Towards a Process and Tool-Chain for Service-Oriented Automotive Software Engineering

Ingolf H. Krüger, Diwaker Gupta, Reena Mathew, Praveen Moorthy,
Walter Phillips, Sabine Rittmann, Jaswinder Ahluwalia
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093-0114, USA
{ikrueger,dgupta,rmathew,pmoorthy,whphilli,srittmann,jas}@ucsd.edu

Abstract

The complexity of automotive software systems continues to increase at a dramatic pace. Traditionally, the interactions between the various software components of a vehicle are addressed only at the later stages in the overall development process. We advocate a fresh approach, where interaction patterns become the defining elements of automotive software services. This shifts the development focus from individual components to their interaction in the early stages of the development process; potentially reducing development and integration costs for both manufacturers and suppliers. We present a formal service notion based on interaction patterns and introduce a systematic, service-oriented development process, substantiated by means of a corresponding tool chain. We illustrate our definitions and results by modeling elements of a Central Locking System, an example from the automotive domain.

1. Introduction

Automotive software systems are becoming increasingly complex. In some cars up to 1400 software-enabled functions are distributed over up to 80 Electronic Control Units (ECUs), communicating over five different network infrastructures within and beyond vehicle boundaries [6,21]. More and more, the complexity ensuing from the collaboration among these distributed and interconnected pieces of software functionality becomes *the* limiting factor in automotive software design. Traditional development processes for automotive software focus on the construction of individual ECUs, starting from more or less detailed specifications provided by the manufacturer; integrating them and the myriad of interacting software functions they implement into a coherent, functional, and safe overall product. This incurs substantial cost and effort for the manufacturer in later stages of the vehicle development process.

We conjecture that turning this picture around, and driving the automotive software development process by the *features* [10] or software *services* instead of by the ECUs that implement them, goes a long way towards more cost-effective and less error-prone automotive software development.

Software services emerge from the interplay of several components collaborating to complete a desired task. Current software development approaches and modeling notations focus mainly on individual components instead of on the interaction patterns defining services as cross-cutting system properties.

We present an approach to designing automotive software starting from service specifications. To that end, we introduce an interaction-based service notion together with a development process and corresponding tool chain for service-oriented software development.

Our approach is driven and validated by a collaboration with Ford Motor Company. In this project, the goal is to derive prototypic executable models – in the form of RTCORBA[25] components – for automotive services that can be handed off as “reference models” (*not* as target code) to suppliers. The suppliers can then validate their proposed solution against the executable model; this substantially streamlines the subsequent integration phase for both manufacturers and suppliers.

1.1 Services, Roles, Components and Architectures

Services play an increasingly important role as a modeling and implementation concept across application domains. Even in the automotive domain – representative for the area of large-volume embedded systems with high dependability requirements – the notion of service is catching on as a means for addressing the ever increasing distribution and ensuing complexity of automotive software[1,21].

Despite the importance and prevalence of services, however, no systematic, methodological approach to service-oriented software development exists to date. In addition, none of the prominent modeling notations currently available address the service concept as a first-class modeling entity. Informal definitions for the term service abound in the literature (cf., for instance, [23,12,29,24,13]); these definitions, however, typically capture only syntactic lists of operations upon which a client can call instead of the interaction patterns behind them. This is inadequate as a starting point for systematic service-oriented development.

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

Defining services by the interactions ensuing from invoking them goes well beyond the predominantly syntactic service notions cited above: it provides a handle at meaningful concepts for service composition, refinement, and validation, and introduces them as first-class *modeling* elements as opposed to being merely first-class *implementation* elements.

Clearly, services, once captured, can be implemented on top of a variety of software architectures. *Components* typically participate in multiple services. For instance, a particular component may act as a server in one service, while it acts as a client in another. To make the different configurations in which a component can participate in service executions explicit, we introduce the concept of a *role*. At any point in time during execution of the system under consideration a component can play certain roles, such as the client or server role in a client/server interaction. Components can change their roles over time; this distinguishes roles clearly, for instance, from the class concept in object-orientation. Semantically, roles map to predicates over the state space of the system under consideration.

While the focus of this paper is on presenting the methodological aspects of a service-oriented development process, and the ramifications for a corresponding tool chain, we refer the reader interested in a formalization of our service notion using the mathematical model of streams to [16,17,18].

1.2 Interaction Modeling: A Central Challenge in the Automotive Domain

According to recent data, up to 40% of a modern car's cost is due to software; 50-70% of the development cost of ECUs is related to software. Primarily, this complexity stems from the myriad of interactions among the various functions realized in software; we will illustrate this using the central locking system found in most modern cars as an example (cf. Sec. 2).

To capture the interaction patterns defining services we use an extended version of Message Sequence Charts (MSC) [9,15]. MSCs have proven useful as a graphical representation of key interaction protocols, originally in the telecommunications domain. They also form the basis for interaction models in the most recent rendition of the UML [30]. In our extended MSC notation, each MSC consists of a set of axes, each labeled with the name of a role (instead of with a component name). An axis represents a certain segment of the behavior displayed by the component implementing the corresponding role. Arrows in MSCs denote communication. An arrow starts at the axis of the sender; the axis at which the head of the arrow ends designates the recipient. Intuitively, the order in which the arrows occur (from top to bottom) within an MSC defines possible sequences of interactions among the depicted roles.

1.3 Service-Oriented Development

For service-oriented system development in the automotive domain, we suggest to follow a systematic development process as outlined in Figure 1.

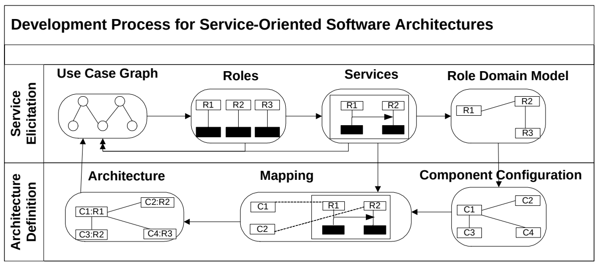

Figure 1. Service-Oriented Software Development

This iterative process mainly involves two phases: (1) define the set of services of interest – we call this set the *service repository*; (2) map the services to component configurations to define deployments of the architecture. Phase (1) starts by identifying the relevant use cases and their relationships in the form of a use case graph. From these use cases the roles and their interactions are derived as defining elements of services. This gives rise to a *domain model* for the roles involved. 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.

1.4 Contributions and Outline

In the remainder of this text we explain the various elements of our service-oriented software development approach together with the process outlined above in more detail. To illustrate concepts and notations we introduce a realistic running example in Section 2. In Section 3 we apply the mentioned development process to the running example, and show how to yield multiple architectural configurations for the same set of services. In Section 4 we describe the tool chain we envision for service-oriented automotive software engineering. Section 5 contains a discussion of related work. Our conclusions appear in Section 6.

2. Example – Central Locking System

To illustrate the central challenges in specifying services, as well as the applicability of the service concept introduced above, we model part of the Central Locking System (CLS) and its periphery as found in typical modern cars. The starting point for our modeling effort is the description of a "real" CLS in [21]; for reasons of brevity we use a simplified model of this CLS here.

In today's cars the seemingly innocent CLS service is interconnected with many other features in the vehicle. In fact, a typical vehicle today contains distinct networks for each of the following: body/chassis control, powertrain control, safety control, and multimedia. All of those are linked in the functionality provided by CLS; it interacts with the locks (in the doors, trunk, sunroof etc.), a remote key, crash notification, the security system, light control (interior and exterior), the tuner, seats and mirrors. A typical scenario for using the features of a correspondingly equipped car is to unlock the car remotely by pressing the key-button: upon receipt of this signal, the CLS un-arms the security system, operates the appropriate locks and switches on the interior lights. The driver's preferences are accessed, based on the key used, to set the mirror, seat and tuner presets accordingly. Flashing the exterior lights signals the opening of the car from the outside. Upon drive-away and beyond a certain speed, the CLS automatically locks all doors.

In particular, we will study the following services for this example: unlocking, transfer key ID, set tuner presets, and crash detection. The unlocking service involves interactions among the key fob, control, lock management and lighting system. To fetch the driver presets upon entry into the vehicle, the locking system control associates the driver identifier stored on the key fob with the corresponding data record in the database. When switched on, the tuner fetches the current presets from the database. When a crash is detected, the central controller initiates unlocking of all doors.

Besides illustrating the system structure, this example also shows how the mentioned services overlap; in particular, a central controller is involved in executing all but the tuner preset services. This is a key characteristic of service-oriented development: services are cross-cutting elements of the system under consideration.

3. Service-Oriented Development: Step-by-Step

In this section we explore the service-oriented development process in more detail. In the following paragraphs, we elaborate on the steps required to define an architecture for CLS, based on the concepts of use cases, roles, services and components.

3.1 Use Cases

Our first step is to determine the relevant use cases of the system under consideration. These use cases can be related to each other by virtue of inclusion and extends relationships [30]; we can express these relationships in terms of a use case graph – this graph will also help determine the relationships between services elicited from the use cases. We consider the following four use cases for the CLS: *unlocking, transfer_ key_ID, set tuner presets, and handle_crash*.

Clearly, these use cases have some overlap: both the unlocking of the car, and the transfer of a key ID are triggered by the user pressing a key on the keyfob. These two use cases can, however, also be considered separate, because there are keys that can unlock the car (mechanically, for instance) but do *not* transmit key identifiers. There are also more subtle dependencies between these use cases: the *set tuner presets* use case has a data dependency with respect to the *transfer key ID* use case; the transferred ID will influence the settings of the tuner. The *handle_crash* use case crosscuts all others: whenever a crash signal occurs the CLS has to unlock all doors.

All of these dependencies can be captured by an appropriately labeled use case graph; dependencies between use cases can be translated into additional services (see below).

3.2 Roles

In order to decouple the notion of service from concrete component configurations (architectures), we introduce the concept of a *role*. At any point in time during execution of the system under consideration a component can play certain roles, such as the client or server role in a client/server interaction.

Within the CLS, for instance, we can identify the following roles: a control unit (Control); a lock manager (LM) responsible for operating the door, trunk and moon-roof locks; a lighting system (LS) responsible for operating the interior and exterior lights, a crash detection sensor (CS); a key fob (KF) responsible for remote and mechanical entry, and storing of the driver identification; a database (DB) storing the presets for each registered driver – the key to the database record is the driver identification stored in the key fob; a tuner (Tuner) for the entertainment system; and a user interface (UI) for operating the tuner. These roles will likely map to a variety of different component configurations depending on the concrete make and model under consideration.

Semantically, roles map to predicates over the state space of the system under consideration.

3.3 Services

We define services as the interaction patterns required to establish a specific task. This identifies services as *partial behaviors* of the system under consideration.

Services associate roles with interaction patterns. We use MSCs to represent how the roles interact to establish the service under consideration. An MSC can also contain information about the states the roles are in during the course of the interaction.

The collection of all services elicited in this way can be considered as a *service repository* for the system. Once we have defined the service repository, the relationships between the different roles of the system are clear. We express the *structural* part of these relationships in the form of a *role domain model*. This domain model allows us to identify the dependencies between the different roles of the system and is used later on to identify and define corresponding component configurations.

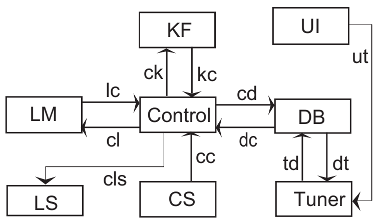

Figure 2. Role Domain Model for CLS

For the CLS we define the role domain model shown in Figure 2; here, boxes and arrows denote roles and directed communication channels connecting the roles, respectively. For instance, roles KF and Control are connected by means of channels *ck* (from Control to KF) and *kc* (from KF to Control).

With this role domain model in place we can start capturing the interaction patterns defining the services of the CLS. Figure 3 shows the interactions of the unlocking service. Here, Control relays an unlck message received from the keyfob KF to the lock management (LM) role. Upon receipt of an ok message from LM, Control sends the door_unlckd_sig message to the lighting system. The labeled hexagons in the MSC indicate local (control) states of role Control: it starts out in the LCKD state; after participating in the depicted interactions, Control switches into the UNLD state.

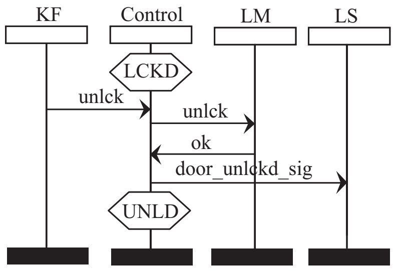

Figure 3. MSC for "unlocking"

Upon receipt of an unlck message the Control role sends a getID message to the keyfob; KF sends the ID to Control, which relays the ID to the DB (cf. Figure 4). Again, Control switches from state LCKD UNLD in the course of executing the service.

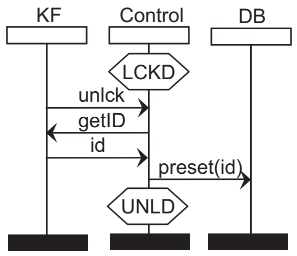

Figure 4. MSC for "transfer driver ID"

The preceding two services are *overlapping* in the sense that both share references to the unlck message and states LCKD/UNLD. This is characteristic of service specifications as each individual service gives only a partial view on the components implementing it. Therefore, to obtain the complete behavior for one of the roles, Control, say, we have to compose all the services it is involved in. Because neither the MSC standard [9], nor UML2 [30] provides composition operators for overlapping interaction patterns, we have introduced a corresponding “join” operator [18] (symbol: ⊗). We write *unlocking* ⊗ *transfer_driver_ID* to indicate the interaction pattern emerging from synchronizing identically labeled (or *unifiable*) messages (and states) in the operand MSCs *unlocking* and *transfer_driver_ID*. All other messages (and states) in these operands are interleaved in the resulting interaction pattern – observing, of course, the respective message orderings in the operands. In the example, the “unlck” message from KF to Control is unifiable and thus identified as a single message in the composition. All other messages are mutually independent between *unlocking* and *transfer_driver_ID*.

The start_tuner service is triggered by an “on” message from the user interface UI to the Tuner role. The Tuner, in turn, requests the presets (stations, volume, etc.) from the database DB (cf. Figure 5).

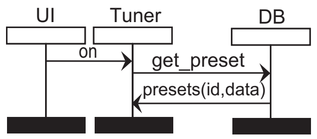

Figure 5. MSC for “start tuner”

From the use case graph we have elicited earlier in the development process we know that there is a data dependency between the services *start*_tuner and *transfer_driver*_ID. This data dependency can be captured semantically by means of a generalized “join” that unifies on message labels (including origins and destinations) *and* message parameters.

The final service we model for CLS is the *handle_crash* service. Its actual interaction pattern is simple (cf. Figure 6).

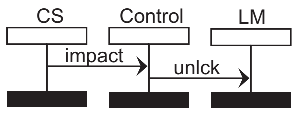

Figure 6. MSC for “handle crash”

It is the relationship between *handle_crash* and the other services we have elicited so far that is noteworthy: *handle_crash* is cross-cutting in the sense that the “impact” message can arrive at any point during the execution of any other service. We refer the reader to [18, 15] for the definition of a “preemption” operator that enables us to identify *handle_crash* as a preemption handler triggered by the “impact” message from the crash sensor CS. Again, the cross-cutting nature of such preemption specifications is inadequately addressed in notations such as the MSC standard or UML2.

Other services we could capture analogously are, for instance, “locking”, “partial locking/unlocking”, “valet unlocking”. Their composition using operators for sequencing, alternatives, parallel composition, and join (as formally defined in [18, 15]) would yield the overall CLS service. For reasons of brevity we restrict our attention to the services explicitly modeled above.

3.4 Components and Architectures

Following service elicitation, we can define the architecture for the system under consideration. Components are the architectural entities implementing roles. Each component may play multiple roles. The mapping from roles to components determines the services a component participates in and, thus, also the component’s interaction behavior. There can be many possible component configurations for a system that implements the elicited services. Therefore, we need to first define the component configuration for which we want to define an architecture. We represent component configurations using a *deployment domain model*. The components defined in this domain model receive their behavior definitions by mapping the services onto the chosen component configuration.

We obtain a trivial deployment domain model from the role domain model by identifying components and roles; then, each component implements precisely one role. In this state of affairs, the role domain model and the deployment domain model coincide.

Another extreme case is to map all roles to a single component; this again is a trivial affair, because we simply need to treat the role domain model as a specification for the “internals” (the substructure) of one encompassing component.

The most interesting and methodologically challenging case arises when we map multiple roles onto the same deployment component. All other cases (such as mapping a single role onto multiple components) can be dealt with by refactoring/refining the role domain model first, and then establishing the mapping to the deployment domain model.

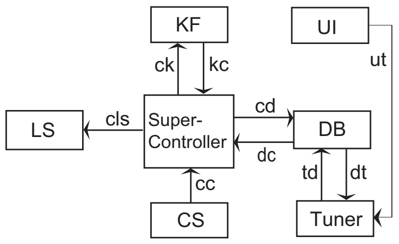

Figure 7. Deployment Domain Model for CLS

We illustrate this case in our example by introducing a “Supercontroller” component, playing both the Control and LM (lock manager) roles. Figure 7 shows the corresponding deployment domain model. If we work with strictly hierarchical component models such as the ones of UML2, UML-RT, or AutoFocus (cf. Section 4.2), one way to establish the mapping of multiple roles onto a single component is to take the role domain model as a staring point, and to replace the roles in question by a single component having the same input and output channels as the replaced roles taken together. Then, the entire network of replaced roles with their supporting channels becomes the hierarchical “child” of the freshly introduced component. This process can be repeated, recursing into all hierarchically decomposed composites, until all role labels have been turned into component labels.

In general, there are many possible component configurations for a system that implements the elicited services. Therefore, we need to first define the component configuration for which we need to define an architecture. Such configurations can be based on various architectural styles [26] such as pipes and filters, layered architecture, or client-server.

Layered architectures are particularly appealing as deployment configurations for services. Based on the dependencies elicited for the use case graph we can determine which services call upon which others. The former make good candidates for being on a higher layer of abstraction than the latter. This approach works well if the dependency graph has no cycles; otherwise these cycles have to be broken (by introducing, for instance, service *proxies*) before a strictly layered architecture can be derived.

Once a component configuration has been defined, the architecture can be determined by selecting the services from the service repository that need to be supported by the architecture and mapping each service to the component configuration and defining the component types. This happens according to the following two steps: (1) Select a service to be supported by the system, (2) Map the roles of the service to the components supporting that service. These steps need to be repeated for all services to be supported by the system. This mapping step allows us to select and implement only those services required for the architecture under consideration; it also allows us to try out various configurations.

Because our service notion is interaction-based, and we use MSCs to capture the relevant interaction patterns, we can map these patterns onto individual roles, and from there onto components (semi)automatically, using algorithms for the derivation of state machines from interaction specifications [14,15]. We explain this in more detail in Section 4.1.

3.5 Service Refinement and Refactoring

In practice, the initial architecture determined using any process, including the one outlined above, will have to be refined as more information about the services and target component configurations (such as the concrete set of ECUs and the supporting network topology available in the vehicle) is revealed. This is supported by the iterative nature of our process, allowing the architect to refine the architecture by collecting increasingly detailed information about the use cases, roles, and services of the system under consideration: the service elicitation step is decoupled from the architecture mapping. Consequently, both the service repository and the target architecture can be modified independently of one another before the next architectural mapping step is performed.

4. Tool Chain

In this section we describe the design and implementation of a prototypic toolchain we have developed for supporting the service-oriented development process introduced above. The purpose of this tool chain is to provide executable models for the services in vehicle software. These executable models are intended as reference models for suppliers to validate their implementations against – not as deployable target code. The tool chain consists of three main ingredients: the modeling tool for roles, interaction patterns and service specifications (M2Code), the modeling, simulation, verification and test tool for component configurations (AutoFocus) and a code generator, transforming the output of either M2Code or AutoFocus into executables for the RTCORBA middleware platform.

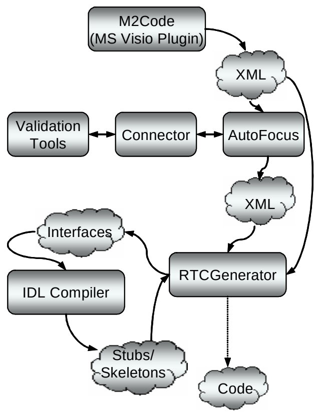

Figure 8. Toolchain Supporting Service-Oriented Development

The general idea is to model roles and services in M2Code based on its capabilities for interaction specification using our full, extended MSC notation. For the resulting role domain model M2Code generates automaton specifications and deployment domain models that can be fed into AutoFocus for simulation, validation and verification; AutoFocus provides connectors to verification tools such as model checkers and theorem provers. The code generator takes deployment domain models and corresponding state models as input and produces code (interfaces, stubs, skeletons, and component implementations) for executable RTCORBA components.

Figure 8 gives a pictorial representation of this toolchain. In the following paragraphs we will describe each of these ingredients in more detail.

4.1 M2Code

M2Code is our tool for modeling roles and their interactions, which are the defining elements for services in our approach. Thus, M2Code covers, in particular, the first phase of the service-oriented development process introduced in Section 1.3. A centerpiece of M2Code is its capability for (semi)automatically deriving state machines from interaction patterns given as MSCs, based on the algorithms described in [14,15]. In particular, it can deal with overlapping, alternatives, parallel composition, loops, and preemption – to name only some important features of dealing with interaction patterns defining services.

The output of M2Code is a role domain model together with one automaton for each role defined in this domain model; these automata implement the respective role's contribution to all the services it is associated with.

M2Code is built as a plugin for Microsoft Visio, and visualizes both the MSCs entered by the developer and, using automated layout algorithms provided in "graphviz" [7], also the generated state machines for validation purposes.

Finally, M2Code supports exporting the modeling elements (both the automata and the system structure, i.e. the role domain model) as an XML document in the format expected by AutoFocus. If no verification/validation of the resulting service model is required, AutoFocus can be bypassed, and the XML file can be directly used as input for the RTCORBA code generator.

4.2 AutoFocus

We have selected AutoFocus as an element in our toolchain, because it is freely available as a research platform and provides strong modeling, verification and validation capabilities. AutoFocus was developed at Technische Universität München [3]. Its notational elements cover system structure, data types, event traces, and state machines for embedded, reactive systems.

For modeling system structure, AutoFocus employs system structure diagrams displaying components and their ports (interfaces), which are interconnected via channels. Components can be either terminal or hierarchically decomposed. For each port and channel, data types may be specified – both default and user defined types are possible. For the behavioral view of the system (the dynamic view), AutoFocus uses state transition diagrams (STDs, automata) and extended event traces (EETs) . EETs are similar in spirit to the MSCs used in this text, but have a much more restricted expressive power; they serve mostly as a graphical representation of simulation runs.

AutoFocus itself has a built-in code generator targeting Java, which allows step-by-step execution and simulation of the model under consideration. Furthermore, AutoFocus provides a number of automated and semi-automated testing approaches, which facilitate conformance tests between expected and actual interaction sequences in executions of the system under development.

AutoFocus also supports a coupling with external verification and validation tools using tool "connectors". For our prototype implementation, we are using the Symbolic Model Verifier (SMV [28]) as the external validation tool. By means of this model checker we can, for instance, formally verify the correctness of our mapping of services onto the deployment domain model. These verification and validation capabilities within the AutoFocus CASE tool framework are our motivation for taking AutoFocus models as the input for our code generation approach.

4.4 RTCORBA Code Generator

The final step towards an executable specification is handled by a code generator, which takes abstract, validated models of distributed, reactive systems as input, and produces executables for the RT CORBA middleware implementing the properties checked for the abstract models. This closes a gap in the development process for reliable distributed and reactive systems, by eliminating the manual transition from captured requirements to implementation on top of RT CORBA. The design of the code generator (an extension of the one discussed in [20]) can easily be adapted to different input languages and target middlewares.

Code generation proceeds in three steps: First, the code generator takes an XML file (generated by AutoFocus or directly out of M2Code) as input, populates a symbol table, and produces an IDL file containing CORBA interface specifications for all terminal components of the AutoFocus model. Next, the IDL file is fed into the IDL compiler; this produces the relevant stubs, skeletons, and "empty" CORBA component implementation prototypes. Finally, the code generator takes the output of the IDL compiler, and populates the files with the code obtained from translating the behavioral aspects contained in the AutoFocus model.

As our target RT middleware we have selected TAO, developed at the University of Washington, University of California at Irvine, and Vanderbilt University. As a consequence, our target programming language is C++.

5. Related Work

The complexity and challenges offered by the problems in the automotive domain have been pointed out earlier (see, for instance, [27]) and the automotive industry has been undergoing a silent software revolution [6,8] – however, to the best of our knowledge, ours is the first attempt at a systematic application of the service notion to software development in general, with a focus on the automotive domain. The notion of services as first-class elements from an *implementation* perspective has seen a lot of research in the emerging context of web services [2, 31]. There has been some earlier work on web service description and composition as well [22]. Using a precise, interaction-based service notion from the onset of the development process and seamlessly across development phases as we advocate here is novel; this is also a major difference to other software development processes, such as the one described in [11].

Although there has been little research on developing an integrated service oriented development process, there has been some work focused on each of the individual stages. Wren [19], for instance, is a tool similar in spirit to our use of AutoFocus, in that it aids in component based software development through composition. It provides several interesting features including component discovery, but does not incorporate any kind of behavioral semantics for its component notion. Attempts have also been made to extend UML for the automotive domain [5] – however, as we have argued, the UML does not treat services as first-class modeling citizens and thus misses many of the important aspects of service modeling, including overlapping services. Baresi et al. discuss modeling and validation of service-oriented architectures ([4]). These could serve as deployment domain models for the services we model in our approach.

There has been some work in composing systems from components and validating such systems from their components without having to validate the system as a whole [32]. However, most of these approaches are "bottom up" (going from components to the system), while ours is a "top down" approach (going from services to the components implementing them), which we have argued is a better suited to address the cross-cutting nature of many important system properties. In this sense, our work can serve as a "front-end" to approaches such as [32].

6. Conclusion and Outlook

The ever increasing complexity of automotive software systems has become a potential road-block for software-based innovations and their transition into production vehicles. Traditional software development approaches are limited in their utility for the automotive domain, because they address the key element of interaction among the myriad of software enabled vehicular functions only in the late stages of the overall development process. Consequently, the integration phase where all components delivered by suppliers are assembled into a whole is costly, time-consuming, error-prone and difficult.

In this text, we have introduced an approach to service-oriented software engineering for automotive systems. Starting from a precise, interaction-based notion of service we have outlined a methodologically founded development process for service-oriented systems. We have substantiated this process by means of a tool chain covering all phases from capturing the interaction patterns defining services to designing deplyment models to implementing these models on top of RTCORBA. We have illustrated our approach using a simplified, yet realistic model of a central locking system.

Future work includes, among others, a thorough assessment of multiple deployment domain models regarding their utility as service-oriented software architectures, an extension of the service-modeling tool chain to include "wizards" for refinement and refactoring of services and domain models, and integration of Quality-of-Service specifications into the service model.

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. We are grateful to K.V. Prasad and Ed Nelson (Ford Motor Company), as well as to the anonymous reviewers for insightful comments on the approach presented here. The authors are also grateful to Manfred Broy, Bernd Finkbeiner and Jon Whittle for helpful discussions on the topics presented in this text. We thank Oliver Müller for developing an initial prototype of the code generator.

References

[1] *AMI-C*: http://www.ami-c.org/publicspecrelease.asp

[2] M. Aoyama, S. Weerawarana, H. Maruyama, C. Szyperski, K. Sullivan, and D. Lea, *Web services engineering: promises and challenges*, in: Proceedings of the 24th International Conference on Software Engineering (2002), ACM Press, pp. 647—648, 2002.

[3] *AutoFocus*, available at: http://autofocus.informatik.tu-muenchen.de/index-e.html

[4] L. Baresi, R. Heckel, S. Thöne, and D. Varrò, *Modeling and validation of service-oriented architectures: application vs. style*, in: Proceedings of the 9th European software engineering conference, ACM Press, pp. 68—77, 2003.

[5] M. von der Beeck, P. Braun, M. Rappl, and C. Schroeder, *Automotive UML: a (meta) model-based approach for systems development*, in: UML for real: design of embedded real-time systems, pp. 271—299, Kluwer Academic Publishers, 2003.

[6] M. Broy, *Automotive software engineering*, in: Proceedings of the 25th International Conference on Software Engineering (2003), IEEE Computer Society, pp. 719—720, 2003.

[7] *Graphviz*: http://www.research.att.com/sw/tools/graphviz

[8] K. Grimm, *Software technology in an automotive company: major challenges*, in: Proceedings of the 25th international conference on Software Engineering (2003), IEEE Computer Society, pp. 498—503, 2003.

[9] ITU-T, *ITU-T Recommendation Z.120 – Message Sequence Chart (MSC96)*, ITU-T, Geneva, 1996.

[10] M. Jackson, P. Zave: *Distributed Feature Composition: A Virtual Architecture for Telecommunications Services*. IEEE Trans. Software Eng. 24(10): 831-847, 1998.

[11] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 1999.

[12] *Jini*: http://wwws.sun.com/software/jini/specs/

[13] *JXTA*: http://jxme.jxta.org/

[14] I. Krüger, R. Grosu, P. Scholz, and M. Broy: *From MSCs to Statecharts*, in: Franz. J. Rammig (ed.): Distributed and Parallel Embedded Systems, Kluwer Academic Publishers, 1999.

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

[16] I. Krüger, *Specifying Services with UML and UML-RT*, Electronic Notes in Theoretical Computer Science, Vol. 65(7), 2002.

[17] I. Krüger, *Towards Precise Service Specification with UML and UML-RT*, in: Critical Systems Development with UML (CSDUML), Workshop at «UML» 2002, 2002.

[18] I. Krüger, *Capturing Overlapping, Triggered, and Preemptive Collaboratons*, in: Mauro Pezze (ed.): Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Lecture Notes in Computer Science 2621, Springer 2003.

[19] C. Lüer, D. S. Rosenblum, *Wren — an environment for component based development*, in: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering (2001), ACM Press, pp. 207—217, 2001.

[20] O. Müller: *Generating RT-CORBA Components from Service Specifications*, Abschlussarbeit, Fakultät für Informatik, Technische Universität München, 2004.

[21] E. C. Nelson and K. V. Prasad, *Automotive Infotronics: An emerging domain for Service-Based Architecture*, in: I. H. Krüger, B. Schätz, M. Broy, and H. Hussmann (eds.), SBSE'03 Service-Based Software Engineering, Proceedings of the FM2003 Workshop, Technical Report TUM-I0315, Technische Universität München, 2003.

[22] C. Pahl and M. Casey, *Ontology support for web service processes*, in: Proceedings of the 9th European software engineering, ACM Press, pp. 208—216, 2003.

[23] *Parlay 3.0*: http://www.parlay.org/specs/index.asp

[24] D. S. Platt and K. Ballinger, *Introducing Microsoft .NET*, Microsoft Press, 2001.

[25] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar, *A high-performance end system architecture for real-time CORBA*, in: IEEE Communications Magazine, 14(2), Feb. 1997.

[26] M. Shaw and D. Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[27] C. Sinz, A. Kaiser, and W. Küchlin, *Formal methods for the validation of automotive product configuration data*, Artif. Intell. Eng. Des. Anal. Manuf. 17, 1 (2003), 75—97, 2003.

[28] *SMV*: http://www2.cs.cmu.edu/~modelcheck/smv.html

[29] J. Snell, D. Tidwell, and P. Kulchencko, *Programming Web Services with SOAP*, O'Reilly, 2002.

[30] *UML 2.0*: http://www..omg.org/uml

[31] S. Weerawarana, *Web services and software engineering: challenges and opportunities*, in: Proceedings of the 23rd International conference on Software Engineering (2001), IEEE Computer Society, p. 683, 2001.

[32] F. Xie and J. C. Browne, *Verifed systems by composition from verified components*, in: Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering (2003), ACM Press, pp. 277—286, 2003.