Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems

Vina Ermagan[1], To-Ju Huang[1], Ingolf H. Krüger[1],
Michael Meisinger[2], Massimiliano Menarini[1], Praveen Moorthy[1]

[1] Department of Computer Science
University of California, San Diego
La Jolla, CA 92093-0404, USA
{vermagan,t3huang,ikrueger,mamenari,pmoorthy}@cs.ucsd.edu

[2] Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
meisinge@in.tum.de

Abstract: The development of embedded systems is a challenging task because of the distributed, reactive and real-time nature of such systems. Distribution of embedded components across buses and networks causes high interaction complexity. We propose a model-based development approach to handle this complexity. We model the individual functionalities of the system – the services – independently from each other in an interaction modeling and architecture definition language. Methodological steps allow us to refine and modify the models. A development process determines the order in which to perform the steps. Our service-oriented development methodology spans the entire development process from requirements analysis to implementation, verification and validation. We have developed integrated tool support that governs this process; it provides an effective means to apply and evaluate our approach. In this paper we introduce our service-oriented methodology and describe our tool as part of an integrated tool suite supporting this process by means of an automotive example: the Central Locking System (CLS).

1 Introduction

Designing complex distributed systems is a difficult task. Systems like these can be found in application domains such as avionics and automotive control systems – in the form of embedded systems – and in telecommunications and business information systems. Also sensor networks and mobile applications are growing areas for complex distributed systems. The common property among all of these systems is that distribution and complex interactions between distributed nodes are key enablers of their success. Distribution makes the system architecture much more modular and decentralized, which contributes to fault tolerance, component reuse, system robustness, maintainability and further positive system properties.

However, heavily distributed systems are among the most complex man-made artifacts known; the high degree of distribution makes development of these systems very challenging. The number of states and conditions – for functional behavior as well as for error conditions – increase exponentially with the number of distributed nodes. Furthermore, logistics and maintenance problems originate from the distribution of entities. In fact, nodes can be developed, maintained, extended and replaced independently, potentially harming the overall system integrity and consistency.

1.1 Problem Definition

Important questions that need to be addressed when designing distributed systems and their architectures include, for instance, what is the functional behavior of the system i.e. what are the individual services that the system and its parts offer to the environment and how are services connected to provide the full system functionality. Important further questions are how to design the objects/components of the system and their interfaces so that they can provide the identified services with the required quality properties; how to connect and distribute them i.e. how to design the communication topology, and how to replicate components for most efficient operation on a given middleware. Answering these questions requires a systematic, iterative approach in system design. Often, this involves changes to the designed system behavior and consideration and exploration of different alternative architecture candidates that can provide the designed services.

Model-based development is a promising approach to mitigating the difficulties and complexities of developing a system from requirements analysis to system execution [Bro05]. Models are abstractions of reality that are targeted to express specific views on the system serving a specific purpose. Different models exist at different stages of the development process. Having a collection of complementing models, it is possible to understand, design and modify concepts that are otherwise too complex to handle. If the models used for system development are integrated and consistent, and if there is a systematic process from models of high abstraction through refinement to the realized system, we speak of model-based development. The degree of precision, the level of formality and the used modeling notations and concepts vary significantly from one model-based approach to another.

One model-based approach to designing distributed embedded systems is to separate an overall system model into logical models (also called *domain models* [Eva03]) and implementation models; approaches advocating this separation are architecture-centric software development [OMG05] and model-driven architecture [OMG03]. The logical models describe functionality, distribution of components and quality properties independently of implementation details and deployment architecture design decisions. The implementation models are consistent refinements of the logical models and contain these details and decisions. The advantage of this separation is that one logical model can be refined into many implementation models. This provides an independence of system functionality from the actual deployment architectures, for instance to defer design decisions or to switch to a different architecture.

A clear separation into logical and implementation models is often difficult to achieve – especially in situations where requirements suggest a tight coupling between the two types of models. This is often the case when requirements include specific performance and other Quality-of-Service properties. A core source of complexity is that the scenarios supported by the system typically involve a multitude of collaborating entities partaking in complex interactions. These interactions are part of both the logical and implementation models. A well-defined mapping needs to exist between the interactions in both models.

Our goal is to provide a solution where a logical system behavior model can be reused unchanged across all implementation models (or target-architectures). The major step toward achieving this goal is to decouple the "features" or "services" a system provides from the architecture on which it is deployed.

A development approach must be supported by efficient tools to provide a practical application environment. The methodology establishes the formal and logical basis on top of which tools need to leverage the practical problems and day-to-day routines. Tools automate methodological steps and enable graphical, iterative system modeling, development and verification. Different tools need to be integrated in order to provide seamless modeling support, a high degree of automation and short turn-around times.

1.2 Service-Oriented Specifications

In this paper, we propose an approach to service-oriented development of distributed embedded systems that establishes a clean separation between the services provided by the system under consideration, and the architecture – comprised of components and their relationships – implementing the services. Our approach is well suited for tool support, which we will explain subsequently.

We use 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 Engineering community [TRH$^+$04]. *Web Services* [STK02] currently receive a lot of attention from both academia and industry, but services are also emerging in embedded systems. Fig. 1 shows a typical "layout" of applications composed of a set of services. Often such systems consist of at least two distinct layers: one *domain layer*, which houses all domain objects and their associated logic; and one *service layer*, which acts as a facade to the underlying domain objects – in effect offering an interface that shields the domain objects from client software. For an embedded system, the service layer, for instance, consists of the functions this system exposes to the environment by exposing an access interface on a broadcast bus network, such as CAN and FlexRay. The domain layer consists of "plain old objects" representing the data and logic of the underlying implementation. 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 electronic control unit (ECU), whereas others, such as *Service n* may be accessible remotely via the bus network.

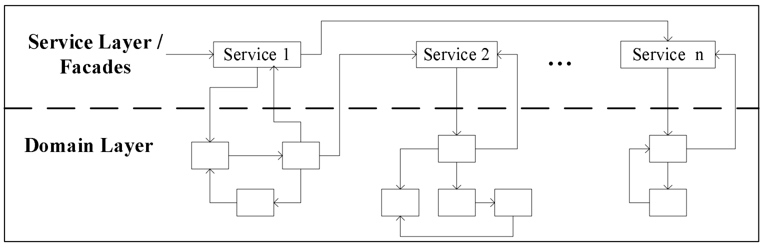

Figure 1: Service-Oriented Architectures

The layout shown in Fig. 1 is prototypical for many domains where complex, often distributed applications are expected to offer externally accessible interfaces. Indeed, service-oriented approaches to system development, leading to similar application structures, are prominent for business information systems using web services [STK02] and in the telecommunications domain [Zav01] and are emerging in the automotive domain [BKM06a, BKM06b]. Abstracting from the domain-specific details we observe that services often encapsulate the coordination of sets of domain objects to implement "use cases".

We view services as specializations of use cases to specify interaction scenarios; services "orchestrate" the interaction among certain entities of the system under consideration to achieve a certain goal [Eva03]. In contrast to use cases, which describe functionality typically in prose and on a coarse level of detail, we define a service via the interaction pattern among a set of collaborators required to deliver the functionality. Services are partial interaction specifications. For a formal definition of the service notion, see [KMLS05].

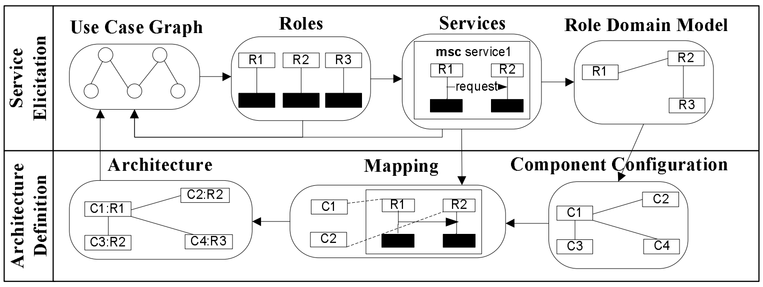

Figure 2: Service-Oriented Development Process

We employ a two-phase, iterative development process as shown in Fig. 2. Phase (1), *Service Elicitation*, consists of defining the set of services of interest – we call this set the service repository, our logical model. Phase (2), *Architecture Definition*, consists of mapping the services to component configurations to define deployments of the architecture – the implementation model.

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 large-scale, scenario-based view on the system.

From the use cases, we derive sets of *roles* and *services* as interaction patterns among roles. Using roles decouples from interaction details, because roles abstract from components or objects. 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 typically will play multiple roles at the same time. The relationships between the roles, including aggregations and multiplicities, develop into the *role domain model*.

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.

The 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.

1.3 Contributions and Outline

As main contribution, this work presents a systematic approach to the development of distributed, reactive embedded systems and an integrated tool suite we have developed that supports this approach throughout the entire development cycle. We present the application of our modeling approach and the accompanying chain of tools by means of a running example through all stages of development.

In particular, we explain the purpose and depencencies of the individual tools: We use our SODA tool to track requirements of service-oriented systems and generate process required documentation. Our M2Code tool models interaction patterns that define services. The Component Synthesizer of M2Code generates state machines for the modeled services. These state machines can then be transformed into executable code (using RT CORBA CodeGen and M2Aspects), and verified for correctness against the specification (using S2Promela, ServiceDebug and MSCCheck).

In Sect. 2, we introduce the automotive Central Locking System (CLS) as our running example and show how it is modeled in terms of services. In Sect. 3, we show how we make use of tools to support our approach. We present our integrated tool landscape and explain the relevance of specific tools in the process. In Sect. 4, we report on experiences for CLS applying our approach and the tools; we also provide a brief discussion. In Sect. 5 we show related work. Sect. 6 contains conclusions and an outlook.

2 Service-oriented model of CLS

To demonstrate our approach, we use the Central Locking System (CLS), a well-studied and documented example of one automotive vehicle functionality. The CLS integrates a multitude of separate subsystems in the vehicle, ranging from safety critical ones (motor control and crash sensors) to comfort functions (automatic seat positioning and tuner presets in luxury vehicles). For reasons of brevity, we present a simplified and abstract adaptation of the CLS. We direct the reader to [NP03, KNP04] for a more comprehensive description. Here, we focus on some specific use cases during the locking and unlocking of the vehicle: *operation of locks*, *signaling*, *transfer driver ID*, and *impact sensing*.

The Central Locking System in the described form acts as a representative for similar problems in automotive control electronics and distributed, reactive systems in other application domains. For instance, business information systems with distributed components communicating via web services, and database systems implementing distributed transactions by two-phase commit protocols share many of the same properties and challenges; thus, the example and approach presented in the following, provide telling insight for these domains as well.

The first step in the service-oriented process is analyzing the requirements, which leads to a number of use cases and actors. The main parts of the CLS system are a remote key fob and a controller within the car, which receives the lock and unlock command signals from the key fob. The controller also interacts with the lighting system in order to operate the lights, the security module, in order to validate a driver's identity by checking the key fob's secure identity token, and the door locking subsystem in order to lock and unlock the doors. In addition, upon impact, an impact sensor will send a signal to the controller. For simplification, we will abstract away the complications of the door locking subsystem by introducing a Lock Manager, which will act as an interface for locking or unlocking the doors.

In the following, we explain in detail how to specify the CLS in our service-oriented modeling approach. We follow our development process, introduced above, to define, implement and verify architectures for distributed, reactive systems. Along the way we introduce our Architecture Definition Language (Service-ADL) that we use for specifying roles, services and architectures; the details of this notation are documented in [KM04, Mat04].

After capturing the use cases and actors, the first step towards a service-oriented system specification is to identify the participating roles, which emerge out of the found actors within the use cases. We identify key fob (KF), controller (CONTROL), lock manager (LM), security module (SM), lighting system (LS), database (DB), which holds the information for each driver ID, and the impact sensor (IS). These roles are the logical entities in our system that communicate locally or over the network to provide the required system functionalities. Fig. 3 shows the role definitions in our Service-ADL.

The next step in our process is to elicit the services that the system needs to support based on the found use cases. For instance, the service *Vehicle Unlocking* involves the communication between the key fob and the controller, which, in turn, communicates with the lock

```
role CONTROL
  description
    Entity acting as a controller for all CLS functions
  states
    (UNLD: Doors unlocked) (LCKD: Doors locked)

role KF
  description
    Represents the receiver unit for a driver's key fob
  states
    (INITIAL: initial state)

role DB description Database component ...
role IS description Impact sensor ...
role LM description Lock manager ...
role LS description Lighting system ...
role SM description Security module ...
```

Figure 3: Role Definitions in Service-ADL

manager (for physical door unlocking), the lighting system (for flashing the lights) and the security module (to validate and store the driver id). Fig. 4 depicts the communication connections between the roles for all services in the CLS example, as part of the service repository definition in Service-ADL.

```
service repository CLS
  description
    Central Locking System
  roles
    (KF)(CONTROL)(SM)(LM)(DB)(IS)(LS)
  services
    (CLS-0)(CLS-1)(UNLK-e)(LCK)(LCK-1)
    (LCK-2)(UNLK)(UNLK-1)(UNLK-2)
  role domain model
```

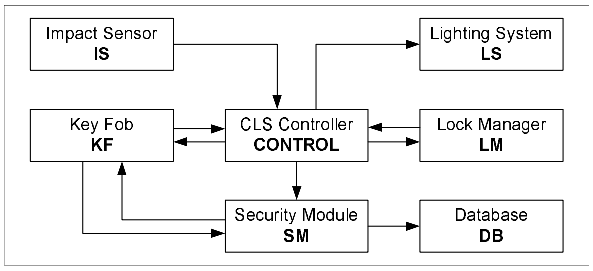

Figure 4: CLS Service Repository Definition

We specify the services of the CLS system using a notation based on Message Sequence Charts (MSC) [IT96, Krü00, OMG05]. An MSC defines the relevant sequences of *messages* (represented by labeled arrows) among the interacting *roles*. Roles are represented as vertical axes in our MSC notation. Fig. 6 and 7 show the specification of several services as interaction patterns. The MSC syntax we use should be fairly self-explanatory, especially to readers familiar with UML2 [OMG05]. In particular, we support labeled boxes in our MSCs indicating alternatives and conditional repetitions (as bounded and unbounded loops). Labeled boxes *on* an axis indicate actions, such as local computations; diamond-shaped boxes on an axis indicate state labels. High-level MSCs (HMSCs) indicate sequences of, alternatives between and repetitions of services in two-dimensional graphs – the nodes of the graph are references to MSCs, to be substituted by their respective interaction specifications. HMSCs can be translated into basic MSCs without loss of information [Krü00]. Fig. 5 shows the use of HMSCs in the CLS example. The left HMSC shows the infinite sequence of locking followed by unlocking back to locking, and the right side shows an HMSC defining the *Vehicle Unlocking (UNLK)* service as a composition of two sub-services.

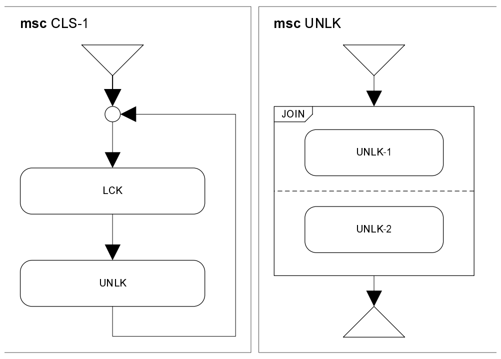

Figure 5: CLS High Level Services CLS-1, LCK

We model the CLS *Vehicle Unlocking* service by joining two modular sub-services (interaction patterns): UNLK-1 and UNLK-2. Fig. 6 shows the UNLK-1 service, which captures the *Operation of Locks and Signaling*. Upon receipt of the *unlck* message from KF, CONTROL issues an *unlck* message to LM. Once LM acknowledges this with an *ok* message, CONTROL requests signaling of the unlocking from LM by means of a *door_unld_sig* message. Once it has issued this message, CONTROL sends an *ok* message back to KF. The *Transfer Driver ID* service – storing the driver's id for further access – is also triggered by the *unlck* message from KF to CONTROL, and is captured in UNLK-2. The corresponding interaction pattern is shown in Fig. 7 as part of a screenshot of our service modeling tool *M2Code*. In a subsequent iteration of the development process we could use the *SODA tool* (see Sect. 3.2) to capture the requirement that a failed security check will *not* unlock the vehicle; consequently the UNLK-1 service could then be modified to include an interaction with SM *before* the locks are operated.

A number of extensions to the standard MSCs warrant explanation [Krü03, KM04]. First, we take each axis to represent a *role* rather than a class, object, or component. The mapping from roles to components is a design step in our approach and will be described below. Furthermore, we use an operator called *join* [Krü00, Krü03], which we use extensively to compose *overlapping* service specifications. We call two services *overlapping* if their interaction scenarios share at least two roles and at least one message between shared roles. The join operator will *synchronize* the services on their shared messages, and otherwise result in an arbitrary *interleaving* of the non-shared messages of its operands.

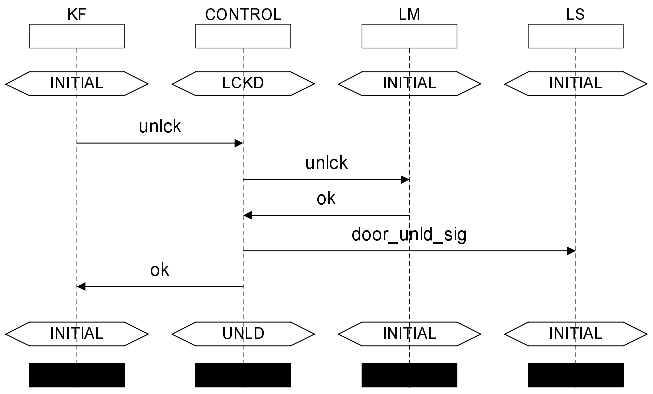

Figure 6: UNLK-1: Operation of Locks & Signaling

Join is a powerful operator for separating an overall service into interacting sub-services. The availability of such an operator also distinguishes our approach from many others in literature. For instance, we use the join operator (as seen on the right side of Fig. 5) to compose the described two unlocking services from Fig. 6 and 7.

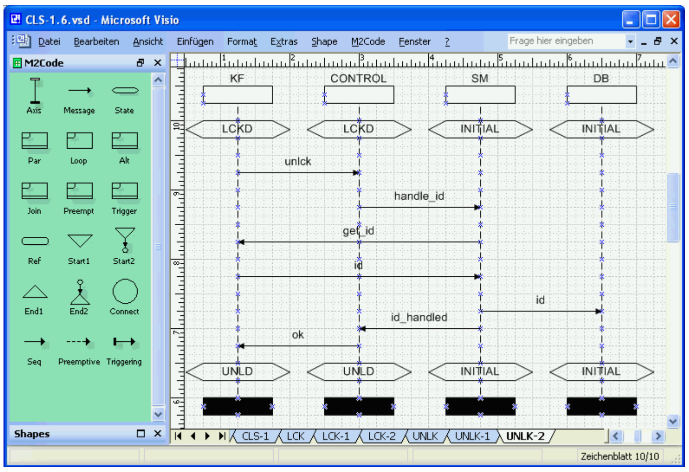

Figure 7: UNLK-2: Transfer Driver ID (screenshot)

To specify preemptive behavior, we introduced the *preempt* operator [Krü00, Krü03]. For CLS, the vehicle must also unlock in case of a crash impact. Upon impact, IS will send an impact signal to CONTROL. At this point, the routine interactions should be preempted, and CONTROL should immediately unlock the doors by sending an unlock message to LM and receiving its acknowledgment. Fig. 8 gives an example for the use of the preempt operator. Both triggering multiple services with the same message and preempting a set of such complex services with another critical service are powerful capabilities of our approach, as shown in the example.

service UNLK-e
 description
  Routine operations (CLS-1) are interrupted when an impact is detected
 interaction

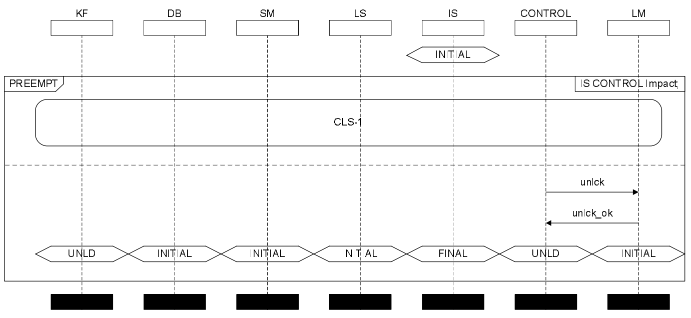

Figure 8: Adding Emergency Unlocking using Preemption

So far, we modeled the logical behavior of the system via services, independently of any underlying deployment architecture. This approach maximizes flexibility and reusability of the logical model. The next step after eliciting the services is to define a suitable component architecture that can provide these services and to define a mapping from the logical model to the deployment model, which includes the mapping of roles to components. We must make sure that the architecture observes the dependencies of the roles and further constraints given by the requirements. Fig. 9 shows how we specify components in our Service-ADL. When a role is mapped to a component, that component plays the mapped role. Intuitively, "playing a role" in an architecture means implementing all interactions in which this role partakes. Multiple roles can be mapped to the same component. The more roles it plays, the more functionality it implements. Interactions between different components usually mean expensive distributed communications, while interactions between roles within one component can be implemented very efficiently as subroutine or method calls. Also, we can map the same role to multiple components, indicating a replication of that role.

```
component type ImpactSensor
 description A sensor to detect an impact
 plays (IS)
 in service (UNLK-e)

component type MainControl
 description The main controller of the CLS
 plays (CONTROL)
 in service (UNLK-1)(UNLK-2)...
...
```

Figure 9: CLS Component Type Definitions

Fig. 10 shows a potential deployment architecture for our CLS case study, defined in our Service-ADL. As shown in the figure, the IS role is played by two different components, meaning that it is replicated, due to its criticality, in order to improve reliability. Also part of the architecture configuration is the mapping of role communication channels to networks, in our case a “Wireless” network and a “CAN Bus”. Later in this paper, we will use this deployment architecture to demonstrate how we can verify architectures for distributed reactive systems. For more information about the architecture mapping and architecture exploration, see [KMM06].

```
architecture CLS Architecture1
 service repository CLS
 description
  Defines a deployment architecture for the CLS logical model
 component types
  (ImpactSensor)(MainControl) ...
 components
  (ImpactSensor1:ImpactSensor)
  (ImpactSensor2:ImpactSensor)
  (Controller:MainControl) ...
 implements
  (UNLK-e) (UNLK-1)(UNLK-2) ...
 configuration
```

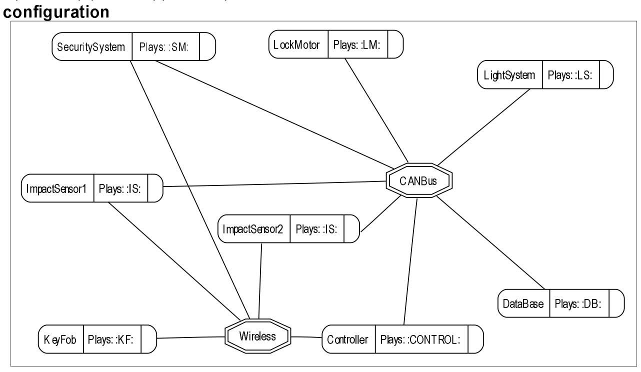

Figure 10: CLS Deployment Architecture

Based on the service repository specification and the deployment architecture definition, we can now simulate the model, verify and model check it, and generate code for prototypical implementations.

3 Integrated Tool Support

In order to demonstrate and experiment with service-based development, we have designed and implemented several novel prototypical tools. Together they form an integrated tool landscape or tool chain. The purpose of this tool chain is to illustrate and support the complete development cycle, from the initial modeling phase to execution on real systems.

3.1 Tool Landscape Architecture

The tool landscape consists of a number of tools supporting interaction-based and service-oriented development. Fig. 11 gives an overview of some of our tools and their dependencies. We divide the tool landscape roughly by the stage in the overall development cycle: *Requirements Elicitation*, *Service Specification and Architecture Design*, *Implementation*, as well as *Deployment and Verification*. In the following sections, we will describe the tools according to their location in the development stage; for a few of the tools we will present more extensive descriptions.

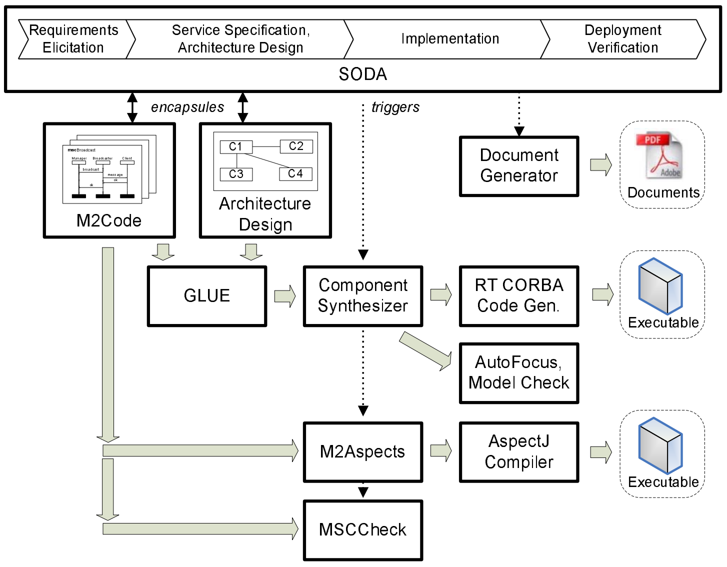

Figure 11: Service-Oriented Tool Landscape

The overall dependencies are as follows: The *SODA tool* tracks requirements of service-oriented systems and generates requirements and architecture documentation. *M2Code* provides facilities for intuitive graphical service modeling with MSCs and HMSCs. M2Code also contains a *Component Synthesizer* to generate state machines for all components partaking in the modeled services. The GLUE tool maps the component state machines to more complicated target architectures, for instance with replicated component instances. The component state machines are the basis for code generation of executable prototypes (using RT CORBA CodeGen and M2Aspects) and verification by model checking (using S2Promela, ServiceDebug and MSCCheck).

3.2 Tools for Requirements Elicitation

We support the process of *Requirements Elicitation* by means of two tools: the *SODA tool* and *M2Code*.

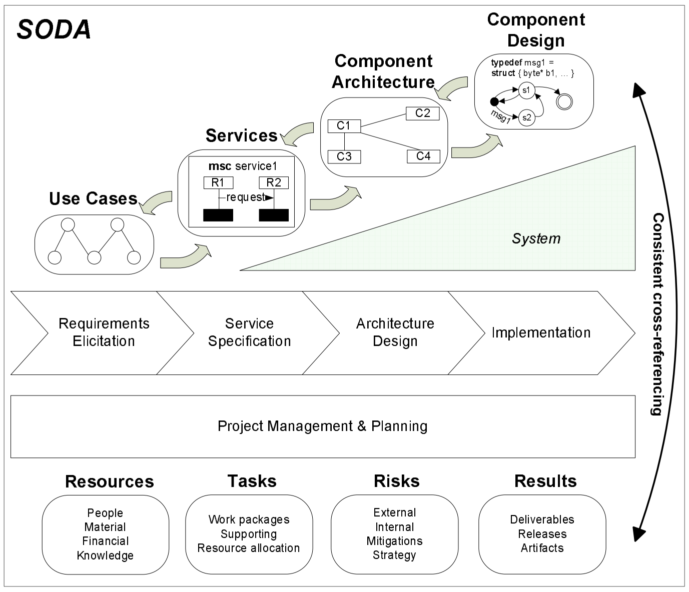

Figure 12: SODA Tool Overview

The *SODA tool* (Service-Oriented Design and Architecture Development) governs the development process and provides project support. It has facilities for managing project resources and tracking project progress. *SODA* embeds and triggers other tools and has the capability of generating project relevant documentation and reports. *SODA tool* is a platform for capturing and cross-referencing information about a system or software under development (see Fig. 12 for a schematic overview). Information is structured according to a changeable domain model and is stored as XML with associated data files. *SODA* contains a generic XML editor that enables editing data documents based on arbitrary XML schemas. We designed an XML schema that contains elements for service-oriented development projects, such as “Use Case”, “Textual Requirement”, “Service”, “Component” and many more. The schema designer can further structure and specify types and attributes for each of these elements Based on the type of element or attribute, specific data editors are invoked. *SODA* provides a powerful linking and cross-referencing facility that applies to all elements, as defined in the XML schema. *SODA* provides powerful mechanisms for editing, linking and post-processing.

In our reference data model for service-oriented development projects, the system developer enters requirements as hierarchically structured use cases, which contain free form text and embedded figures. Use cases are linked to structured textual requirements; both use cases and requirements are additionally linked to the services of the system in development. All the before mentioned object types and their cross-references are defined in the underlying XML schema.

SODA has a plug-in architecture that enables consistency checks of the currently edited data model. For instance, *SODA* can detect referencing errors (such as link cycles) and missing references (such as a use case with no linked service). *SODA* also has powerful plug-ins for generating LaTeX, PDF, HTML and other type documents from the model. We use this to generate requirements and software architecture documents out of the data model.

The second tool supporting requirements elicitation is *M2Code*. It is our service modeling tool and will be described in detail in the next section. For requirements gathering we use high level services - in form of HMSCs - without providing detailed service specifications and interaction patterns.

We are currently developing the link between *SODA* and *M2Code*, so that services can be edited directly and stored within *SODA*'s system data model.

3.3 Tools for Service Specification and Architecture Design

A central part of our tool landscape are the modeling and specification tools. *M2Code* enables the system designer to specify functionalities as a complex interplay of logical components (called *Roles*) exchanging messages. Each system functionality is modeled separately from the others. Those functionalities are called "services" in our terminology. Services can be structured hierarchically and referenced from other services. This enables us to design service-oriented applications in a modular way, which results in reusable service definitions. Simpler services are combined to form more complex ones providing advanced functionalities by means of composition operators.

M2Code (see Fig. 7 for a screenshot) is the principal modeling tool of our tool set. It allows modelers to specify system functions (services) using Message Sequence Charts (MSCs). Additional information, such as real time constraints, logical to deployment mappings, failure hypothesis, etc. can be specified using other tools to enrich the model specified by M2Code.

The tool is implemented as a Microsoft Visio plug-in. This architecture allows us to leverage the powerful design and export capabilities of Visio and include interaction specifications in various types of documents. The plug-in provides graphical elements to represent Message Sequence Charts, High Level Message Sequence Charts (HMSCs) and the various operators we use. Fig. 7 for instance, depicts an MSC designed with M2Code. Apart from the graphical capabilities that enable us to design specifications with MSCs and HMSCs, M2Code contains the *Component Synthesizer* to generate state machines and role structure diagrams out of specifications.

We make use of role structures and state machines for the individual components [KGSB99, AKMP05] in other tools for various purposes. The role structure captures the communication links between the roles participating in the specified services.

The generated state machines, one for each role, capture the local behavior of each participant. They define how each node of a distributed system engages with its environment to carry out the specified services. Role structures and state machines are exported to an XML file.

To demonstrate the use of our tools in addressing the needs of the Automotive industry we have used M2Code to model the CLS example described in Sect. 2. We advocate an iterative refinement process that must be supported by powerful language constructs and adequate tools to be effective. Fig. 8 exemplifies how M2Code supports iterative service-oriented development: we use the *PREEMPT* operator to enrich an initial service specification (defined in the referenced service *CLS-1*, see Fig. 5) with emergency unlocking behavior (defined in the lower part of the graph). The parameter on the right part of the *PREEMPT* box specifies that a message "*Impact*" can be sent by the role *IS* to the role *CONTROL* at any moment. If the message is sent, the normal interaction defined by *CLS-1* is interrupted and replaced by the one defined in the lower part of the *PREEMPT* box. In the example if the "Impact" message is sent the car doors unlock.

This simple example shows how, with the right set of tools and languages, service-oriented development can be supported even in application domains not supported by service-oriented middleware. Once the two services are composed, M2Code will then take care of synchronizing all roles participating in the interaction and ensure that the specified logical behavior will be observed in the generated state machines.

M2Code can directly generate state-machines out of service-oriented specifications if the target deployment architectures are simple and roles are mapped one-to-one to executable components. In case of more complex architectures, we have designed the *GLUE tool* [Rus06], which takes the service specifications of M2Code as an input, together with the deployment architecture specification in Service-ADL (see Fig. 9 and 9). *GLUE* will then generate MSCs for each component instance with unique message identifiers, for further processing by a code generator etc.

3.4 Tools for Implementation

We have implemented two tools that take a service-oriented specification and generate executable prototypical implementations out of it: *RT CORBA CodeGen* and *M2Aspects*. *RT CORBA CodeGen* is a template-based code generator. Currently it targets mainly the RT CORBA [Obj02] infrastructure. However, the flexible template-based architecture enables easy ports to other infrastructures and middlewares. We utilize the code generator to create prototypes of our architectures for simulation and validation. This includes runtime verification of Quality of Service (QoS) properties [AKMP05]. In contrast to other simulation tools we have at our disposal, for instance AutoFocus [Aut06], the RT CORBA based runtime system we have implemented provides monitoring and validation mechanisms for *both* logical flow *and* real-time properties.

The code generator itself is written in Java and uses an XML file generated by M2Code as input. To bridge the gap between the abstract XML specification and the executable code, we have developed a runtime library that uses the facilities provided by the RT CORBA platform. Our runtime system was intentionally kept simple and straightforward by using the Real-Time Event Service (RTES) messaging facility. The RTES provides the fundamental abstraction for asynchronous message passing, enabling each component to operate independently. We have also made use of hooks the RTES provides to incorporate a real-time scheduler. The Time Service provides a distributed, synchronized, global clock to all components in the system. Mechanisms for globally synchronized clocks exist on many embedded platforms and so the runtime system is easily portable. For instance, we could have an implementation that targets a CAN (Controller Area Network) bus for embedded applications and another targeting an enterprise service bus for large federated corporate networks.

The code generator currently provides two templates, supporting two different execution models: *Synchronous* and *Asynchronous*. Both generate C++ code for the RT CORBA platform. The Synchronous execution model operates in a time-synchronous mode; messages are exchanged via one-place buffers between components. Each component waits for inputs to arrive on *all* of its input buffers, then executes an enabled transition of its associated state machine, and finally writes to all of its output buffers; this scheme is further described in [HSE97]. The Synchronous execution model is supported by the validation and verification tool AutoFocus [Aut06]. This, however, results in "lock-step" executions of the components that have to be coordinated by an abundance of control messages on the communication medium.

To better cater to the reactive environment in the automotive domain we have developed the *Asynchronous* execution template. It implements an execution model where, upon receipt of a message, each component immediately executes an enabled transition and sends output on the appropriate ports. This model eliminates undesirable "waiting" as well as network flooding by control messages. We have developed a set of tools providing validation and verification of systems developed using the asynchronous execution model: one tool (S2Promela) is supporting formal verification and one (ServiceDebug) does testing. A third tool (MSCCheck) is currently work in progress; it will allow us to perform formal verification in a compositional fashion. The goal is to be able to verify also implementations and models that are too complex for the current tools.

Besides the *RT CORBA CodeGen*, we have developed *M2Aspects* to efficiently generate executable aspect-oriented implementations of service models [KMM06]. *M2Aspects* translates services into aspects; the aspect-weaving capabilities of the AspectJ compiler then create the executable implementation. We use multiple such prototypical implementations of the same service repository but with different target architectures for quick architecture exploration and validation [KMM06].

3.5 Tools for Verification and Validation

We have different options for validating M2Code models. AutoFocus [Aut06] is a good solution to address the synchronous model of execution. In the automotive domain, however, we usually prefer the asynchronous one. *S2Promela*, *ServiceDebug* and *MSCCheck* are our solutions for the asynchronous domain. All these tools work on the XML files generated by M2Code and provide different facilities to analyze the models.

ServiceDebug (Fig. 13) uses the code generation facilities described in the previous section and steps through the execution of each component via an interactive graphical interface. To support debugging of models, the tool shows a graphical representation of the state machines and their current states. The tool allows the user to choose the order each transition is taken by all state machines and even to inject messages on behalf of some component.

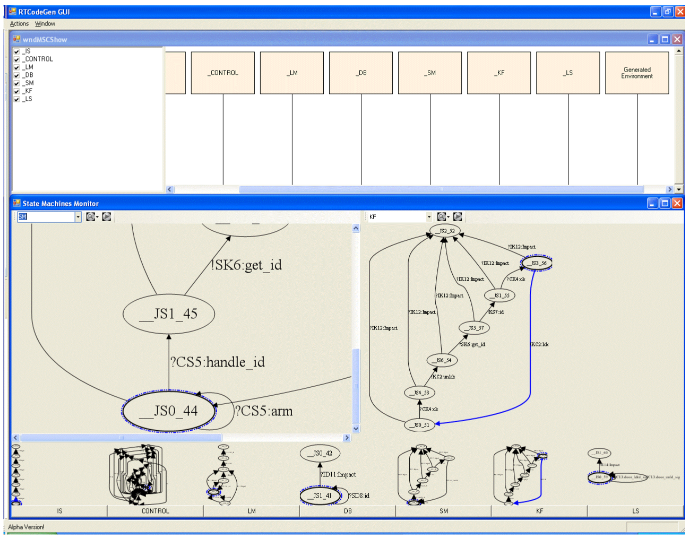

Figure 13: ServiceDebug User Interface

Another approach to validation applies formal verification techniques. To this end we have developed a prototypical tool able to translate the M2Code-generated component state machines to a suitable input file for a model checker. In particular we have chosen to generate a Promela source file that can be verified by the SPIN model checker [Hol03]. The tool is called S2Promela and is written in Java.

As described before, the service-oriented approach we apply is based on a rich ADL that provides two different description levels: a logical one, where the interactions between abstract roles are defined, and an implementation one, where logical entities are mapped to physical nodes of the target system. Our S2Promela tool uses models obtained from M2Code, by means of the XML interchange file, and additional information about the mapping of the service model to a deployment architecture, to generate a Promela description of the system.

Promela allows us to easily model concurrent programs communicating by message passing. We use the language facilities (*proctype* and *chan*) to map elements from our model to Promela constructs. In our translation each role is converted to a Promela *proctype* definition. The *proctype* parameters are *chan* variables representing the input/output channels of the corresponding role. The translation of the state machine for each role is performed according to the schema suggested in [Hol03]. Each state is encoded by a label in the *proctype* and each transition by a guarded goto statement. Multiple enabled transitions are non-deterministically selected by using if-statements.

Each *proctype* defines a template for the behavior of a role. To activate each role and to allow it to perform a given function the *proctype* must be instantiated by means of a run-statement in the initialization (*init*) process. The process of initialization allows us to configure a system as required by the deployment model. In our CLS case study, for instance, there are two physical components playing the role IS (Fig. 4). This will be translated in two run-statements one for each instance of the role. Channels are instantiated and mapped accordingly to the deployment architecture.

The resulting Promela can be used to verify properties with the SPIN model checker. The properties to verify can be specified in all the ways accepted by SPIN. We have experimented with two ways of specifying properties. The first, is using LTL formulas. SPIN is able to convert those formulas to *never* claims which are included in the Promela file to perform the verification. This strategy for specifying properties is more indicated to specify constraints over the states of the automata generated by M2Code. For instance, we could express that the CONTROL Role mapped to Controller component will eventually reach state UNLD when one IS Role is in FINAL state. However, using LTL formulas is difficult in this setup to verify constraints on the messages exchanged on the various channels. For this reason we use another Promela construct: the *trace/notrace* command. Using these commands it is easy to express the interaction pattern to verify.

In our CLS example we have verified the property that when an *Impact* message is sent by an *ImpactSensor*, eventually an *unlck_ok* message is sent by *LockMotor*. The encoding of the property is:

```
notrace{  do  ::IC10!Impact;
                    do
                       ::IC10!Impact;
                       ::LC1!lck_ok;
                    od
              ::LC1!lck_ok;
              ::LC1!unlck_ok; od }
```

The *notrace* statement raises an error during verification if the sequence of messages is possible in any run. The fact that the verification did not return any error signifies that there exists no trace where the *Impact* message is sent and an *unlock_ok* does not follow. The SPIN model checker has explored and stored more than 11 million states to prove the correctness of our assertion and has used 1.7 gigabytes of RAM.

4 Experiences and Discussion

We have applied our service-oriented approach extensively to various projects, from the automotive domain to sensor networks to enterprise integration architectures [KMMP06].

We have used M2Code extensively in developing these projects. With the exception of the trigger operator, which is used for liveness specifications only, all of the operators mentioned above have been implemented. Our model-based approach also provides means to specify cross-cutting QoS properties of interaction-based scenarios on all levels of granularity, from an entire service down to a single interaction. In addition, it is a good platform for further tool integrations. We have successfully used the RT CORBA CodeGen tool to generate distributed prototypes and monitor real-time properties of case studies in the automotive domain.

We have used our S2Promela and ServiceDebug tools to verify critical properties of real time automotive domain projects. Because all tools make use of XML as the data exchange format, we can readily chain the mentioned tools, creating a tool chain from interaction definitions to code generation, runtime property monitoring, verification and model checking. Our service-oriented process is well-suited for building *fail safe* real-time systems, by concentrating on services as the main modeling elements. Failures are mostly cross-cutting; separation of logical and deployment models in our service-oriented process gives us the capability to lift failures from the deployment layer up to the logical layer, addressing them at the crosscutting service level. Building failure management into our process and tool chain is currently in progress.

5 Related Work

Triggered by its success in the telecommunications domain [Par02, ITU06] the term service has become quite prevalent in the literature, especially in the context of "web services" [STK02]. So far, however, services have been used mainly as an *implementation* concept, not as a first-class *modeling* entity. Consequently, existing definitions for the term service capture only syntactic lists of operations upon which a client can call. These definitions are inadequate for a systematic treatment of services throughout the development process. This is especially true also for the UML [OMG05] or SysML [Sys06], which do not recognize services as separate modeling entities. In our approach, the interaction-based service notion emerges as a cross-cutting modeling element regarding both system structure and behavior. In particular, we have established a decoupling between services (functions) as modeling elements and implementation infrastructures on top of which services can be implemented. We use a generalized notion of a system service in our interaction-based modeling approach. In [KNP04, KMM06] we present our service modeling approach based on the modeling of role interactions. It is related to the role concept introduced in [Pae97] and the activities of [KM96]. While our service concept is based on interaction patterns, stressing the cross-cutting nature of services, the roles of [Pae97] describe projections of such patterns onto individual components; to yield the overall picture the latter have to be recomposed into a global interaction specification. Activities of [KM96] capture global interaction properties as we do in our service definition; in contrast to our approach, however, [KM96] views activities as classes and roles as extensions to these classes.

The component-based development approach [Szy02] certainly has many advantages, including support for encapsulation, modularity, defining a unit of deployment, fault-containment, and many more. However, it falls short for cross-cutting aspects including interaction patterns. In contrast, by establishing a decoupling between service modeling and deployment of the resulting services on top of component architectures, we allow for "late binding" between functionality and components. Services provide a level of abstraction higher than components because services hide the components that implement them. This induces a choice regarding the architecture on top of which the services are implemented.

Our approach allows for the system to be understood at the granularity of individual features instead of components. The ability to gracefully deal with faults, both predictable and unpredictable, is an important property of embedded systems. Although we have not elaborated on this topic here, our approach allows for a better understanding of failures that emerge from the interplay of multiple components; the component-based approach accounts for faults localized to individual components.

Our approach is related to the Model-Driven Architecture (MDA) [OMG03] and architecture-centric software development (ACD) [OMG05]; 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 model. Our work is related to the work of Batory et al [SB98]; we also identify collaborations as important elements of system design and reuse. Our approach in particular makes use of MSCs as notation and is independent from any programming language constructs.

Architecture and tool support are the key instruments to address the complexity of real-time distributed systems. In [GH06], for instance, a framework is presented that allows replay of distributed real-time system based on architecture models. Our RT-CORBA CodeGen tool leverages similar model information to generate code for monitors that run in parallel to the system and inform the developer of unsatisfied deadlines.

Many graphical tools supporting software modeling have been researched and implemented. In [LMB$^+$01], for instance, the Generic Modeling Environment (GME) is presented, a tool aiming to support many modeling paradigms thanks to meta-model based configurations. M2Code, on the other hand, focuses on extended MSCs and HMSCs – features not currently supported by GME.

A tool suite particularly targeted for service engineering is the jABC environment [MS06] with its predecessor METAFrame [SM99]. Both offer behavior-oriented development, incremental formalization, and library-based consistency checking. The tool suite supports a service engineering development process and a modeling theory [MS06]. Synthesis is applied to generate executable prototypes which are abstracted in views, modified and verified until the behavior satisfies the requirements. The service notion we advocate in this paper generalizes the plugin-based service notion of jABC; in fact, as explained in the context of our Service-ADL, our approach to service-orientation brings forward cross-cutting interaction aspects as elements of the logical architecture so that they can be implemented on a wide variety of deployment architectures.

6 Conclusions and Outlook

The high complexity of developing distributed, reactive systems in the embedded domain and other application domains requires effective development methodologies that mitigate these complexities. Complexities are often caused by the high degree of interactions in these systems. Model-based approaches promise to provide a solution to these challenges.

In this paper we have described a model-based approach to developing distributed, embedded systems. Our approach puts the concept of service in the focus of interest. Services are the first class elements that guide the development process from requirements analysis to deployment and execution of the realized system.

Tool support is essential to show the efficacy of a development approach and to increase the efficiency of its application in practical use. We have described how our methodology can be supported by means of tools. Our tool landscape contains tools for requirements analysis, service specification and architecture design, implementation and deployment and verification. The tools are connected in form of a tool chain. We have shown how our tool chain enables a software engineer to specify the basic reactive behavior of our case study example, the Central Locking System (CLS) and generate a distributed system prototype out of the specification model on top of a chosen deployment infrastructure and architecture configuration.

Future work will include a higher degree of integration between the tools and a seamless support of all model elements and operators through all phases of the development cycle.

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 the anonymous reviewers for insightful comments.

References

[AKMP05] Jaswinder Ahluwalia, Ingolf Krüger, Michael Meisinger, and Walter Phillips. Model-Based Run-Time Monitoring of End-to-End Deadlines. In *Proceedings of the Conference on Embedded Systems Software (EMSOFT)*, 2005.

[Aut06] AutoFocus. Website, 2006. `http://autofocus.informatik.tu-muenchen.de/index-e.html`.

[BKM06a] Manfred Broy, Ingolf Heiko Krüger, and Michael Meisinger, editors. *Automotive Software - Connected Services in Mobile Networks. Proceedings of the Automotive Software Workshop San Diego 2004*. Lecture Notes in Computer Science, Volume 4147, Springer, New York, 2006.

[BKM06b] Manfred Broy, Ingolf Heiko Krüger, and Michael Meisinger, editors. *Pre-Proceedings of the Automotive Software Workshop San Diego 2006*. UCSD, 2006. `http://aswsd.ucsd.edu/2006`.

[Bro05] Manfred Broy. The Impact of Models in Software Development. In *Lecture Notes in Computer Science, Volume 2605*, pages 396–406. Springer Verlag, 2005.

[Eva03] Eric Evans. *Domain Driven Design*. Addison-Wesley, 2003.

[GH06] Holger Giese and Stefan Henkler. Architecture-driven platform independent deterministic replay for distributed hard real-time systems. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 28–38, New York, NY, USA, 2006. ACM Press.

[Hol03] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[HSE97] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent Graphical Specification of Distributed Systems. In *FME'97*, volume 1313 of *LNCS*, pages 122–141, 1997.

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

[ITU06] ITU. Sancho Definitions Database. Website, 2006. `http://www.itu.int/sancho`.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[KM96] B.B. Kristensen and D.C.M. May. Activities: Abstractions for Collective Behavior. In *ECOOP'96*, volume 1098 of *LNCS*, pages 472–501. Springer Verlag, 1996.

[KM04] Ingolf Heiko Krüger and Reena 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.

[KMLS05] Ingolf H. Krüger, Reena Mathew, Stefan Leue, and Tarja Systä. Component Synthesis from Service Specifications. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 255–277. Springer Verlag, 2005.

[KMM06] Ingolf Krüger, Reena Mathew, and Michael Meisinger. Efficient Exploration of Service-Oriented Architectures using Aspects. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[KMMP06] Ingolf Krüger, Michael Meisinger, Massimiliano Menarini, and Stephen Pasco. Rapid Systems of Systems Integration - Combining an Architecture-Centric Approach with Enterprise Service Bus Infrastructure. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 51–56. IEEE, 2006.

[KNP04] Ingolf Krüger, Edward C. Nelson, and Venkatesh Prasad. Service-based Software Development for Automotive Applications. In *CONVERGENCE 2004*, 2004.

[Krü00] Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, Technische Universität München, 2000.

[Krü03] Ingolf Heiko Krüger. Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In Mauro Pezzè, editor, *FASE 2003*, volume 2621 of *LNCS*, pages 387–402. Springer Verlag, 2003.

[LMB$^+$01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. *Workshop on Intelligent Signal Processing, Budapest, Hungary, May*, 17, 2001.

[Mat04] Reena Mathew. Systematic Definition, Implementation and Evaluation of Service-Oriented Software. Master's thesis, University of California, San Diego, 2004.

[MS06] Tiziana Margaria and Bernhard Steffen. Service Engineering: Linking Business and IT. *Computer*, 39(10):45–55, 2006.

[NP03] Edward C Nelson and K V Prasaad. Automotive Infotronics: An emerging domain for Service-Based Architecture. In I. H. Krüger, B. Schätz, M. Broy, and H. Hussmann, editors, *SBSE'03 Service-Based Software Engineering, Proceedings of the FM2003 Workshop*, Technical Report TUM-I0315, pages 3–14. Technische Universität München, 2003.

[Obj02] Object Management Group: Real-time CORBA specification, 2002. `http://www.omg.org/technology/documents/index.htm`.

[OMG03] OMG (Object Management Group). Model Driven Architecture (MDA). MDA Guide 1.0.1, omg/03-06-01, 2003. `http://www.omg.org/mda`.

[OMG05] OMG (Object Management Group). UML, Version 2.0. OMG Specification formal/05-07-04 (superstructure) and formal/05-07-05 (infrastructure), 2005.

[Pae97] Barbara Paech. A Framework for Interaction Description with Roles. Technical Report TUM-I9731, Technische Universität München, 1997.

[Par02] Parlay 3.0, 2002. `http://www.parlay.org/en/specifications/`.

[Rus06] Yenny Rusli. Methodological Translation of Service-Oriented to Component-Oriented Specification. Master's thesis, University of California, San Diego, 2006.

[SB98] Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of ECOOP 1998*, volume 1445 of *LNCS*, pages 550–570. Springer Verlag, 1998.

[SM99] Bernhard Steffen and Tiziana Margaria. METAFrame in Practice: Design of Intelligent Network Services. *LNCS*, 1710:390–415, 1999.

[STK02] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[Sys06] Systems Modeling Language (SysML), 2006. `http://www.sysml.org/`.

[Szy02] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[TRH+04] David Trowbridge, Ulrich Roxburgh, Gregor Hohpe, Dragos Manolescu, and E.G. Nadhan. *Integration Patterns. Patterns & Practices*. Microsoft Press, 2004.

[Zav01] Pamela Zave. Feature-Oriented Description, Formal Methods, and DFC. In *Proceedings of the FIREworks Workshop on Language Constructs for Describing Features*, pages 11–26. Springer-Verlag, 2001.