ABSTRACT
Software development for the automotive domain is currently subject to a silent revolution. On the one hand, software has become the enabling technology for almost all safety-critical and comfort functions offered to the customer: Cars have evolved into distributed computing platforms, which host dozens of networked and interacting processors, implementing hundreds of software-enabled features. On the other hand, the complexity induced by this large number of functions, their interactions, and their supporting infrastructure is becoming the limiting factor for automotive software development. Industry standards, including OSGi[1] and AMI-C[2], address these issues by introducing service-based software-architectures and corresponding middleware layers as modeling and deployment abstractions. This marks a significant shift from component- to service-oriented software development in the automotive domain. As a consequence, development methods and tools for analysis, design, implementation, and quality assurance of automotive software services are dearly needed. This paper reports on a service-based software development approach currently being pursued as a collaboration between Ford and the University of California at San Diego. Its centerpieces are the capturing and design of the interaction patterns defining service executions on distributed platforms, the mapping of the captured interaction patterns to concrete deployment architectures, and the validation of system executions against the resulting specifications.

INTRODUCTION
The increasing complexity of automotive electronics systems has led to the evolution of distributed multi-network systems within a vehicle. Today's cars have 20 to 30 processors in them, with as many as 80 in fully equipped luxury vehicles. In some vehicles the entertainment system alone uses nearly 20 processors, occupied with processing and delivery of digital media, including radio signals, CDs, MP3 files, DVD movies and other data; this data enters the car in the form of hardcopies (disks, tapes), portable players (PDAs, MP3 and DVD players), and increasingly via the Internet. Because of the rapid development of standards, content, and applications supporting the digital lifestyle of passengers and drivers, the complexity of automotive subsystems and networking infrastructures for digital media processing and delivery will grow even more significantly in the foreseeable future.

The traditional top-down systems engineering approach to vehicle electronics software integrates application software into individual components directly on top of the low layer hardware support code. However, the features supported by automotive software and electronics are increasingly dependent on the interactions of distinct components designed by different suppliers. Because of the increasing level of interaction between different components, the top-down approach is no longer adequate. As a result automakers are moving towards defining a middle layer of software that is organized in terms of services. Ford Motor Company, for instance, is investigating a service-based software architecture for its next-generation vehicles that is based on the AMI-C standard for vehicle services.

Defining the semantics of the services in this layer is one of the major challenges that must be resolved to carry this approach through.

A major technological advantage of a service-based vehicle-electronics software architecture over a traditional component-based one is the ability to move the hardware-module-oriented partitioning of the vehicle system to a later point in the design cycle, allowing greater flexibility in integrating functions into hardware and potential elimination of redundant hardware across the vehicle. To leverage this advantage it is imperative to be able to model the vehicular software architecture
on multiple levels, from static models of software structure to executable, time-accurate models of the ultimate system. This, in turn requires specifications for services that are sufficiently formal to allow tools to be built that check the integrated architecture for consistency and completeness, and to allow modeling tools to use the service-oriented specifications directly. The remainder of this paper is structured as follows:

- First we develop the need for a service based software architecture in the vehicle by examining two examples in the area of Body systems and infotainment systems and the issues that they pose when developed under the current component based architectures.

- In the next section, we define what we mean by a service and by a service based architecture and describe how these concepts would be applied to one of the two examples of the previous section.

- In the third section we discuss what is required to engineer a service based architecture in terms of tools and methodologies and describe the ongoing research that we are engaged in to construct such tools, as well as prototype vehicles that we have built that incorporate service based architectural concepts.

THE NEED FOR A DISTRIBUTED SOFTWARE ARCHITECTURE

Microprocessor based automotive electronics has grown from single processor systems in the late seventies to systems that contain 20 to 30 processors in a typical vehicle today. The amount of software in these systems has grown even more rapidly. More significant than the growth in processors and code, however, is the growth in interactions between different components of the system. Virtually all of these processors are connected by in-vehicle networks, and a vehicle may contain from three to five networks. Initially, in-vehicle networks were used to limit the number of wires in the system. Once in place, however, networks enabled components to interact with other subsystems to obtain information that would lead to improved or more reliable functionality. Even a simple function such as locking the vehicle now interacts with a significant number of other functions. Figure 1 shows a simplified context diagram for a central locking system. There are not only interactions with the obvious modules, such as those controlling the individual door locks, but with less obvious systems as well, such as the vehicle speed sensor (to implement lock on drive away), the exterior lights (for remote lock acknowledgement) and the radio tuner and seat controllers (for setting driver preferences on unlock).

The various interacting features in such a system are distributed across a number of different component modules, which are typically produced by different suppliers. Each supplier is responsible for the design and implementation of the software in the module that they produce. As interactions between different subsystems increase, the features themselves become distributed across a number of components. This leads to increasing integration issues as features come to be implemented by software produced independently by a number of different suppliers.

Fig. 1. Central Locking System (CLS) interactions

The need for a component independent software architecture is even greater in the infotainment arena, because there is even a greater variety of configurations in entertainment systems than in body systems. In addition, we are beginning to see the implementation of systems in which consumer devices, such as cell phones, are brought into the vehicle and are expected to work with the vehicle. An example of a feature that is best implemented as a service rather than as a part of a component is a voice recognition system. Voice recognition may first be provided in a vehicle to support hands free phone operation. However it is also useful for a navigation system and to provide part of the human machine interface (HMI) for vehicle functions such as climate control and control of the audio/entertainment system. Different vehicles produced by an OEM may implement these systems using different components from different suppliers, or may provide different subsets of these systems. The optimal implementation architecture for such a vehicle depends heavily on the vehicle configuration. The most effective location for the voice recognition system might be in the navigation system, because of the opportunity to share resources, but in vehicles that do not have a navigation system, another component, such as the audio head unit, must
house the voice recognition software. For both configurations, the voice recognition must present the same service interface to the climate control system. Issues such as these have led feature integration to be the most significant part of the system's implementation.

The figure below illustrates a component based infotainment architecture, while the following figure shows a service based architecture.

As the figure indicates, applications become pluggable in the sense that they do not need to be rewritten to accommodate specific configurations of vehicle hardware or of infrastructure software. In the long run this will lead to a plug and play architecture in which new applications can be added to the vehicle over its lifetime.

**FROM COMPONENT-ORIENTED TO SERVICE-ORIENTED DESIGN**

Before we address the exact meaning of services and service-oriented software architectures, we first look at current practices for developing embedded automotive systems using the central locking system (CLS) example of the preceding section. This example allows us to illustrate the key concepts of our approach comprehensively yet concisely. We will show how these practices lead to the automotive systems architectures illustrated in Figure 2. Then we clarify the notion of service we are proposing. A central aspect of our service definition is that it combines syntactic and semantic information about the interaction protocol underlying a service, which leads to methodologically rich specification, design and implementation support. Finally, we discuss a development process focusing on services, and leading to service-oriented software architectures. Along the way, we will detail the transition from components to services motivated in the Introduction.

**STATE OF THE ART**

A traditional, component-oriented approach to designing automotive software might proceed as follows: (1) Determine the functionality to be implemented for the vehicle; in our CLS example, this would include the locking/unlocking of doors, trunk, and – possibly – windows. (2) Consider the hardware or deployment infrastructure of the specific vehicle in which the functionality is to be provided; this entails, among others, identifying what bus-systems are being used, what signals and time windows for issuing them are available. (3) Develop a specification of the system to be designed based on these findings; (4) hand off this specification to a supplier; (5) Supplier implements the functionality in form of a new ECU (or as an “add-on” to an existing one), and performs unit tests; (6) Supplier hands off the ECU (prototype) to the OEM for further unit and integration tests.

This process with its strong focus on the construction of individual ECUs, starting from more or less detailed specifications provided by the manufacturer has led to vehicle software/hardware architectures as depicted in Figure 2. Here each piece of functionality (or a small number of them) is implemented by the supplier as a separate “box”; this box is attached to a common communication infrastructure. Integrating them and the myriad of interacting software functions they implement...
into a coherent, functional, and safe overall product incurs substantial cost and effort for the manufacturer in later stages of the vehicle development process. Errors that are unearthed at the integration stage of the development process are particularly costly to remedy.

This development process has worked successfully for more then a decade; it is easy to see, however, that it does not scale well to the strong growth of software we are seeing today. Size and weight restrictions, available power and capacity on communication buses, cabling etc. pose strict boundaries on the number of ECUs that can be placed in a car. Furthermore, as we have shown in the Introduction for the CLS example, the different pieces of functionality provided in the car cut across multiple ECUs and networking infrastructures.

If we abstract from the concrete underlying communication bus, but still follow the purely component-oriented development process, we might end up with a system structure as indicated in Figure 4. We indicate components using labeled boxes, and directed communication channels between them using labeled arrows. In this example, we have identified the key fob (KF), a lock manager (LM), the lighting system (LS), the crash sensing system (CS), a database for storing driver preferences (DB), the radio tuner (Tuner) and its user interface (UI), and a central controller (Control) for the CLS. In a real car, most of these entities would be implemented on different ECUs (KF being a likely exception).

The next step would be to have each of these ECUs developed separately. Then, the OEM would integrate the components to yield a fully functional CLS.

There are several interesting points to observe in this example. First, due to the presence of implementation platform specifics early in the development process, the outcome will be highly specialized to the vehicle under design. Second, the same functionality is often needed across vehicle platforms; this is certainly true for the CLS, which today is a standard feature across manufacturers and product lines. Therefore, the entire process has to be repeated again to yield another specialized solution for each target platform. Third, the desired functionality emerges from the interplay of multiple components; in our example, the Control component coordinates the interplay between the key fob, the lock management and the lighting system to implement the locking and unlocking of the car doors with corresponding signaling to the person operating the key fob.

The outcome of the traditional process would be eight separate component specifications; each individual component specification is complete in the sense that it has to address all the different functions the component in question might be involved in. In particular, the cross-cutting nature of the functionality is lost when we look at each individual component; this results in the mentioned labor- and cost-intensive integration effort in late development stages.

Because the overall functionality for CLS is spread out over multiple components, it is practically impossible to address the functions provided by the CLS individually – unless the Control component has been designed to provide a general-enough interfaces that would allow other functions to incorporate CLS functionality into their own. “Remote unlocking” via a cellular network is an example where we would like to have access to individual functions (unlocking) of the CLS.

TOWARDS A METHODOLOGICALLY USEFUL NOTION OF SERVICE

The difficulties with purely component-oriented development have been observed in non-automotive domains as well. The telecommunications domain has long embraced features to refer to pieces of functionality independent from the infrastructure they are implemented on. Recently, the notion of service has attracted attention both in industry and academia as a means for structuring highly complex, distributed and reactive systems. Web services [14] have emerged in the business domain as an attempt at simplifying distribution, publishing, discovery, addressing, and accessing of software functions across the Internet. The key idea here is to assign an identifier (a service name) to a piece of functionality, and to publish the location where the functionality is implemented prominently. As a consequence clients can call the service from anywhere across the Internet.

This approach is appealing also in the automotive domain, because it directly addresses some of the issues identified earlier as drawbacks of the purely component-oriented development approach. For instance, we could identify “lock_doors” and “unlock_doors” as services offered by the CLS to the entire vehicle; other services and applications could use them to integrate the locking or unlocking of doors into their own behavior. As an example, it seems reasonable to compose the “lock_car” service from the “lock_doors” and “lock_trunk” services. A second example is the “remote unlocking” feature mentioned already above.
This idea gives rise to an entirely different software structuring principle. Figure 5 shows an idealized, layered architecture, as it is usually found today in telecommunications systems, operating systems, middleware technologies, and business information systems. Its basis is the notion of service as an individually addressable piece of functionality. At the bottom of the hierarchy we find a layer for hardware abstraction. On top of this layer there is often a layer containing “core” services, which are fundamental to all others. On top of the core services layer typically more general services are located in one or more layers of increasing levels of abstraction. This is often followed by another layer holding a number of applications that make use of the services; often, there is little distinction between a service and an application (other than that the application only calls upon other services, and does not offer services of its own.) Such architectures are often called “service-oriented” architectures; sometimes they are augmented by a “vertical” layer for services that cut across the horizontal layering. The recently established AutoSAR consortium uses a similar organization for the standardized automotive software architecture they promote [3].

We view services and their defining interaction patterns as modeling elements independent of the target implementation architecture. In particular, we identify services as cross-cutting [12] modeling elements that can be implemented on top of any underlying software infrastructure.

Identifying services with the interaction patterns they evoke shifts the development focus from individual components to their collaboration as a starting point of the development process: a service is a projection of the overall system behavior onto one particular task.

SERVICE-ORIENTED DEVELOPMENT

The service notion we have just motivated augments the service notions known from the literature by adding semantic information about the protocol underlying an execution of the service.

Now we build on this service notion, and show how we can use it to create service specifications systematically. To address the challenges of service-oriented software and systems development in the automotive domain we are exploring the development process outlined in Figure 6 [13].

Despite its popularity, however, the notion of service has yet to be established as a first-class development entity in modeling notations, design approaches, and development processes. What defines a service? What distinguishes a service from, say, a method call upon an object in some programming language? What specification techniques and development methods do we need to develop services systematically? How do we map services, once elicited, to corresponding software architectures? How to validate/verify implementations against the captured services? What does adequate tool-support for service-oriented development look like?

Motivated by the deficiencies we have identified in the purely component-centric development process as applied above, we place the interaction of components that deliver a service in the center of the development approach. In other words, a service is defined by the interaction among the entities involved in establishing a particular piece of functionality.
There are interesting relationships between these use cases: the set tuner presets use case has a data dependency with transfer_key_ID; the transferred key ID will influence the settings of the tuner.

The next development step is to identify roles for the use cases we have identified. As a first approximation we can reuse the structure diagram we have shown above as the role domain model. These roles will likely map to a variety of different component configurations depending on the concrete make and model under consideration. For instance, in a concrete implementation, the central controller (Control) and the lock management (LM) might end up on the same ECU, whereas the database (DB) and the lighting system (LS) might reside on others.

To capture the interaction patterns defining services we use an extended version of Message Sequence Charts (MSC) [9,10]. 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 [11]. 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.

Figure 7 shows an example; here we depict the interactions defining the “unlocking” service. It consists of a triggering message “unlock” from the keyfob to the central controller. The latter forwards the “unlock” message to the lock management (LM). By introducing the LM role we abstract from the concrete number of locks present in the vehicle (doors front/back, trunk, moonroof, windows, security system, …) When the locks have been operated, LM returns an “ok” message to the control role. Upon its receipt, the control role issues a “door_unlckd_sig” message to the lighting system role, which handles the signaling of the locks’ states to the driver. In the MSC we also indicate that the control role changes from the LCKD state to the UNLD state; labeled hexagons represent local role states during the course of an interaction.

Clearly, this is just one course of actions that may happen during the execution of the unlocking service. The extended MSC dialect we use enables succinct specification of such alternatives [10, 15].

The next use case we turn into a service is “transfer_key_ID”. Upon receipt of an unlock message the control role sends a getID message to the keyfob; KF sends the id to Control, which relays it to the DB (cf. Figure 8.) Again, Control switches from state LCKD to UNLD in the course of executing the service.
The preceding two services are overlapping in the sense that both share references to the unlock message and states LCKD/UNLD.

To compose these services into an overall service specification we have to identify the overlapping messages, and “synchronize” the execution of the services on these joint messages.

This example illustrates the overlapping of messages within services; this is yet another indication for the partiality of service specifications. Another service dependency emerges when we consider the “start tuner” service. Upon receipt of an “on” message from the UI, the tuner requests the driver presets from the DB to switch to the correct station (cf. Figure 9.) This constitutes a data dependency between services.

The handle_crash service has a particularly simple interaction pattern (cf. Figure 10): whenever the control role receives an “impact” message it responds by sending “unlock” to the lock management role, resulting in the unlocking of the vehicle. Methodologically this can be handled by introducing a “preemption” concept that treats the response of the control role as the handling of a preemption triggered by the “impact” message.

In this example, we have seen that services require composition operators not generally available in component-oriented development: the concept of overlapping components is not very common. Roles, on the other hand, by definition capture a partial view on all components playing that role – to be composed with other partial views to yield the overall behavior of the component under consideration. The composition of the services as elicited above yields a service specification.

The mapping from a service specification to a set of components implementing the services in the next phase of the development process is a design step. This step entails fixing a component architecture, and an association between the components and the roles they play to support the given set of services.

In the CLS example, we could decide, for instance, to have just one component to implement the Control and LM (lock management) roles. This gives rise to a component-oriented “deployment” architecture.

If the target architecture supports the definition and deployment of individual services, however, we can encapsulate the interaction protocols contained in each of the extended MSCs we have presented, and publish those as individually accessible services within service-oriented software architectures as outlined above.

In both cases we gain insight into the key interaction patterns defining services by applying the development process for service-oriented software architectures. The step from the service specifications we have now available to prototypic implementations can be supported by appropriate software engineering tools as we will see in the next section.

TOOL-SUPPORT FOR SERVICE-ORIENTED DEVELOPMENT

The central research challenges we address in our collaborative project is to develop a thorough understanding of the notion of service to the degree it is applicable in the automotive domain, to devise methodological development steps for service-oriented design of automotive software, and to support the service-based development process by adequate tool prototypes.

To that end, we are developing a prototypic tool chain [4] consisting of four main components (cf. Figure 11) – the service specification and modeling interface (M2Code), a connection to validation tools, a code generator, and a component-based simulation testbed that connects these three (AutoFocus, developed at Technische Universität München, Germany.) By integrating an existing component-based modeling tool into our tool-chain we leverage its capabilities with respect to
simulation, verification, and test-case generation within service-oriented specification and design.

![Diagram](image)

**Fig. 11 – Tool-Chain**

At a high level, development using this tool chain proceeds as follows: (1) Interaction patterns are specified as MSCs, and executable specifications (state automata) for the components involved are synthesized using the tool “M2Code.” (2) M2Code exports an XML file containing the specification of component structure and behavior for input to other tools, such as AutoFocus. AutoFocus is a simulation and verification environment for embedded systems; one of its strengths is that it has connectors to a wide variety of model-checking tools and theorem provers. (3) AutoFocus is used for behavior simulation and consistency checking. It can also be used for validating the system in conjunction with the validation tools. (4) AutoFocus exports an XML file for input to the code generator. (3) The code generator outputs executable Real Time CORBA [7, 8].

The purpose of the code generator is not to produce code ready for deployment in the vehicle; rather, we view this as an executable specification, which can be handed off to suppliers. Based on this executable specification the suppliers can validate their implementation. We have chosen Real Time CORBA as the target platform because it holds the promise that even end-to-end Quality-of-Service (QoS) properties can be captured in the executable specification. To leverage this feature, we plan to extend the specification techniques we have developed to also allow QoS specifications as adornments to the interaction patterns defining services. The rationale behind this extension is that most QoS properties are inherently non-local in nature and their fulfillment depends on the interaction of multiple entities. In combination with Real Time CORBA as the implementation infrastructure for executable specifications, this opens up opportunities for *validation* of QoS properties. Our algorithm for creating executable specifications from MSCs [5] can be used to synthesize monitors from interaction patterns adorned by QoS properties: each property is turned into a Real Time CORBA component that listens on the event channel for messages belonging to the interaction pattern, and compares their occurrence times (or number, latency, etc.) with the adorning QoS specification. If there is a deviation, the simulation environment raises a “flag” to alert the developer. Initial experiments in this direction [6] are underway.

This tool chain bridges the gap between service- and component-oriented software design for embedded systems, while leveraging the validation and verification tools available for component-oriented systems also for services. In particular, we can verify systems that have been developed in a service-oriented way, and generate code that complies with this specification.

One of the next steps in our collaborative research will be to move from the “toy” CLS as described for purposes of illustration in this text to a realistic specification of an existing CLS; this will provide insight into the scalability of our approach.

**CONCLUSION**

In this text, we have discussed the silent revolution currently underway in the automotive domain. Driven by the increasing interaction complexity of automotive software, purely component-oriented software and systems engineering approaches are reaching their limits.

Based on their success in other application domains, the notions of service and service-oriented software architectures are becoming increasingly prominent in the automotive domain. Services hold great potential in lowering development and deployment costs across product lines, and increasing reliability by reducing the complexity of automotive software.

To deliver on these promises, a thorough understanding of what defines services and service-oriented software architectures is indispensable. Methodologically founded development steps that guide the developer from capturing service requirements all the way through to the mapping of services onto target software architectures need to be designed.

We have introduced a precise notion of service, placing their key interaction patterns in the center of the development process. We have shown how this notion of service gives rise to a systematic development process and corresponding tool support for modeling, validation and generation of code in the sense of executable specifications. The resulting specifications, automatically generated from service specifications, can be handed off to suppliers for validation purposes.
A thorough evaluation of the proposed methodology on a broad spectrum of automotive services, as well as an integration of Quality-of-Service properties into the specification and validation effort are promising areas for future work.

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.

REFERENCES

11. UML 2.0: http://www.omg.org/uml

CONTACT

Ingolf H. Krüger (ikrueger@ucsd.edu) is leading research efforts in the area of component-based and service-oriented software and systems engineering at the California Institute for Telecommunications and Information Technology, CAL-(IT)². He directs the Service-Oriented Software and Systems Engineering Laboratory (SOSA) within the Computer Science and Engineering Department at UCSD. He designs and conducts innovative software projects in collaboration with the automotive industry.

Edward C. Nelson (enelson7@ford.com) works with the Infotronics Research & Advanced Engineering Group at Ford Motor Company's laboratories in Dearborn, Michigan. He has also served as a software team leader for the Automotive Multimedia Interface Coalition (AMI-C).

K. Venkatesh Prasad (kprasad@ford.com) leads the Infotronics Research & Advanced Engineering Group at Ford Motor Company's laboratories in Dearborn, Michigan.