

# Mastering complexity through modeling and early prototyping

*Reinhard Stolle, Christian Salzmann, Tillmann Schumm*

BMW Car IT, Petuelring 116, 80809 München, Germany

{reinhard.stolle,christian.salzmann,tillmann.schumm}@bmw-carit.de

**ABSTRACT: Modern systems of automotive electronics and software are characterized by a high degree of complexity and heterogeneity. Important techniques in the design, decomposition, implementation and integration of such systems include standardization, modeling and prototyping. In this paper, we describe these techniques and discuss their benefits.**

## 1. Modeling and prototyping

In science and engineering, models are the tool of choice to deal with complex systems. A model is an abstract representative for a concrete artifact or system. Models can represent artifacts that do not exist in reality, which means that modeling is particularly useful for reasoning about hypothetical systems or situations. For example, a seismologist may use models to envision a possible earthquake that may arise from a particular geological situation. Likewise, a civil engineer may use models to design a bridge that will withstand that earthquake. Such models are typically expressed in a suitable formalism – for example differential equations and finite elements in the case of the bridge model. Based on a body of knowledge that is expressed in the same formalism, it is then possible to formally derive predictions about the properties and the behavior of the modeled system.

Aside from the formal, mathematical or logical models described in the previous paragraph, there is also the kind of model that one builds in order to experience and assess the aesthetics – the “look and feel” – of artifacts. For example, the designer of a bridge may build a miniature model of the planned bridge and its surroundings in order to envision whether the bridge’s appearance will be appealing and how well the bridge will fit into the landscape or urban surroundings.

In this paper, we present a selection of modeling approaches and techniques in the domain of automotive software. For clarity of vocabulary, we will refer to the kind of formal, mathematical and logical model as *model*. The kind of model that was exemplified by the miniature

bridge in the previous paragraph will be called *prototype*.

Automotive software systems are characterized by a high degree of complexity because they typically consist of several dozens of heterogeneous, distributed and interconnected embedded control units. The development process for systems of such complexity typically spans several years and involves a large number of stakeholders and organizations. In the early phase of this process, the overall system design is derived from the collection of requirements that the system is supposed to meet. The resulting overall design is consecutively decomposed into smaller and smaller subsystems. The design of these subsystems becomes more and more detailed and finally turns into a full implementation. In the integration phase of the process, the implemented subsystems are step by step composed, which eventually results in an implemented overall system.

In order to detect design (and implementation) errors as early as possible, it is necessary to superpose several cycles of modeling and prototyping onto the process of decomposition and integration. Using models and prototypes, one can integrate hypothetical subsystems without actually fully implementing them. This approach is often referred to as *virtual integration*.

Models and prototypes can play various roles. They can act as stand-ins for components that have not been implemented yet. In this way, they help validate both the design of the function itself and the design of the function’s integration into the overall functional network. Validating the function design itself means ensuring that the function delivers on the original requirements from which the design was derived. Validating the function’s network integration means ensuring that it properly produces what its consumers expect, and vice versa. At a finer-grained level it means validating the communication patterns between functions via the specified interfaces.

Models differ from prototypes with respect to a number of characteristics. As illustrated earlier

in the bridge example, models are expressed in a formal system, making available formal theories and tools to manipulate, operate on, and reason with the models. The goal of modeling is then to formally derive assertions about the designed system. This approach is called *formal validation and verification*. Examples of models include all sorts of equations (differential equations, characteristic curves, etc.) for continuous systems, state machines for discrete systems, mixtures of continuous equations and state machines for hybrid systems, constraint labels for time-critical systems, and others. The assertions that one is able to make using such models typically concern dependencies between state variables, the combinatorics of states, configurations and versions, completeness of state spaces, reachability of states, compatibility of components, and so on. Examples of questions that can be answered using models are “What happens to the heat production if the voltage is doubled?”, or “Is there any state in which the driver cannot launch a phone call?”

As described in the previous paragraph, formal techniques using models are particularly powerful in dealing with complex domains whose basic elements and compositionality can be formalized. Prototypes, on the other hand, are helpful in answering questions about properties or behavior that cannot be easily formalized. Questions of aesthetics, look and feel, usability and appeal are prime examples.

A second reason for building prototypes is the need for validating implementation approaches. Whereas models typically only represent the properties and behavior at the interfaces (“black-box”), prototypes also anticipate aspects of the eventual implementation up to a certain depth of realization. For example, a prototype of a time-critical software application may be realized on top of a hardware platform and an operating system that closely resembles the environment of the eventual target system; in this fashion, one can reasonably expect that observations about the prototype’s properties resemble the eventual system’s properties with a certain accuracy.

A third reason for using prototypes lies in the fact that for some systems or system aspects (such as timing behavior, for example) the relevant properties are often not known or have not been specified with the rigor that would be necessary to allow formal modeling.

The approach of using prototypes as stand-ins to allow experiments and tests with systems that have not been fully implemented yet is called *experimental validation and verification*. One of the main goals of this approach is to draw attention to aspects of the system that

have been overlooked or misjudged in the original design.

A third technique – in addition to modeling and prototyping – for mastering complexity is standardization of interfaces and components. In the following sections, we discuss standardization, modeling and prototyping in more detail. We present examples that illustrate these three techniques and how they are employed in practice.

## **2. Standards as a way of mastering complexity at the application level**

Decomposing a monolithic system into smaller modules is the classical way to lower the complexity and to increase the reusability of software.<sup>1</sup>

Successful decomposition can only be reached by separation of concerns at several levels: architecture, design and implementation (language-dependent). Popular techniques, well known from business IT, are layered architectures or middleware (architectural level), software components, aspects (design) and object-oriented languages such as Java.

To increase the reliability of these reusable software blocks and to facilitate the emergence of a market to lower the costs, standardization of software modules is a successful way. Examples such as codec standards, software libraries illustrate this. In addition to the obvious cost advantage (the bigger the potential market, the lower the prices) another aspect of standardized software should not be missed: the more targets run a software module, the more tested and reliable the software is.

AUTOSAR is an emerging standard for automotive embedded software. In AUTOSAR software is modularized into components. Components are typed and use explicit communication points, so-called ports, for exchanging messages. A port is either provided or required and owns a set of interfaces that define the set of messages that can be exchanged via the port. Two ports of the same type can be connected via a connector instance that represents the channel used by the ports to exchange messages. The AUTOSAR standard allows for the specification of the interfaces, message types and even the communication variants the software modules use in a standardized manner. Therefore an AUTOSAR software component (or module) can potentially be reused across system and company borders.

On the basis of interfaces, specified in a standardized way, further system aspects besides

static compatibility at the level of data types and functions can be tackled. Dynamic aspects of the behavior such as timing constraints and protocols at the functional level could be specified in such a way that the level of composability of independently developed modules can be handled.

### 3. Modeling of function networks

Automotive board nets consist of a high number of electronic devices: up to 70 embedded control units and about 200 to 300 sensors and actuators. The customer can choose among many different car configurations. The large variety of these car configurations is reflected by a large variety of configurations of the electrical system. Typically, in the architecture phase of the development process, several millions of configurations need to be considered.

The massive parallelism in the data board net can be described by function networks. Such networks consist of functions and signals; functions produce output information from input information, and the signal is the information itself. The devices in the board net and their interfaces (e.g., bus communication) are designed by mapping functions onto technical components.

#### 3.1. Verification of distributed architectures

Using the method of function nets, it is possible to model and verify certain properties of the architecture, for example:

- Are all communication requirements fulfilled?
- Is it possible to replace a standard component by an optional component?
- Is it possible to replace a subsystem by a compatible variant?

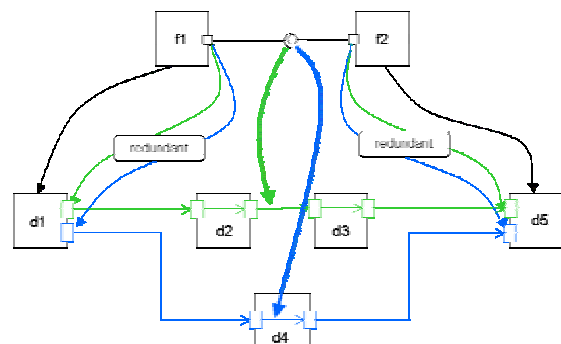
Here, the verification criterion is whether all the interfaces that are required by functions in the network are delivered by functions in the network. This step is a form of static verification, i.e., the dynamic behavior of functions is not considered.

#### 3.2. Architecture development

The massive parallelism in the distributed system requires that the timing and the ordering of communication be specified. The basic idea is to specify properties of functions locally and verify the resulting network globally. For example, the local specification of a function may include its sampling rate or the maximal signal latency. Propagating these local specifications

through the network across the function ports, one can verify whether the network meets all requirements globally.

An additional aspect of function nets is the concept of redundancy. This aspect is especially important for safety-related functions. Again, the redundancy requirements are specified locally with the functions in the function net. After having mapped the function network onto a network of technical devices, one can verify that the redundancy requirements are met. Figure 1 shows an example of such a mapping.



**Figure 1: A redundant realization of a function network: The functions f1 and f2 and the link between them show an example of a function net. This function net is mapped onto the technical components d1..d5. The signal link between f1 and f2 is realized redundantly by the mapping to two communication paths that are independent of each other, namely the path d1-d2-d3-d5 and the path d1-d4-d5.**

### 4. Modeling and prototyping of HMIs

In this section, we illustrate how a flexible prototyping platform is used to gain early feedback on the design of advanced comprehensive human machine interaction (HMI) systems for driver's entertainment and information functionality.

The need for early prototyping is especially critical for systems that concern human machine interaction, that is, systems that directly present information to the driver. The amount of available information is growing rapidly; the scarce resource in this endeavor is the driver's attention; therefore, the design of the HMI must be carefully tuned to present the right information in the right form in the right situation. Likewise, it is important for the driver to be able to make use of the available functionality in a way that is intuitive to the driver and non-intrusive to the main task of driving.

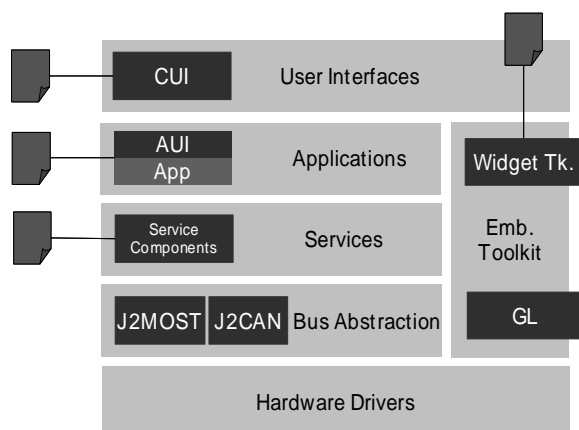
How well a given system design meets these requirements is extremely difficult to judge based on static (i.e., non-executable) design specifications. Similarly, it is difficult to formal-

ize the metrics that measure the quality of HMI design in such a way as to allow static design descriptions to be evaluated along those measures. What is needed is a design specification that can be executed, thereby simulating the behavior of the specified HMI system up to a certain degree of accuracy. In other words, the HMI specification along with its execution framework becomes an HMI prototype. This HMI prototype can then be employed to perform usability tests with human subjects. The experience gained in those usability tests results in design improvements, and the new design is then evaluated in a new round of usability tests. In order to be able to run as many usability test iterations as possible in a given time period, it is important for the prototyping framework to be flexible enough to facilitate typical design changes.

We have built FLUID (Flexible User Interface Development), a prototyping framework that provides exactly this kind of flexibility for a design and prototyping process that involves a number of heterogeneous stakeholders, such as graphic designers, user interface designers, ergonomics specialists, psychologists, specification engineers, software engineers, programmers, and test engineers.

#### 4.1. FLUID framework

The layered architecture of FLUID, shown in Figure 2, was designed to allow for easy modifications of various aspects of the overall HMI system. In this architecture, different user interface aspects can be tried out for the same functionality and vice versa. For example, the same list of traffic jams may be presented as scrollable lists in various sorting orders or as icons in zoomable geographical maps.



**Figure 2: The layered architecture of the FLUID framework provides flexibility in the HMI prototyping phase. Much of the description of the HMI system that is being prototyped takes the form of declarative specifications.**

The bottom three layers hide the details of the communication buses and the employed hardware devices. As a result, the service layer provides a common abstract programming interface for the software modules.

The application layer mediates between the services that are supplied by the devices and the structure of the user interface. It may combine several services into an application, or provide only a particular aspect of a given service. This layer also manages the “business logic” of the HMI, i.e., it controls the dynamics of the interaction between the driver and the system. The so-called “abstract user interface” (AUI) contains abstract control and presentation elements such as buttons, lists, containers, boards, etc.

Finally, the concrete user interface layer (CUI) prescribes for each modality (graphics, haptics, speech) the concrete expression of an abstract element. For example, it specifies the colors, sizes, positions of buttons. It also describes, for example, whether the driver has to shift or turn the iDrive commander in order to navigate to the left on a map.

The widget toolkit is a library of graphical widgets that are the basic elements of the CUI and are implemented based on the graphic library (GL) provided by the embedded processor.

In summary, the layered architecture ensures that local changes to the design of the user’s HMI experience can be realized by local changes to the software. This achievement is noteworthy because traditional monolithic embedded HMI systems seldom provide this degree of flexibility. With FLUID, the desired CUI design can be modified, e.g. adapted to different car models or line fit levels, literally within minutes.

#### 4.2. Models as a by-product of prototyping

As described in the previous section, in our approach HMI prototypes are executable specifications, i.e., they consist of a set of declarative specifications, which can be executed by the FLUID framework. A large fraction of the typical modifications to HMI prototypes (in the course of early prototyping cycles) require only changes to the declarative specifications, without touching the code of the prototyping framework itself. In this way, the result of the prototyping phase is not only a runnable prototype that facilitates the assessment of the look and feel of the HMI system, but also a set of formal specifications that describe the graphics, the widget tree, the widget logic, the dialog logic and the event logic of the HMI system. These specifications can form the

basis for later steps in the process, such as target code generation and test automation.

In a related but different approach to modeling of HMIs, one describes the behavior of the HMI system as a state machine. Typically, a state represents a “screen” and transitions represent screen changes that are triggered by actions of the user or the system. Based on this modeling approach, it is possible to employ the technique of model checking in order to formally derive properties of the modeled system.<sup>2</sup> Examples of properties that can be formally assured in this fashion include the number of user steps needed to change the currently playing radio station, or the reversibility of certain user actions.

The kind of model that is needed to enable model checking is different from the kind of specifications that currently result from our prototyping approach. However, the information that is encoded in those specifications is the same kind of information that is contained in typical state machine models. We are currently working on defining a formal transformation between these two specification formats.<sup>3</sup>

## 5. Conclusion

Both modeling and prototyping are indispensable techniques to ensure early in the process that the system design is valid and that the subsystems will eventually be able to be integrated into a valid overall system. In an ideal world, the transformation from early models via prototypes to final target code follows a deterministic process. Designing the formalisms, techniques and supporting technologies for such deterministic development processes is one of the foci of our current efforts.

---

1 Parnas, D. L.: “On the criteria to be used in decomposing a system into modules.” Communications of the ACM 1972 Vol. 15 Nr. 12 (December) pp. 1053-1058.

2 Kistler, G.: “Ein model-checking-basierter Ansatz zur Prüfung von Nutzungsschnittstellen.“ M.S. thesis, Technische Universität München, Fakultät für Informatik, 2004.

3 Scheickl, O., Benedek, T., Stolle, R.: “Modellierungsarten für Automotive HMIs.“ Modellierung 2006, Workshop Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, Innsbruck, 2004.