Tool Supported Development of Service-Based Systems*

Martin Deubler, Johannes Grünbauer,
Gerhard Popp & Guido Wimmel
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
{deubler, gruenbau, popp, wimmel}@in.tum.de

Christian Salzmann
BMW Car IT GmbH
Petuelring 116
80809 München, Germany
christian.salzmann@bmw-carit.de

Abstract

Service-based systems engineering is a recent paradigm that has proven useful for the development of multifunctional systems, whose functions may be used in different contexts and have strong interrelations and dependencies between each other. Integrated into a service-oriented development process, we present an approach for tool supported design of services and execution scenarios describing their interaction, using the tool AUTOFOCUS. *It includes the application of simulation, verification of typical requirements for service-based systems using model checking, and code generation. We report on our experience with this approach by means of a case study from the automotive domain, a fairly new field of application for service-based systems engineering.*

1. Introduction

In the automotive domain the types of applications are not anymore limited to classical embedded systems, such as airbag control software, but cover a broad range from mission critical embedded systems in the X-by-wire field to infotainment and personalization in the MMI (Man Machine Interface) area. The MMI is a central software system in the car that manages the access of the user (driver) to the functions of the car such as infotainment, phone and Internet access, climate control and navigation system. In the 7 series of BMW, for example, the MMI system consists of a set of about 270 functions that can be triggered by the user and are distributed over about 40 electronic control units.

A fundamental difference of such a system in comparison to classical applications is in the interaction and dependencies of the functions. One function, such as the volume control of the amplifier, can be triggered not via one single user interface, but by a set of other functions, e.g. when the navigation system informs the driver with a voice output the software lowers the volume of the audio system for the time of the voice information.

This type of system, which is characterized by multiple function interactions, strong interrelations and dependencies between functions, is called a *multifunctional system* [7]. The functional aspect is more important than the whole application, as one function may be used in different contexts. Hence, we have to deal not only with the functions themselves but also with their encapsulation, their dependencies and their combination with other functions. The unit of function with its encapsulation, dependencies and combination is called a *service*.

A service is not only a functional entity representing a certain behavior. It is also *the* abstract modeling unit, which is central for our development process. In a nutshell a service comprises:

- *a syntactic interface*—types of messages that could be exchanged with the service,
- *a behavior*—the service interface, a partial function relating valid sequences of input messages to sets of sequences of output messages,
- *a set of properties*—self-description, quality of service attributes, and
- dedicated *relationships* to other services.

Relationships to other services are illustrated in Fig.1. For instance, service F_3 is related to the services F_1, F_2 and F_4 by r_2, r_3 and r_5, respectively.

An application area of our approach is system integration. The state of the art in system integration is to specify the logical architecture of a system independently from its technical architecture (the deployment) by specifying functional networks and their dependency. However, the depen-

* This work was supported by the Bavarian high-tech funding program (High-Tech Offensive), within the project MEWADIS [1].

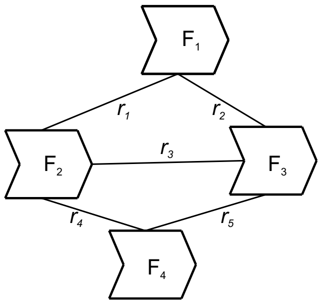

Figure 1. Service relationships

dency is exclusively based on communicational dependencies: one function f depends of a function g if f gets input signals that are output of g. We see now that this notion of dependency is not sufficient: does a function f also depend on g if the output signals of g are only looped through function f? It seems clear that functional dependency, which is the basis for many problems in the system integration, should be based on a more formal definition that includes behavior to express this dependency.

In this paper, we explain our approach by referring to an industrial case study and follow the service-oriented development method introduced in [7]. For the development of multi-functional software systems that kind of development process suits well: the system functionality is structured [5]—the behavior of individual components is a secondary objective. Right from the start the development process focuses on the relations between system functions and the interactions of them. That way, unwanted feature interactions can be prevented at an early stage. They often arise with conventional development processes, since interactions are considered only at a late stage. In our opinion the service-oriented approach is also more intuitive to get from requirements to the system design. It is easier than designing a component architecture right from the requirements and specifying the behavior of individual components. In particular, requirement specifications usually describe the expected system behavior (use case views, "how is the system used?" [5]) and are expressed by activity flows and expected results of activities. The case study originates from the automotive domain and has been carried out in cooperation with the automobile manufacturer BMW.

Section 2 briefly explains the case study's environment and requirements. Section 3 illustrates the application of the service-oriented development process. In particular, we make use of an extended notation and modeling technique. In Section 4 we describe the tool support for the process. Section 5 gives the conclusion, a discussion and an outlook to future work.

2. The case study

The central concept of our industrial case study is an adaptive display control within an MMI application. The purpose of the display control is to handle display requests from other parts of the system. The actual information displayed must be "context sensitive", that is, it must be adapted to the car's current state. As an example, we consider speed: at higher speeds, less information should be displayed on the screens for safety reasons. Furthermore, if there are important events (with a higher priority), like an incoming telephone call, the display has to show the corresponding information immediately. As an example, we consider a display control connected to two different displays, in interaction with software for a CD changer and a telephone.

The first display under consideration is the graphical Control Display (CDSP). It is used for displaying multimedia information to the driver and the fellow passenger. The second display is the Multi-Information Display (MID). It is located below the tachometers and informs the driver about mileage, alerts, etc. The MID is a monochrome display which can show either two lines of plain text or a symbol.

In our case study we specify an adaptive display service which shows information on both displays dependent on the priority of the information and the current speed of the car. When a CD is played in the CD Changer at a low speed (less or equal than 100 km/h), the track number and the full title including the interpreter and the song title will be shown on both displays. When the driver accelerates the car to a speed greater than 100 km/h, for safety reasons only the track number is shown in the MID. When a phone call comes in, both displays show the information about the caller (in this case, independently from the speed, but we may hazard the safety consequences because we're just dealing with a model), and the driver may answer the telephone. After hanging up, the CD information is shown again.

3. Description of the development process

In this section, we present a development process for service-based systems. The basis for our development process is a phase oriented development process like the waterfall model [21] or object oriented development methods such as the Unified Software Development Process [13] or the Catalysis Approach [8], just to name a few of them. We integrate services as a central concept and guideline of the development. The presented process is model-based, i.e. the development is driven by explicit models of both the development artifacts (product model) and of the process itself. The models of the development artifacts have a predefined struc-

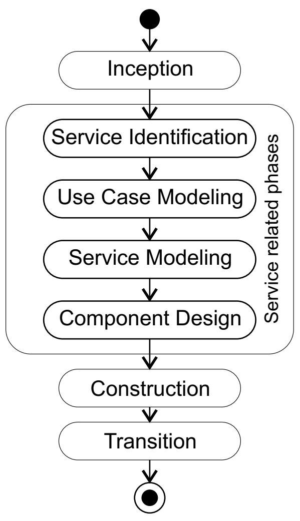

Figure 2. Phases of the service oriented development process

ture (which does not rule out textual parts), and the process model describes how development proceeds in the different phases, in terms of the product model. The process is incremental in that it can be repeatedly applied to add new functionality in small steps, which considerably reduces risk.

Fig.2 shows the phases of the service oriented process. We give a short overview about them and explain the service related ones in more detail in the following sections.

We assume that we have an *Inception Phase*, in which the project is born and both a project mission and the requirements are elaborated. This is the starting point of our process. The concept of service does not yet appear and thus the Inception Phase is not affected by our approach. The results of this phase are documented in a project mission document and in a requirements specification list.

After the Inception Phase, the sequences of actions at a high abstraction level are modeled in the *Service Identification Phase*. In this phase a separation of the system takes place. The service identification is covered in Section 3.1. Results of this phase are activity diagrams modeling the run of service functions.

A first elaboration of the services, which are identified in the Service Identification Phase, is worked out in the *Use Case Modeling Phase*. The flow of events of every service function (as a subfunction of a service) is specified as well as the input and output data, preconditions for the processing, security requirements and service dependencies. The Use Case Modeling Phase is discussed in detail in Section 3.2. As a result, this phase leads to a use case model with a structured textual specification of the use cases, sequence diagrams for one or more flows of events for a service function, a textual description of the security requirements, and a *logical service architecture*.

The sequence diagrams from the Use Case Modeling Phase make up a first version of the analysis model, which will be worked out in the *Service Modeling Phase*. The behavior of a service is specified formally in an abstract way by relating its inputs and outputs, e.g. using state transition diagrams. Execution scenarios are derived as compositions of services (logical architecture), and the security requirements are concretized in terms of the formal model. The service modeling is described in Section 4.

The next phase, the *Component Design Phase*, ends the service specific activities in the process. Here, the services are mapped to system components. Thereby the logical service architecture, which provides a functional view on the system, i.e. the definition of services and service dependencies, respectively, is transformed into a system architecture. Until the Component Design Phase, no structural constraints are taken into account. Here, the services can be mapped to one or even a number of component architectures. However, designing components with respect to composing services into components is not the focus of this paper. For detailed investigations, see for instance [8, 22, 23]. The result of the component design is a set of components with assigned services and a behavioral modeling of the components.

Afterwards, the system development is commonly (e.g. according to [15]) continued with the *Construction Phase* and the *Transition Phase*. These phases can be carried out conventionally, as service-specific issues have been resolved previously by the above mentioned mapping.

3.1. Service identification

In this phase, requirements have to be divided and they have to be arranged to actors, whereby actors can be represented by roles, systems or services.

In a first step, the requirements of the requirements specification list from the former phase are transformed to flows of activities, which are sufficiently fine-grained such that each activity can be carried out by one actor.

In a second step after this division, we have to arrange these activities to their executing actors, i.e. the actor who gets information from this activity or who sends information to it. In the model, the actors are swim lanes within an activity diagram, the nodes are the activities and the arrows show their causal (and temporal) relationship.

Since we deal with service-based modeling we have to extend the actor model. Conventionally, we have to deal with two types of actors. The first actor type is the abstraction of a real person within a role, e.g. a driver role as abstraction for a person who drives and operates a car.

The second actor type represents external systems interacting with the system to be developed (e.g. via the different automotive bus systems), which are not part of the system to be developed. For service-based development, we add a third actor type, a *service*. The system to be developed is not modeled by one actor but rather by a set of actors of type service, i.e. small self-contained functional entities responsible for a number of activities belonging together. Since services interact both with the above mentioned two actor types and with other services, we can treat service-interactions in the same way as other actor-interactions.

In the context of service modeling, we call the activities performed by actors of type service *service functions*; otherwise we call them *actions* (human actor) and *system functions* (other interacting system), respectively. The needed functionality of a service is given by all service functions assigned to the corresponding service actor. In such a way we build up a usability driven model of services and service functions. For a better understanding, we annotate the swim lanes with actor type symbols: an arrow symbol represents a service, a stickman stands for a human actor, and a box symbol for other interacting systems.

Example An example of an activity flow from our case study is given in Fig.3, namely for the requirement *Display*. We have to deal with the human actor *Driver*, the subsystem actor *Automotive* and the service actors *Customization Service*, *Menu Service*, *CD-Changer Service*, *Telephone Service* and *Display Service*, all represented by swim lanes. Service functions are e.g. *show menu in CDSP* or *Stop CD*.

3.2. Use case modeling

In the recent years, the concept of *Use Cases* [14, 13, 2] has become widely accepted within object oriented development methods. The basic idea of this approach is that both the domain objects as well as the user interaction of the system are modeled within early development phases. The use cases are more than a construct for capturing system requirements: they drive the whole development process and they provide major input when finding and specifying classes, subsystems, interfaces and test cases (for more information cf. [13]). Furthermore, use cases are adequate for an iterative development.

While uses cases have already been integrated into object oriented development, they do not cover aspects of service-based modeling. In this case, we have to deal with the following particularities:

There is no need for a *common domain model*, which has to be worked out e.g. in information systems during the business modeling and has to be refined during the Use Case Modeling Phase. Services only deal with a subset of system objects, the ones we need for information storage within the services and as input and output objects for the service.

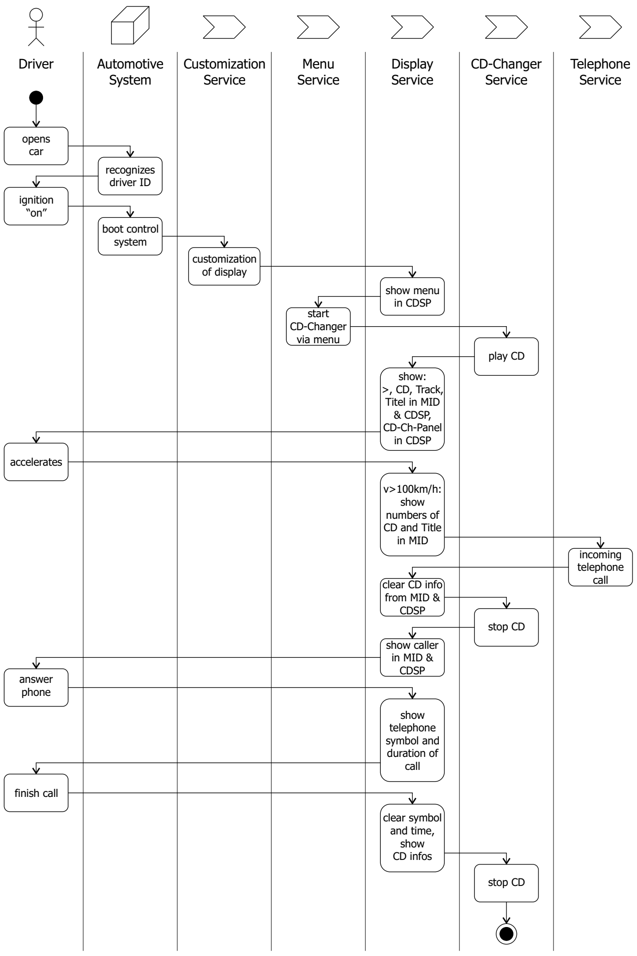

Figure 3. Activity diagram for the requirement "Display"

Furthermore, we consider *security requirements* within this phase. In common systems, we describe use cases in a structured textual way. Beside the actors a textual description covers the processing, variants illustrate peculiarities in the processing, the types of input and output data are specified and often a precondition for the execution is given. Here we add a security section where we describe possible threat scenarios for every protection goal. Thereby we have to check possible violations of the protection goals confidentiality, authenticity, integrity, non-repudiation and availability. More information about security within the requirements phase is given in [3].

Finally, we identify the "involved services" that highlight all concerned services. After that, we can refine the use case and identify connections between the involved services, as indicated in Fig.1. This leads to a *logical service*

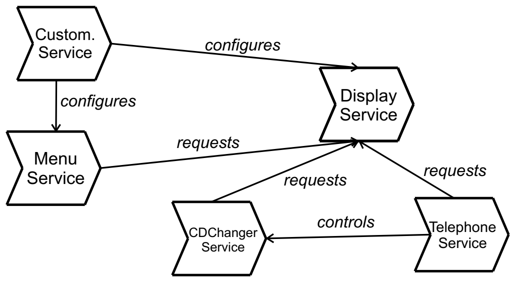

Figure 4. Logical service architecture

architecture, which provides us a structured view on the system functions and how they are related (cf. [4, 5]). At first, the relations are not further specified. The most abstract relation is the "*interacts*" relation. In later steps, the logical service architecture is refined and the relations are stated more precisely like "*configures*", "*requests*" or "*controls*" (see Fig.4). The logical service architecture replaces in our approach common use case diagrams, because these diagrams do not support the necessary refined relations.

In our service-based modeling, we make use of a structured textual description. For a complete service use case model, we have to include every service function we have identified in the Service Identification Phase (cf. Section 3.1) in such an extended use case. Note that this is a partial description, since the activity flow diagram shows just an exemplary run of the service. We obtain a complete service description by merging all partial use case descriptions.

In addition to the textual description, we build in the Use Case Modeling Phase first analysis diagrams in form of sequence diagrams. For each service function with its belonging textual description, we model the textually specified processing within a sequence diagram where we depict the message flow between the different actors.

In the following, we sum up the steps which have to be carried out for service functions to model them as use cases. Note that we have no step for building an object model, because we do not have a domain model in service-based modeling as mentioned above.

1. Elaborate structured use case descriptions.
2. Specify threats and protection goals and add them to the textual use case descriptions.
3. Identify service relations and add them to the textual use case descriptions and work out parts of a logical service architecture.
4. Formalize the use case descriptions exemplarily in one or more sequence diagrams.

Example Table 1 shows the use case description for the Display Service according to Fig.3. Fig.4 shows the logical service architecture and Fig.5 the corresponding sequence diagram for our use case.

Field	**Description**
Use Case	Display
Actors	Menu Service, CD-Changer Service, Driver, Telephone Service
Precondition	Control system booted, ignition *On*
Processing	After the control system boot, the display will be customized from user and cartype settings and the display service shows the main menu. After an incoming *StartCDChanger* request, the CD-Information will be shown on the display. If the current speed is lower than or equal to 100 km/h, detailed CD information will be displayed. If the car drives faster, only little CD information may be displayed. An incoming telephone call interrupts the CD playing and replaces the CD information with the telephone number and the call duration. If the speed is greater than 100 km/h, only a telephone symbol will be displayed. When the call is finished, the CD player goes on with playing at the stored position and displays the CD information.
Variants	The CD information is not available and must be downloaded from the Internet and stored in the system.
Input	Display requests from Menu Service, CD-Changer Service, Telephone Service, Automotive System, Driver
Output	Output messages and output strings for the display unit.
Protection Goals	*Confidentiality:* No threats. *Authenticity:* To ensure for internal communication. We assume that no unauthorized sender may write on any display. *Integrity:* We assume that for the internal communication, integrity will be provided by the system. *Non-Repudiation:* No threats. *Availability:* No threats.
Involved Services	Menu Service, CD-Changer Service, Telephone Service, Customization Service

Table 1. Use case description for the partial display service

4. Service modeling and tool support

In the Service Modeling Phase, we use the sequence diagrams and the textual use case descriptions developed in the Use Case Modeling Phase as a basis to develop a more detailed analysis model. The analysis model we employ is a formally (i.e., mathematically precisely) defined model for service architectures.

The analysis model of a service-based system consists of a number of actors, which are of the three actor types described above. For the development of service-based systems, we focus on the specifications of the service actors. Human roles and external systems form their environment.

We specify the behavior of services as partial functions from sequences of inputs to sets of sequences of outputs. Partiality is characteristic for services in that only the service relevant behavior is specified. The security require-

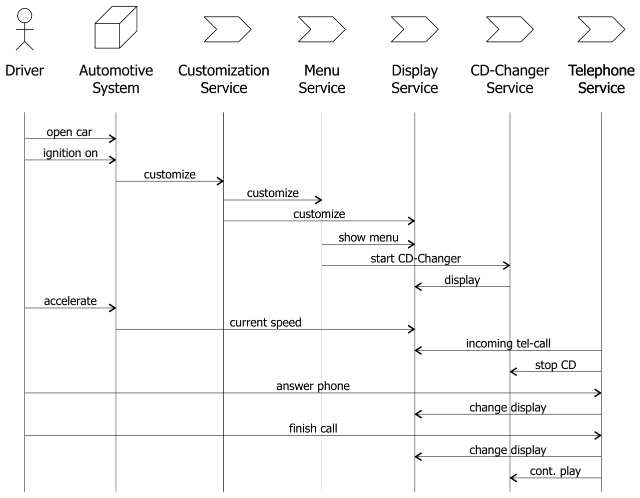

Figure 5. Sequence diagram for use case

ments are concretized in terms of this model, as properties in temporal logic.

4.1. Tool support with AUTOFOCUS

For modeling service architectures and behavior, we use the tool AUTOFOCUS [12, 24]. AUTOFOCUS is a CASE tool for graphically specifying distributed systems. It is based on the formal method FOCUS [6], and its models have a simple, formally defined semantics. AUTOFOCUS offers standard, easy-to-use description techniques for an end-user who does not necessarily need to be a formal methods expert, as well as state-of-the-art techniques for validation and verification. Through various tool connections, it features simulation, code generation, test sequence generation and formal verification of the modelled systems.

Systems are specified in AUTOFOCUS using static and dynamic views, which are conceptually similar to those offered in UML-RT, a UML profile for component-based communicating systems. In AUTOFOCUS, the behavior of services, i.e. the above mentioned function from input sequences to sets of output sequences, is modelled in form of extended finite automata. An example for a more abstract specification using general relations between sequences of inputs and outputs is described in [23] (in the formal method FOCUS).

To specify systems, AUTOFOCUS offers the following views:

- **System Structure Diagrams (SSDs)** are similar to data flow resp. collaboration diagrams and describe the structure and the interfaces of a system. In the SSD view, a system consists of a number of communicating components resp. services, which have input and output ports (denoted as empty and filled circles) to allow for receiving and sending messages of a particular data type. The ports can be connected via channels, making it possible for the services to exchange data. SSDs can be hierarchical, i.e. a service belonging to an SSD can have a substructure that is defined by an SSD itself. Besides, the services in an SSD can be associated with local variables.
- **Data Type Definitions (DTDs)** specify the data types used in the model, with the functional language Quest [20]. In addition to basic types as integer, user-defined hierarchic data types are offered that are very similar to those used in functional programming languages such as Haskell [25].
- **State Transition Diagrams (STDs)** represent extended finite automata and are used to describe the behavior of a service in an SSD. The automata consist of a set of states (one of which is the initial state, marked with a black dot) and a set of transitions between the states, where each transition t is annotated with
  - $\mathsf{pre}(t)$, a boolean precondition (guard) on the inputs and local variables
  - input patterns $\mathsf{inp}(t) = \mathsf{inp}_1?\mathsf{pat}_1; \mathsf{inp}_2?\mathsf{pat}_2; \ldots$, specifying that values are to be read at the ports inp_i that should match the patterns pat_i (terms in the functional language that specify values of data types and can include variables). During the execution of t, variables in the patterns are bound to the matching values.
  - output expressions $\mathsf{outp}(t)$ of the form $\mathsf{out}_1!\mathsf{term}_1; \mathsf{out}_2!\mathsf{term}_2; \ldots$
  - postconditions $\mathsf{post}(t)$ of the form $\mathsf{lvar}_1 = \mathsf{term}_1; \mathsf{lvar}_2 = \mathsf{term}_2; \ldots$

  In the concrete syntax of the STDs, the annotation is written as $\mathsf{pre}(t) : \mathsf{inp}(t) : \mathsf{outp}(t) : \mathsf{post}(t)$. Leaving out components is interpreted as true for preconditions, and as an empty sequence in the other cases. A transition is executable if the input patterns match the values at the input ports and the precondition is true. At each clock tick, one executable transition in each service fires, outputs the values specified by the output patterns and sets the local variables according to the postcondition. The values at the output ports can be read by the connected services in the next clock cycle.
- **Extended Event Traces (EETs)** finally make it possible to describe exemplary system runs and test cases, in a similar way as sequence diagrams (such as the one depicted in Fig.5).

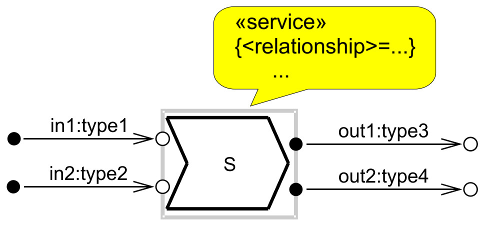

Figure 6. Service actor

We make extensive use of AUTOFOCUS in the service modeling phase. Fig.6 shows the representation of a service S. Services have an interface consisting of input and output ports (as described above, denoted by empty and filled circles) and communicate with the environment via typed input and output channels connected to the ports. The arrow symbol inside the grey border and the annotation «`service`» (in the annotation bubble) identifies this actor as a service. Besides, a service in AUTOFOCUS carries as annotations the relationships to other services identified in the use case modeling phase, denoted in the form `{<relationship>=<related-services>}`. Furthermore, one can specify attributes, like quality of service (QoS) properties. An interesting one is to specify whether a service is adaptive or not. Other QoS-attributes may be time constraints, e.g. that the service answers a request in a given amount of time. The latter can help to guarantee that the Display Service changes the display quickly when the driver accelerates to a speed greater than 100 km/h.

In AUTOFOCUS the behavioral functions are specified in an executable way, as STDs. AUTOFOCUS can be used to model an actual execution scenario, given as a network of actors (roles, systems and services) connected via channels. These models provide a basis for consistency checks, behavioral verification of safety and security properties specified in temporal logic, test case generation, code generation, and further development (Component Design Phase).

Fig.7 shows the system structure diagram (SSD) of the execution scenario corresponding to the sequence diagram in Fig.5.

We notice that most of the communication channels between the services and the system (automotive system) are derived from the sequence diagram, which helps us to build the SSD. The driver is not modeled explicitly and forms the environment of the system. Inputs from the environment are received via the Automotive System and the User Input Service. The latter is an additional service we introduced during the service modeling phase in order to distribute user inputs to the respective services. Furthermore, there are two channels leading to the environment. These channels represent the output to the physical displays, the MID and the CDSP. For the simulation mode, AUTOFOCUS provides an

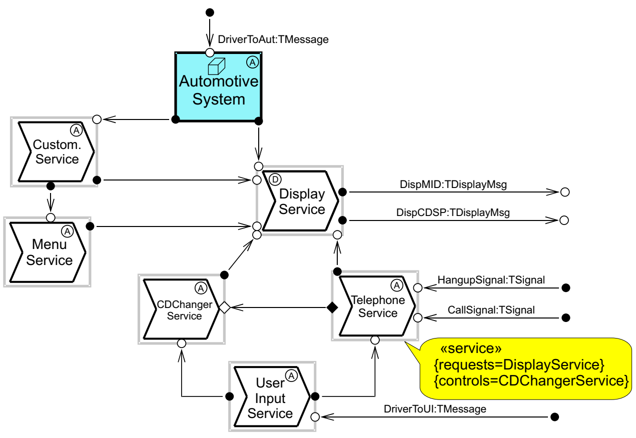

Figure 7. SSD of the "Display Scenario"

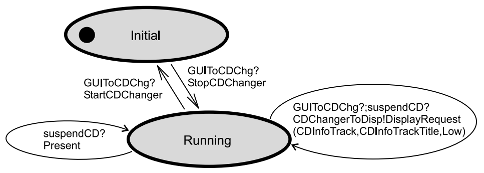

Figure 8. STD of the CDChanger service

interface where the user easily can interact with the modeled system.

The Ⓐ inside the service boxes indicates that there is an STD attached to the service, which specifies its behavior. As an example, Fig.8 shows the behavior model (STD) of the CDChanger Service. Furthermore, a service can be decomposed into more than one STD, which is denoted by a Ⓓ (as in the Display Service, Fig.7).

The annotations in Fig.7 also show that the telephone service has a "requests" relationship to the display service and a "controls" relationship to the CD changer service, as specified in the logical service architecture (see Fig.4). These annotations form the basis for *consistency checks* for the execution scenario. A simple example is that a service with a "requests" or "controls" relationship to another service must at least have an output port that connects it to the other service via a channel (which is fulfilled in our example). We intend to extend the semantics of the annotations to be the basis of model transformations, such as the insertion of an appropriate protocol into the model.

4.2. Model checking with SMV

In this subsection, we describe how to verify a service model with regard to safety and security properties. For this purpose, AUTOFOCUS generates an input file for the symbolic model checker SMV, which carries out the actual model checking process using symbolic model checking based on Binary Decision Diagrams (BDDs) [16]. We specify the required properties of the system using the temporal logic CTL (Computation Tree Logic, see [10]). In addition, to facilitate the formulation of properties, AUTOFOCUS supports the definition of specification patterns (such as those presented by Dwyer et al. in [9]) appropriate for the specification domain that are automatically translated to standard CTL formulas.

The properties are translated to the SMV language as well, and during the model checking process, SMV checks if they are true with respect to the model. If SMV finds any error in the system, a message sequence chart is generated which helps to understand what went wrong and helps the developer resp. service engineer to fix the problem.

At the modeled abstraction level, model checking performance was not an issue as the computation time is in the order of a few minutes per property. For example, one of the required properties for the Display Service is that the message "Incoming call" can only be shown on the display if really an incoming call took place before. In our formalism, this is specified by $\mathsf{precedes}(\mathsf{is\_Msg}(\mathsf{CallSignal}), \mathsf{DispMID} == \mathsf{IncomingCall})$, which could be shown to be true for the model. Here, $\mathsf{precedes}(s, t)$ is a specification pattern translated to the CTL formula $\neg\,\mathsf{E}(\neg s\,\mathsf{U}\,t)$ meaning that there should be no execution where t is fulfilled at some state without s being fulfilled at some earlier state. Another important property is that the title of the played song is not shown on the MID at a speed of 150 km/h. This is specified by $\mathsf{AG}((\mathsf{AutomotiveSystem.current\_speed} == \mathsf{Speed150}) \Rightarrow (\mathsf{AX}(\mathsf{AX}(\mathsf{not}(\mathsf{DispMID} == \mathsf{CDInfoTrackTitle})))))$, meaning that at all reachable states (AG) with current speed $\mathsf{Speed150}$, the display does not show $\mathsf{CDInfoTrackTitle}$ two clock ticks later ($\mathsf{AX\,AX}$). The reason for the delay of two clock ticks is that this is the time that it takes the display in the specification to react to a change of the current speed. Again, this property could be verified to be true.

4.3. Code generation

From AUTOFOCUS models, code can be generated by the tool in various target languages such as Java, C or Ada. We applied code generation to create a prototype of the user interface for the considered service scenario. As a target language, we used Java. Fig.9 shows the structure of the prototype. The actual graphical user interface (GUI) is designed using the GUI building features of a Java IDE. The Java code generated by AUTOFOCUS offers an API to set input values, trigger execution steps and read output values. In addition, some glue code is necessary to translate inputs from

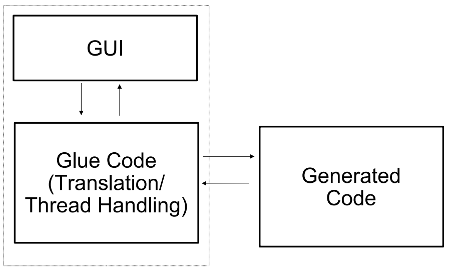

Figure 9. Structure of prototype

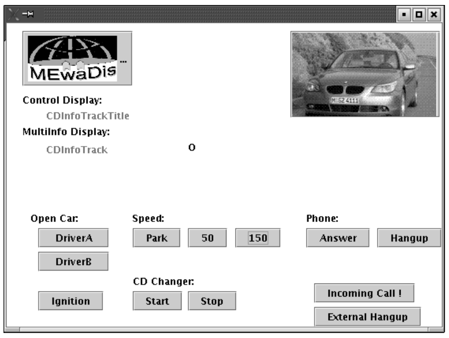

Figure 10. Prototype screenshot

the GUI to inputs to the generated Java code, display outputs from the generated Java code accordingly and perform thread handling (triggering execution steps at fixed time intervals). Automated generation of such glue code is under consideration, but not yet supported by the tool.

A screen shot of the prototype is shown in Fig.10. The user interface of the prototype consists of buttons corresponding to actions that can be initiated by the driver (such as "Start CD Changer") or that can originate from the external system (such as "Incoming Call").

Such prototypes are particularly useful in the development of multifunctional systems where functions influence each other and can be triggered via several different parts of the user interface: without knowledge of the design, the user can examine the modeled behavior for different possible interactions and give feedback to the developer. During the development of the prototype for the display scenario, we found a number of undesired interactions (for example, a completely empty display for one time tick during an incoming phone call) that could easily be prevented by correcting the model.

Further applications of code generation include:

- **Behavior monitoring**, i.e. having the generated Java code for a service run against an independently developed implementation and verifying if the input/output behavior corresponds to the specification.

- **Deployment of the generated code**, i.e. using the generated code itself as part of the implementation.

For this purpose, AUTOFOCUS features the generation of server code (listening on some TCP socket) from AUTOFOCUS components, which are tailored to the service infrastructure. In our case study, the service infrastructure is an OSGi [19] platform, a Java framework supporting the deployment of services. The services are mapped to a number of so-called OSGi "bundles". The Java code generated by AUTOFOCUS is post-processed such that the AUTOFOCUS services are registered as OSGi services and export their interfaces (Java methods to read and write the ports and to trigger the execution). The main application consists of glue code importing the services and managing their communication. Using this approach we generated a version of the prototype described above running on actual car hardware and interfacing e.g. to the built-in CD changer.

5. Conclusion, discussion and future work

To manage the large number of mutually dependent functionalities and the increasing complexity in current and future systems (as in the automotive domain), we presented the service paradigm and a model-based development approach based on the concept of services as the basic building blocks in the elaboration phase of development. We abstract as long as possible from concrete component architectures but instead focus on the system as a logical network of units of function (the services) with abstract behavioral patterns and relationships to other services.

We showed how to apply the concept of service in a phase oriented development process, with focus on particular modeling techniques for the service related phases. In order to use, manage and accept such a service paradigm with its assigned modeling techniques in practice and theory an adequate tool support is necessary. In this paper, as an integrated part of the process, we applied the CASE tool AUTOFOCUS for service modeling. Furthermore, AUTOFOCUS can be used in this context for the definition of execution scenarios, simulation, code generation for prototypes and for the verification of typical requirements of service-based systems. Such a verification includes consistency checks and the proof of security properties.

Service oriented modeling techniques are a fairly new field of research. Some related work, focusing on features and feature interaction, can be found in [28, 27]. AUTOFOCUS has been used for verifying distributed systems and security, e.g. in [11, 26]. There is a large number of modelling languages for the development of systems consisting of distributed objects, such as the UML EDOC profile [18] and the parts of it incorporated into UML 2.0 or various architecture description languages. For UML models, tool support can be offered based on the Model Driven Architecture (MDA) [17]. The general concepts presented in this paper — mainly, the use of a well-defined, formal concept of services with specific properties (see Section 1) and its application throughout the development — does not depend on the use of AUTOFOCUS and its particular description techniques. Its main prerequisites are an executable, extensible component-based description technique with code generation and verification support. We chose AUTOFOCUS in particular because of the comprehensive code generation and verification support and its sound formal basis.

In several student projects, we have executed and evaluated our development process and the AUTOFOCUS tool support. Although the introduced process from Fig.2 seems at first to be a strong waterfall process, a system can be created incrementally. At the beginning only a few services for a simple CD player were worked out in a student project. In further iterations this CD player was extended to a CD changer with CDDB access and furthermore to a configurable packet radio service.

During all these projects we made extensive use of the tool AUTOFOCUS. An execution scenario conforming to the logical service architecture was developed as a system structure diagram and the behavior of the services in state transition diagrams. In the construction phase, the functional parts of the system were generated through the AUTOFOCUS code generator. In the constructions, only the graphical parts and the glue code between user interface and functionality had to be implemented manually. For our stepwise development, this process was very appropriate, because early prototypes could be built of the CD player and after extending the model, only few changes had to be done on the glue and display code.

The service modeling phase and the construction phase are well supported by the tool AUTOFOCUS. What is missing at the moment is support for the service identification and for the use case modeling. For the former, an AUTOFOCUS extension for the modeling of activity diagrams will be necessary. For the latter, tool support is a hard task, because the use case specifications are mostly done in a very informal, textual way, where automation can not be applied.

Services, when based on formal behavior specification, are a powerful concept for system integration. However, formal methods are not convenient to handle and are slowing the development process down. We therefore see the future of our approach, especially the formal foundation and the application of service composition [23] inside of tools. Given a graphical specification technique as sketched out in this paper, enriched with a formal foundation, we are able to compose services to components in a far more precise way and detect possible hazards like feature interaction with tools.

At the moment, we plan to extend the presented service-based approach by more fine-grained ways to define relationships between services leading to more comprehensive logical service architecture models. We are working on tool support for the modeling of a logical service architecture as a step towards an integrated CASE tool supporting service modeling. Therefore, we are analyzing the integration of that kind of modeling technique into existing modeling tools such as AUTOFOCUS.

In addition, we will look both at incorporating quality of service attributes and modeling service contexts (such as the current speed of the car) for context-adaptive services. We also plan to develop a generic security model for service-based systems in the automotive domain.

References

[1] MEWADIS website at `http://www4.in.tum.de/~mewadis`. In German.

[2] R. Breu. *Objektorientierter Softwareentwurf – Integration mit UML*. Springer-Verlag, 2001. In German.

[3] R. Breu, K. Burger, M. Hafner, J. Jürjens, G. Popp, G. Wimmel, and V. Lotz. Key Issues of a Formally Based Process Model for Security Engineering. In *Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA03)*, 2003.

[4] M. Broy. Modeling Services and Layered Architectures. In H. König, M. Heiner, and A. Wolisz, editors, *Formal Techniques for Networked and Distributed Systems*, volume 2767 of *Lecture Notes in Computer Science*, pages 48–61. Springer-Verlag, 2003.

[5] M. Broy. Multi-view Modeling of Software Systems, 2003. Keynote. FM2003 Satellite Workshop on Formal Aspects of Component Software, 8–9 September, Pisa, Italy.

[6] M. Broy and K. Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces and Refinement*. Springer-Verlag, 2001.

[7] M. Deubler, J. Grünbauer, G. Popp, G. Wimmel, and C. Salzmann. Towards a Model-Based and Incremental Development Process for Service-Based Systems. In M. H. Hamaza, editor, *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 183–188, Innsbruck, Austria, February, 17–19 2004.

[8] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison Wesley Publishing Company, 1998.

[9] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. 21st International Conference on Software Engineering (ICSE)*, 1999.

[10] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science Publishers, 1990.

[11] J. Grünbauer, H. Hollmann, J. Jürjens, and G. Wimmel. Modelling and Verification of Layered Security Protocols: A Bank Application. In S. Anderson, M. Felici, and B. Littlewood, editors, *The 22nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2003)*, volume 2788 of *lncs*, pages 116–131, Edinburgh, UK, September 23–26 2003. Springer-Verlag.

[12] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch. Tool supported Specification and Simulation of Distributed Systems. In *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164, 1998.

[13] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley Longman, Inc., 1999.

[14] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison Wesley Longman, Inc., 1992.

[15] P. Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, 2000.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[17] OMG. MDA Specifications. Available at `http://www.omg.org/mda/specs.htm`.

[18] OMG. UML Profile for enterprise distributed Object Computing (EDOC) v1.0, 2004. Available at `http://www.omg.org/technology/documents/formal/edoc.htm`.

[19] Open Services Gateway Inititative. OSGi$^{\text{TM}}$ Service Platform Specification. Release 3, March 2003, `http://www.osgi.org`.

[20] J. Philipps and O. Slotosch. The Quest for Correct Systems: Model Checking of Diagrams and Datatypes. In *Asia Pacific Software Engineering Conference 1999*, 1999.

[21] W. W. Royce. Managing the Development of Large Software Systems. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 328–338. IEEE, 1987.

[22] C. Salzmann. *Modellbasierter Entwurf spontaner Komponentensysteme*. PhD thesis, TU München, 2002. In German.

[23] B. Schätz and C. Salzmann. Service-Based Systems Engineering: Consistent Combination of Services. In *Proceedings of ICFEM 2003, Fifth International Conference on Formal Engineering Methods*. Springer-Verlag, 2003.

[24] O. Slotosch. Quest: Overview over the Project. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods – FM-Trends 98*, pages 346–350. Springer LNCS 1641, 1998.

[25] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

[26] G. Wimmel and J. Jürjens. Specification-based Test Generation for Security-Critical Systems Using Mutations. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 2495 of *Lecture Notes in Computer Science*, pages 471–482, Shanghai, China, Oct. 22-25 2002. Springer Verlag.

[27] P. Zave. Formal description of telecommunication services in promela and z. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, Proceedings of the Nineteenth International NATO Summer School, pages 395–420. IOS Press, 1999.

[28] P. Zave. *An Experiment in Feature Engineering*, pages 353–377. Programming Methodology. Springer-Verlag, 2003.