterns, combined with a systematic, flexible, iterative

development process for services and service-oriented architectures. Using the BART case study, we explain the benefits of our notations and process, as well as the tool-chain we have developed.

The main contribution of this paper is to show the applicability and efficacy of service-oriented development for complex distributed systems with control parts. We show how service-oriented development helps to develop effective architectures for complex distributed systems and how control algorithms can be independently developed and integrated into this approach. This helps us to manage the integration complexity that is caused by the high degree of distribution and thus parallelism of the system. Figure 3 depicts the development process and how we deal with control. The control problem is reduced to a set of local actions, the algorithms are developed and implemented independently from the service-oriented architecture and are called from the various roles in their local actions.

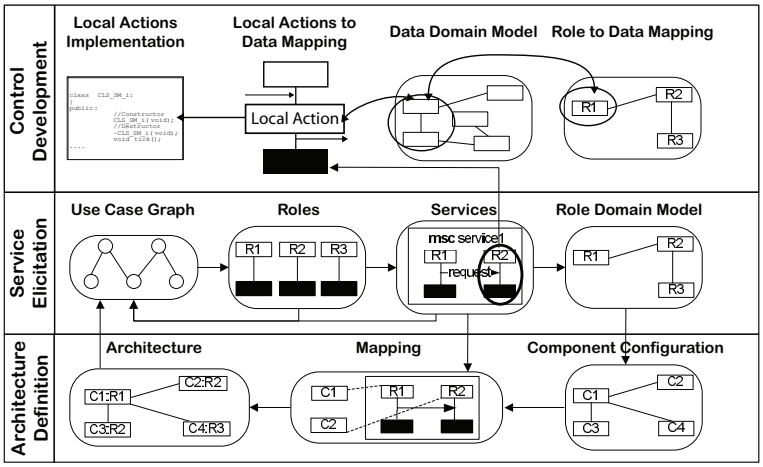


Fig. 3. Service-Oriented Development Process with Control

2 Complex Systems with Reactive and Control Parts

The systems we are addressing with our service-oriented approach are complex distributed systems. The complexity we refer to here stems from the need to integrate multiple different parts whose interplay is difficult to grasp with traditional techniques. Rather than treating component interplay as an afterthought, addressed only during late stages of deployment and integration, we focus on *services*, defined as the interaction patterns among roles, throughout the development process.

Complex reactive systems are often used in control applications. In this field, often the control is applied to actuators and sensors that interact with the physical world. Some of the complexity arises then from the fact that part of the domain (the physical world to be controlled) is best modeled using continuous data types and behavior, wheres the remainder of the domain can best be characterized using discrete data types and behavior. This system class is known as *hybrid* systems [2]. Particularly challenging are complex hybrid systems where the complexity of the distributed communication is increased by real time requirements of control algorithms. For instance, control algorithms can impose tight constraints on the latency and jitter of the communication infrastructure. Furthermore, if an algorithm has to deal with continuous measures the task of sampling and discretizing the control can transform a simple set of differential equations into a storm of messages that needs to be exchanged between components.

Distributed control systems, if developed in an ad-hoc fashion, result in tight coupling between modules and complex, inflexible data exchange to establish and maintain global state. To alleviate these problems, various software infrastructures and middlewares [24, 7] have been developed. The complexity of developing new control application from scratch time and again has led to the introduction of reusable standard platforms [17]. For instance, in industrial control the use of Function Blocks (IEC 61499) allows isolation of the control algorithms from the distributed interaction.

Because of what Leveson defines as the "curse of complexity" [16] it is, however, difficult and error-prone to separate the control blocks from the distributed communication infrastructure. The real challenge is to keep a system-level view while breaking down the problem into subproblems of a manageable size. To this end, our service-oriented approach permits breaking down the system into services capturing the interaction patterns among roles. Role states and their transitions capture the partial state-based behavior of any component that participates in the execution of this service.

Our way of integrating the hybrid aspects into a system specification is to associate the control parts with local activities of the roles. These activities are invoked as the corresponding service is executed.

In the remainder of this paper we focus on the analysis of the interactions between system entities. The control algorithms can be modeled and developed using well-established techniques and be called via local activities upon the reception of some message by a role.

2.1 The BART Case Study

The BART case study [25] describes parts of the Advanced Automatic Train Control (AATC) system of the Bay Area Rapid Transit (BART) system. BART is the San Francisco area, heavy commuter rail train system. The case study describes the part of the train system that controls spreed and acceleration of the trains. Certain other parts such as communication error recovery and train routing have been left out for the purposes of the case study. The part of the

AATC system described here is suitable as a case study, because it provides a relevant level of detail and shows the complexity and interdependencies of the entire system, yet still remains of manageable size. BART was previously used as a case study in the area of distributed systems and for the application of formal methods [9].

The BART system automatically controls over 50 trains on a large track network with several different lines. Manual operation of the train control is limited mostly to safety issues and to cases of emergency or malfunction. Tracks are unidirectional. Certain sections of the track network are shared by trains of different lines. The system needs to operate switches and gates to ensure correct traffic flow. Tracks are separated into track segments, which may be protected by gates. Gates operate similar to traffic lights and establish the right-of-way where tracks join or merge at switches.

The AATC system controls the train movement – switch and gate handling will transparently be provided by another system. One important AATC requirement is to optimize train speeds and the spacing between the trains to increase the throughput on the congested parts of the network, while constantly ensuring train safety. The AATC system operates computers at the train stations which each control a local part of the track network. A station is responsible for controlling all trains in its area. Stations communicate with the trains via a radio network and with neighboring stations using land-based network links. Each train has two AATC controllers on board with one being the master. Trains receive acceleration and brake commands from the station computers via the radio communication network and feed back information about train speed and other engine status values. The radio network has the capability to track the trains' positions.

The case study concentrates on the parts of the AATC system that controls the trains' acceleration and braking. Controlling the trains must occur efficiently with a high throughput of trains, while ensuring certain safety regulations and conditions. The specification strictly defines certain safety conditions that must never be violated, such as a train must never enter a segment closed by a gate, or, the distance between trains must always exceed the safe stopping distance of the following train under any circumstances.

The system operates in 1/2 second cycles. In each cycle the station control computers receive train information, compute commands for all trains under their control and forward these commands to the trains. All information and commands are time-stamped. Commands to trains become invalid after 2 seconds. If a train does not receive a valid command within 2 seconds, it goes into emergency braking. The control algorithm needs to take this delay, track information and train status into account to compute new commands that never violate the safety conditions. To ensure this, each station computer is attached to an independent safety control computer (VSC) that validates all computed commands for conformance with the safety conditions.

Computing the trains' commands is a complex control problem. Inputs to the corresponding algorithm include the train position estimates, train speeds

and accelerations, static track data (track grades, maximum speeds), switch and gate information from the interlocking system, information from the neighboring stations, interceptions from the safety control computer. The control algorithm needs to balance and optimize train throughput, adherence to the schedule, passenger comfort (not too strong braking and acceleration changes), engine wear and most importantly safety. In normal operations, the station computer computes the train commands in fixed time cycles. However, in case of a detected emergency condition, the system needs to react immediately and take appropriate measures to ensure maximum safety of passengers and equipment.

We focus on modeling the reactive behavior of a station computer and the trains, the safety control computer and the interlocking system as well as certain other external interfaces, as described, in detail, below. We apply our service-oriented development approach to distinguish the different services of the system and to specify a service model than will help us to design a service-oriented architecture. This architecture needs to be effective in supporting the requirements that are listed in the case study. We show how we can abstract from the actual control problems and integrate the necessary computation results and trigger conditions into our reactive model.

Our approach enables rapid architecture design for the AATC; this results in a high level design model that can systematically be refined into an implementable system. We can ensure the correctness of the reactive behavior and integrate the required control parts that trigger the reactive behavior.

3 Applying Service-Oriented Development to the BART Case Study

We have applied our previously introduced service-oriented development approach to the cross-cutting interaction aspects of the BART case study [25]. We have followed the process described above to elicit use cases and an initial role domain model and subsequently have identified and specified the basic services of the system. It is interesting to notice that in the BART case study the set of requirements include very specific information about the prescribed deployment of the system. We used the requirements, which are part of the architecture definition, as part of the input to our service elicitation phase as suggested by our iterative development process. This allowed us to refine our model for a suitable target architecture and to generate prototypic executable code to test the system under development.

In the following, we will explain the steps of the process we have followed in more detail.

3.1 Use Case Elicitation

From the requirements that are present in form of the BART case study document, we came up with a list of use cases:

1. A *train* determines its current status from different sensors in a *consist* (group of cars in a train).
2. A *train* communicates its current status (position, speed, acceleration value) to the responsible *control station*
3. A *control station* receives status messages from all trains in the controlled area in regular time intervals
4. A *control station* receives external input for the controlled area from the *interlocking system* (gate&switch) control and manual speed limit settings
5. A *control station* computes speed and acceleration commands for each *train* in the controlled area
6. A *control station* forwards all commands of an interval cycle to the *VSC* for a reliable safety check
7. The *VSC* relays all safe commands via the comlink to the *trains* in the area
8. A *train* receives a command from its responsible *control station*, and checks the command validity (timestamp). It applies the command to all actuators in the consist.

Each use case, of course, can be broken down into more detailed steps, leading to a comprehensive use case view of the BART system. Analyzing these use cases leads to a first list of basic actors, or *roles*, which we depict in form of a structure diagram. From the use cases, we identify Train, Control Station, the Safety Computer (VSC) and an External Data Source as actors. This leads us to an initial role domain model where we depict the connections between the different actors. Fig. 4 shows the initial role model.

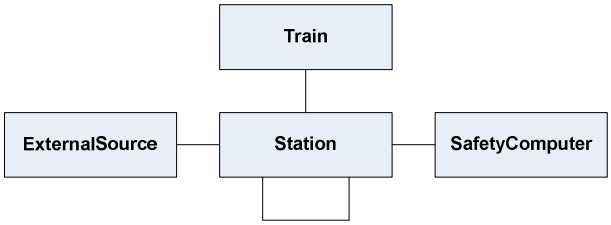


Fig. 4. High Level BART Role Domain Model

3.2 Modeling Services and Roles

We model roles and services together. We start with the initial role domain model of Fig. 4. We systematically go though the list of use cases and identify interaction patterns defining services. In a sense, the services we identify are a refinement of the elicited use cases. In the process of identifying interaction patterns, we may identify further actors; we add these as roles to the role domain model. Finally, after modeling all services the resulting role domain model looks as depicted in Fig. 5.

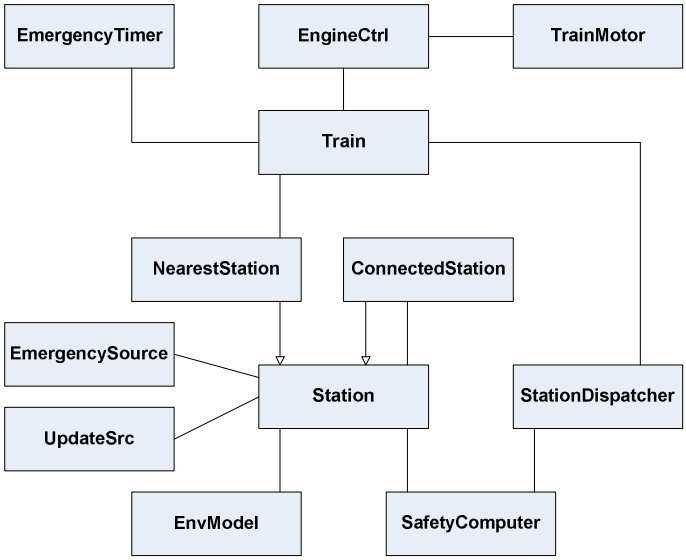


Fig. 5. BART Role Domain Model

For specifying the services, we use the extended MSC notation of [10, 15]. This notation is based on the Message Sequence Chart [8] standard and provides an intuitive graphical language for specifying interaction patterns and is well-accepted among engineers. Extensions to the standard notation were cautiously made based on a formal semantics to provide increased expressiveness and more powerful operators suitable for modeling service-oriented systems. To model the services, we can make use of our tool-chain introduced in [12].

In our service model, we capture the interactions between the station computer (and its subcomponents) with a train (and its subcomponents). Other entities, such as external data sources, are part of the interactions as well. In modeling the interactions, we concentrate on specific use cases and abstract from any concrete deployment architectures. In particular, we do not yet take any multiplicities of the entities into account. We specify the interactions be-

tween a train and the station computer, for instance, no matter of how often this specific interaction happens subsequently or in parallel.

Good design principles suggest a hierarchic design of the service model. The requirements imply a continuous, cyclic operation of a station computer unless an emergency happens. The High Level MSC (HMSC) in Fig. 6 specifies this concept. Intuitively, an HMSC is a graph depicting a *roadmap*, or *flow*, through a set of services. The HMSC in Fig. 6 shows an infinite flow of activities of normal train operation, preempted by exceptional behavior in case of an emergency situation, which needs to be solved after which the situation returns to normal operations. This MSC shows how we model infinite flows of behavior, hierarchic MSCs and preemptive behavior; we introduce each of these aspects now in more detail. Our notation allows us to specify preemptive behavior based on the occurrance of a preemptive message (indicated on the dashed arrow) in an interaction. In this case the interaction at the tail end of the dashed arrow is preempted and continues with the interaction referred to at the tip of the dashed arrow; this can be seen as a preemption handler.

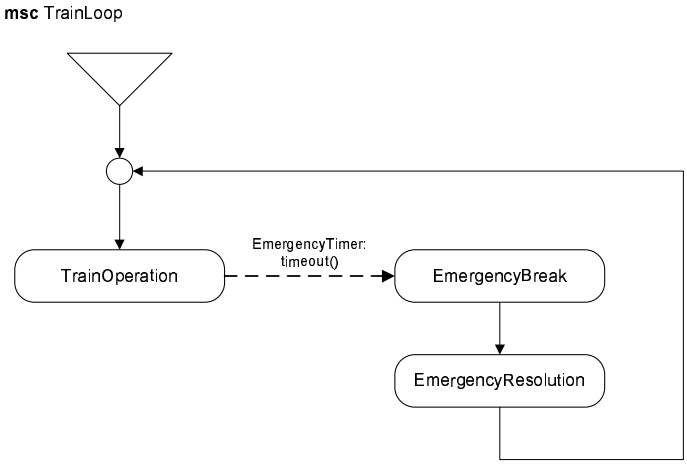


Fig. 6. TrainLoop HMSC for the BART specification

The MSC in Fig. 6 does not yet specify any detailed interaction behavior. MSC references, depicted by the labeled rounded boxes, indicate that more detailed specifications of parts of the behavior are to be found in further MSCs. The functionality for "TrainOperation", referenced in Fig. 6, for instance, is specified in the MSC shown in Fig. 7. This HMSC shows a composition of four services by means of the "join"-Operator, depicted as $\otimes$. The semantics attached to the join of two services is the interleaving of the two behavior specifications, synchronized on common messages.

The *join* operator is a powerful means to combine and synchronize *overlapping* services – this ability to disentangle service specifications is central in

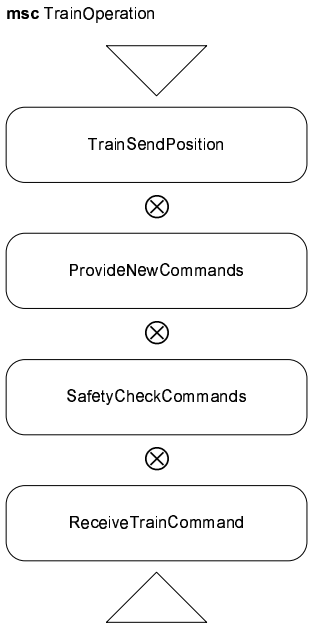


Fig. 7. TrainOperation HMSC for the BART specification

our approach. We call services overlapping if they share at least two roles and at least one message between shared roles. *join* synchronizes its operands on shared messages, while imposing no ordering on all others; in other words, a join is the parallel composition of its operands, with the restriction that the operands synchronize on shared messages. Interactions that are shared in both services will occur only once in the resulting service. This means that all interactions causally before a shared interaction within *both* services must have occurred before the shared interaction can itself happen. The *join* operator does not change the order of interactions in any of the operands. It only restricts the occurrence of shared messages. For a formal definition of the join semantics, see [10, 14].

In general, the operators available in HMSCs are as follows:

- **Sequence**, by connecting two MSC references with an arrow. This operator expresses that the behaviorat the tail end of the arrow precedes the behavior at the tip of the arrow.
- **Non-deterministic choice** is indicated by means of multiple paths leading out of a reference (or a small circle, used for graphical convenience). At execution time the path to follow is chosen nondeterministically.
- **Join***, represented by $\otimes$, which joins two or more services as described above.
- **Parallel**, which represents the interleaving of its operands.
- **Preemption***; the preemptive message (or set of messages) is indicated as a label to the dashed arrow. The service at the tail of the arrow is preempted

if and when the preemptive message occurs; in that event, the execution of the service at the tip of the arrow commences.

Operators marked by * are extensions of the MSC standard. All operators have a precisely defined semantics, which is given in detail in [10]. HMSCs can be transformed into Basic MSCs by applying the algorithm given in [10]. This algorithm transforms an HMSC into a finite state automaton. Subsequently, using the well-known algorithm for translating finite state machines into regular expressions, this automaton is transformed into an expression using only basic interactions (message exchanges) and operators for Basic MSCs. Therefore, in our methodology we do not distinguish between HMSCs and Basic MSCs.

Fig. 8 shows the specification of the functionality of a train sending current status values to the nearest station, processing this information. This specification uses the syntax of an extended Basic MSC, which shows an interaction among roles. Messages are depicted as horizontal arrows between two roles (represented as vertical axes labeled with the name of the role). Messages can have parameters to indicate transmission of data values. The roles visible in this diagram are a subset of the roles of the entire system. Furthermore, some of the roles are specializations of previously introduced roles. The role *NearestStation*, for instance, is a specialization of the *Station* role, which represents the external interface of a station computer for interactions. *NearestStation* represents the station computer of the station, in whose area of responsibility the train currently is. How this distinction is implemented, is irrelevant at this level of abstraction. The *EnvModel* role in this MSC represents an entity responsible for managing all data related to conditions in the environment of a station. Fig. 5 shows the dependencies of roles in the role domain model.

We make use of MSC operators, depicted as labeled boxes, to express repetition and choice in the interaction flow. The $LOOP< * >$ box around all the interactions in the MSC expresses repetitive behavior. In our case we are interested in specifying an infinite loop of interactions for activities required to submit a train's position to the responsible station. The *ALT* boxes indicate alternative or optional behavior. Different alternatives are separated by horizontal dashed lines through the box. To indicate optional behavior, we leave one of the alternatives empty. By means of state markers at the top of each compartment we indicate the conditions determining which alternative is chosen. In Fig. 8, a station only processes a train's information if the train is in the station's area of responsibility.

In general, we use the following operators in Basic MSCs:

- **ALT** to express choice, guarded by conditions. If conditions are omitted, the choice between the alternatives is non-deterministic.
- **LOOP** to express repetitive behavior. Loops can be limited to a certain number of repetitions, can be infinite or or can be guarded by a loop condition. If the loop condition is true, the interaction behavior in the box will occur.
- **PAR** to express interleaving. The interactions in both compartments occur independently of one another.

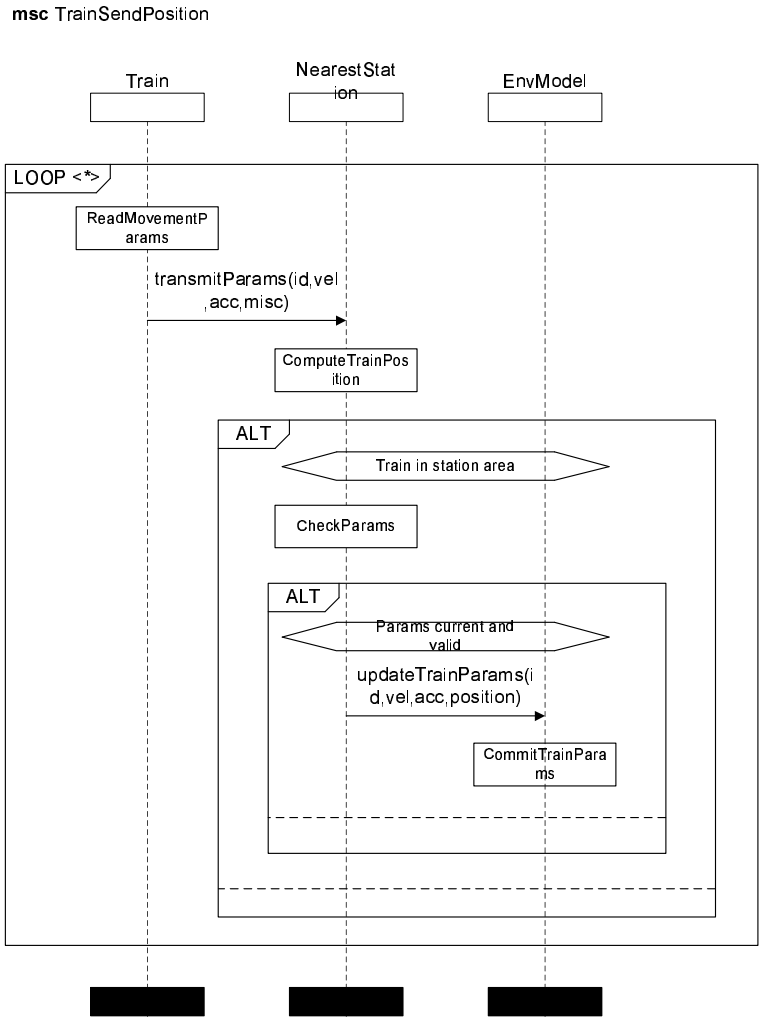
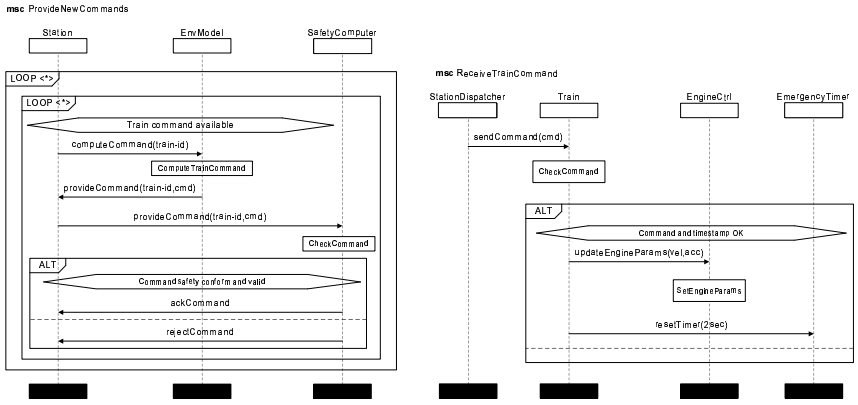


Fig. 8. MSC TrainSendPosition

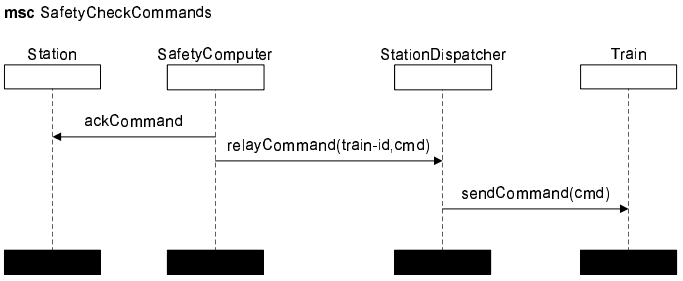
- **JOIN*** to express interleaved composition synchronized on common messages. Common messages are equally named messages between the same two roles.
- **PREEMPT*** to express preemptive behavior. The behavior in the upper compartment is preempted if the specified preemptive message occurs. In this case, the behavior resumes as specification in the lower compartment.
- **TRIGGER*** to express liveness conditions. Whenever the behavior in the upper compartment occurs, it is followed, eventually, by the behavior specified by the lower compartment.

Similar to HMSC operators, all Basic MSC operators marked by * are extensions of the MSC standard. All operators have a precisley defined semantics explained in [10].

We integrate control aspects into reactive interaction specifications by means of *local actions*. Local actions are depicted as labeled boxes on role axes. The meaning of this syntax is that a role performs an activity based on the information available until this point in time. Information can be local variables, data previously received via messages and the role state. The local activity may have a duration, but does not include any communication with other roles while it executes. Activities may change local variables and the role state, which can be used in further interactions or to determine alternative branches of behavior. For instance, the local action *CheckParams* is executed by the role *NearestStation*. The local action can be engineered and implemented independently, given its interface (such as the variables it accesses and controls) is well defined.



(a) ProvideNewCommands (b) ReceiveTrainCommand



(c) SafetyCheckCommand

Fig. 9. Some of the BART specification MSCs

ProvideNewCommands (Fig. 9(a)), *SafetyCheckCommands* (Fig. 9(c)) and *ReceiveTrainCommand* (Fig. 9(b)) show other examples of services specified

in the BART case study. The *ProvideNewCommands* MSC, for instance, contains the *ComputeTrainCommand* local action. It implements a complex control algorithm that is based on the position and state of all trains, knowledge of the physical constraints the train is subject to and other requirements, such as the maximization of travelers' comfort. The *TrainSendPosition* MSC guarantees that the data required will be delivered to the *EnvModel* role, and the *CommitTrainsParams* action persists the data to the role-local state, to be available to the *ComputeTrainCommand* action.

3.3 Mapping the Service Model to Components

The first step in transitioning from a service model with roles and interactions to an implementable architecture is to define the *component types* of the architecture. Component types are blueprints of *component instances* in the architecture. We have to define component types, their communication interfaces to other component types and the services they implement.

Component Type	**Role**	**Description**
FastCPU	Station EnvModel StationDispatcher	A fast CPU computer for operative station control
SlowCPU	SafetyComputer	A slow CPU computer with high reliability (MTBF) used for checking safety conditions
Train	Train TrainMotor EmergenyTimer EngineCtrl	Train computing unit on board of a train
InterlockingSystem	UpdateSrc TrainMotor	The interlocking system, which controls switches and gates

Table 1. BART Role Mapping

It is required that the component model is a refinement of the structural role model introduced above. Fig. 1 shows an example role-to-component mapping for the BART case study.

The behavior of the component types can be derived from the service specifications with the following procedure (described in detail in [18]):

1. For each role that a component implements, project the interaction behavior into a state machine. This state machine will be enabled for all incoming messages that the role will receive as part of the interaction, and it will produce all messages that the role is sending. Furthermore, it will implement the local actions and control the role's local variable and control state.

2. Compose all particular role state machines for that component type into a single product state machine. Perform minimization and optimization steps to produce a result with a manageable number of states.

Repeating these two steps for all component types results in state machine specifications for each of the component types. Each component type combines the behaviors of all the services it is involved in according to the role-to-component mapping. These state machine specifications fulfill the reactive behavior specified by the services and perform the required local actions. A concrete algorithm efficiently implementing these steps in the component synthesis algorithm is described in [10, 11].

3.4 Defining a Component Architecture

Fig. 10 shows a simple sample architecture, which can implement the service model that we have specified above.

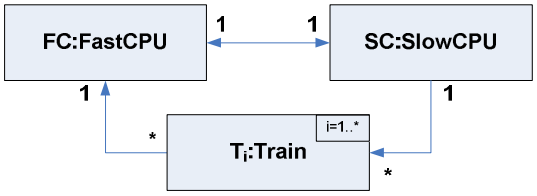


Fig. 10. BART Components Architecture

The component architecture shows the structure of the system's components and their connections. Components are instances of a certain component type and can be present multiple times in a system configuration. Each instance has a defined name and a specific type.

3.5 Designing an Efficient Component Architecture Using Services

Our process distinguishes between roles and components, and provides methodological steps to map a set of roles to a component type. An interesting question is how to design the underlying architecture. The process is iterative in nature. This means that the system modeler can start with a high level, simple view of the system through all steps of the development process and later iteratively refine the respective models.

It is important to structure the service model and similarly the component architecture so that they can be extended and modified efficiently, and are also intuitive to understand and communicate. These are basic principles of architecture design. For component architectures we know several structuring patterns and best practices, described, for instance, in [5, 3, 4, 22]. Layered architectures and pipes-and-filters architectures are well-known examples.

For service-oriented architectures, the question of how to structure a service model and its roles arises. In our approach, we basically follow the same proven principles for designing component architectures with extensions required for handling our more powerful model of roles. Roles capture structural dependencies (decomposed subroles, communication links to other roles); in addition, they can also assume a certain state or condition (such as the *NearestStation* or *ConnectedStation* vs. the *Station* roles). One heuristic we apply in modeling roles is to let the structural decomposition be the guiding principle. The arrangement of roles then follows the classic rules of architecture design. Within the structural framework, we allow for a further refinement of roles with guard conditions and the states roles assume. This heuristic works particularly well, in deployment contexts that are component- rather than service-oriented. If the deployment platform supports service implementations, as exemplified by the web services platform, then the structural decomposition need not be the guiding principle – rather, services *are* the components in such platforms; the roles the can be chosen to represent the external interface of the service, i.e. the behavior of the environment in which the service operates.

Another use for roles relating to the specification of complex control systems is to let them represent *operational modes* of components – in the BART example, for instance, we have used this to describe interactions with the "nearest station". These operational roles represent predicates on the state space of the component implementing the respective role in a service execution.

4 Evaluation and Discussion

The service-oriented methodology introduced in this paper enables us to separate system structure and behavior, as well as interaction behavior and control aspects. We model the computations that need to be carried out to fulfill certain environmental constraints (such as the Worst Case Stopping Profile mentioned in the BART requirements specification) as local activities of system entities that produce output conditions and data – provided that sufficient hardware is in place and all required input data is present. Thereby, we abstract from the actual computations while still being able to react to the pertinent system states. This allows us to separate the development of the communication infrastructure, the system level orchestration of components and the development of control algorithms for the various parts of the system.

For instance, in case the system identifies a hazardous condition requiring immediate attention, it transitions into an emergency state that immediately triggers appropriate reactions. All affected trains get immediately notified of the emergency situation and are commanded to perform emergency braking; all surrounding control stations get notified as well. This behavior preempts the regular operation of cyclically computing the appropriate train movement commands and communicating them to the trains.

In presenting this case study, we have shown how to model the recurrent (cyclic), reactive and continuous behavior of the AATC part of the BART sys-

tem. We have shown how to interface computational results and interaction and state-based behavior of the system. We ave demonstrated how the service-oriented development process can be applied for complex systems that are precisely specified and where extensive safety, convenience and interface constraints need to be met to ensure the reliable, correct operation of the system.

The experience we had in working on the BART case study and on other complex systems, such as in the automotive domain, helped us in refining the service-oriented technique we are developing. To cope with problems where there is a complex control component we developed a way to isolate the control part in local actions of roles. We then just need to guarantee that enough information is available in the role state to enable independent development of the corresponding control algorithms. We found that decomposing the problem using services allows us to focus on the various scenarios separately and address control issues independently by most of the high level system integration effort. Of course, the application of a new service-oriented approach to control problems has the usual drawback of any new methodology: there is a learning curve involved in adopting it. However, we believe that the benefits in tackling complexity that the use of SOA grants is well worth the effort.

The work we have presented in this paper has connections with the work on monitoring end-to-end deadlines we presented in [1] and on the exploration of service-oriented architectures using aspects [13]. In fact, in [1] we used a template-based code generation technique to create code that monitors the deadlines in an implementation of a distributed system starting from a service-oriented specification. The code generator inserts ad-hoc calls to procedures implemented independently by the specific system to verify message deadline expirations. A similar approach can cater to our control problem by calling procedures explicitly named in local actions. The aspect-oriented approach described in [13] converts services to aspects (using AspectJ), defines a component architecture using classes and weaves the aspects into an executable that can be used to evaluate different architectures. This aspect-oriented approach can be used to weave implementations for the control parts into the interaction-oriented framework derived from the service specifications as illustrated in this paper.

5 Related Work

Our approach is related to the Model-Driven Architecture (MDA) [19] and architecture-centric software development (ACD) [23]; similar to MDA and ACD we also separate the software architecture into abstract and concrete models. In contrast to MDA and ACD, however, we consider services and their defining interaction patterns as first-class modeling elements of *both* the abstract and the concrete models. Furthermore, we do not apply a transformation from abstract to concrete models. Our work is related to the work of Batory et al [20]; we also identify collaborations as important elements of system design and reuse. Our approach, in particular, makes use of MSCs as the notation for interaction patterns and is independent from any programming language constructs.

Often, the notion of service-oriented architectures is identified with technical infrastructures for *implementing* services, including the popular web-services infrastructure [21]. Our work, in contrast, supports *finding* the services that can later be exposed either as web-services, or implemented as "internal" services of the system under consideration. Because our entire approach is interaction-based it is perfectly general with respect to the types of architectures we can model.

In contrast to [6], we associate the hybrid behavior with local actions rather than with local states of the roles; this enables us to reuse the automaton synthesis algorithms we have developed in [11] almost verbatim – we just have to introduce transition annotations to represent the calls to the evaluation functions for the control functions.

6 Summary and Outlook

We have applied a service-oriented development process and corresponding notations to a portion of the BART system as a case study, demonstrating the applicability of our methodology to this domain area as well as the power of our approach to manage the complexity of this distributed, reactive system. In the paper we have addressed the problem of creating a service-oriented architecture using a suitable specification language, to describe systems where distributed control is required. Using our interaction-oriented service notion we were able to disentangle the concerns of describing the interactions between entities in the system and the development of control strategies for the various entities. We found our technique to be successful in tackling the complexity of the system class we have explored.

As future work mention updating the existing tools to support a complete and automated development approach for service-oriented systems with substantial control parts, following the process outlined in this paper.

7 Acknowledgments

Our work was partially supported by the UC Discovery Grant and the Industry-University Cooperative Research Program, as well as by funds from the California Institute for Telecommunications and Information Technology (Calit2). Further funds were provided by the Deutsche Forschungsgemeinschaft (DFG) within the project *InServe*. We are grateful to Roshni Malani for her comments on an earlier version of our case study.

References

1. J. Ahluwalia, I. Krger, M. Meisinger, and W. Phillips. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Conference on Embedded Systems Software (EMSOFT)*, 2005.
2. M. S. Branicky. Introduction to hybrid systems. *Handbook of Networked and Embedded Control Systems*, pages 91–116, 2005.
3. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley, 1996.
4. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
6. R. Grosu, I. Krüger, and T. Stauner. Hybrid Sequence Charts. In *In Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*. IEEE, 2000.
7. B. Heck, L. Wills, and G. Vachtsevanos. Software technology for implementing reusable, distributed control systems. *IEEE Control Systems Magazine*, 23(1):21–35, February 2003.
8. ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.
9. F. Kordon and L. Michel, editors. *Formal Methods for Embedded Distributed Systems*. Springer, 2004.
10. I. Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.
11. 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.
12. I. Krüger, R. Mathew, and M. Meisinger. From Scenarios to Aspects: Exploring Product Lines. In *Proceedings of the ICSE 2005 Workshop on Scenarios and State Machines (SCESM)*, 2005.
13. I. Krüger, R. Mathew, and M. Meisinger. Efficient Exploration of Service-Oriented Architectures using Aspects. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.
14. I. H. Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In M. Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer Verlag, 2003.
15. I. H. Krüger and R. Mathew. Systematic Development and Exploration of Service-Oriented Software Architectures. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 177–187. IEEE, 2004.
16. N. G. Leveson. System safety in computer-controlled automotive systems. In *SAE Congress*, March 2000.
17. R. Lim and R. Qu. Control and communication mechanisms in distributed control application platform. In *IEEE International Conference on Industrial Informatics, 2003*, pages 102–106. IEEE, August 2003.
18. R. Mathew and I. H. Krüger. Component synthesis from service specifications. In S. Leue and T. J. Syst, editors, *Scenarios: Models, Transformations and Tools International Workshop, Dagstuhl Castle, Germany, September 7-12, 2003, Revised Selected Papers, Lecture Notes in Computer Science*, volume 3466. Springer, 2005.
19. OMG Model Driven Architecture. `http://www.omg.org/mda`.
20. Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of ECOOP 1998*, volume 1445 of *LNCS*, pages 550–570. Springer Verlag, 1998.

21. J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.
22. D. Trowbridge, U. Roxburgh, G. Hohpe, D. Manolescu, and E. Nadhan. *Integration Patterns. Patterns & Practices*. Microsoft Press, 2004.
23. UML 2.0. `http://www.omg.org/uml`.
24. L. Wills, S. Kannan, B. Heck, G. Vachtsevanos, C. Restrepo, S. Sander, D. Schrage, J.V.R., and J. Prasad. An open software infrastructure for reconfigurable control systems. In *Proceedings of the 2000 American Control Conference. Vol4*, pages 2799–2803, 2000.
25. V. Winter, F. Kordon, and L. Michel. The BART Case Study. In F. Kordon and L. Michel, editors, *Formal Methods for Embedded Distributed Systems*, pages 3–22. Springer, 2004.