Integration of User Interface, Middleware, and Services in Automotive IVI Systems

Björn Decker

comlet Verteilte Systeme GmbH
Amerikastraße 27
66482 Zweibruecken
Bjoern.Decker@comlet.de

Marius Orfgen

Innovative Factory Systems (IFS)
German Research Center for Artificial Intelligence GmbH (DFKI)
Trippstadter Strasse 122
67663 Kaiserslautern
Marius.Orfgen@dfki.de

Abstract: This paper describes a model-based approach for integration of car infotainment systems. These systems consist of different hardware and software components from different vendors with no common standardized interfaces. We propose modeling these components to create a system description that can be transformed semi-automatically into the so-called “glue code” that is used to string the different components together. The approach fosters reuse of glue code fragments, leading to reduced development effort and higher reliability.

1 Introduction

This paper describes the ongoing efforts in the German research project automotiveHMI to improve the development process of infotainment systems in the automotive industry. It specifically deals with the connection between the user interface layer and the application/middleware layer. We will first describe a high-level standardization of the information exchange between these two layers and will then provide a way of handling the varying frameworks and communication concepts found in real-life infotainment projects.

2 A high-level description of UI/application communication

It has been common practice in software projects for decades to separate the parts dealing with the user (the user interface, UI) and the parts implementing the business logic (the actual functions of the program). Reasons are the different development processes of these parts (iterative development for the UI, more-or-less a single big iteration for the business logic) and the separation of concerns concept to keep the overall runtime complexity of the program low. Explicitly separating these two parts in a software project introduces the need for describing the communication behavior between them. Different architectures/patterns like Client-Server, Model-View-Controller[Kra88]

or the Seeheim model[Pfa83] provide examples on how to handle this separation on an architectural level. On the implementation side, there are the general distinctions between message-based communication, allowing for asynchronous behavior in the program, and direct method invocations, providing a more synchronous style. In the first case, messages and their parameters have to be defined and form the software interface between the two program parts. In the second case, a list of invokable methods on both sides forms this interface.

Infotainment development projects in the automotive industry are currently using varying concepts to realize this communication, both from a conceptual as well as from an implementation view. The concrete realization is influenced by the selection of the middleware/framework, which in turn is driven by non-functional requirements concerning communication latency and expected message sizes as well as overall system performance. Since infotainment systems are realized using (low cost) embedded hardware, tuning the performance of these systems using specialized hardware components, frameworks and local optimization is crucial to creating systems that react in reasonable time frames. This very heterogeneous field has to date hindered the conception of a standard description between the UI and applications like a media player or a navigation component.

The German research project automotiveHMI[aHMI] is currently developing an interchange format for infotainment specifications (called the "Infotainment Specification Format", ISF). The goal is to have a common model that can be used for the communication between interaction designers, graphic designers, developers and testers in an infotainment project. Part of these efforts is the creation of a standardized, implementation-independent description of the communication behavior between UI and applications. The current version of this description will be presented here.

From a high-level point of view, the software for an infotainment system consists of the HMI, the middleware providing basic services (e.g. communication channels) and applications implementing specialized functions (e.g. media playback). These components are normally developed by different teams of developers or being reused from older projects. This results in different interfaces and communication concepts, which requires so-called "glue code" to string these components together (see Figure 1).

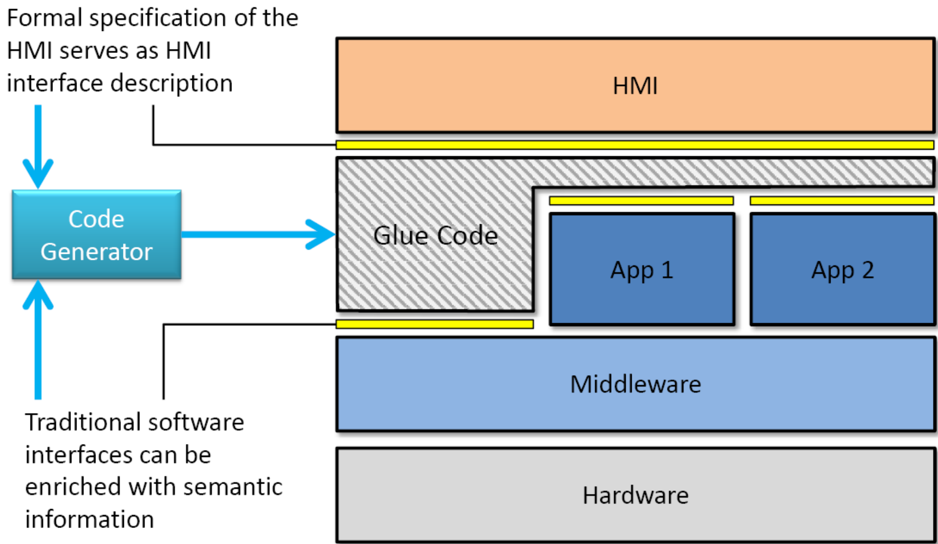

Figure 1: High-level view on the components

The standardized description is realized as a message system. It defines different system components, among them the UI as well as applications and middleware components. Each of these components can send messages consisting of a message type, a message topic and additional parameters. Components can register to one or more message topics and then receive all messages of these topics.

This description allows to model synchronous behavior like method invocations using a pair of request/response messages as well as normal asynchronous communication. It is formal enough to be understood by code generators for the different frameworks and can be mapped to more concrete communication concepts.

One example for this mapping is the use of a framework that has a data-pool concept. In the automotive domain, a data pool is a centralized storage for data concerning the UI as well as applications. The data pool consists of a set of named variables as well as a callback listener concept. Components are either allowed to write into one of these variables or to read from them, being notified if the variable is changed. Using a data pool currently means defining these variables manually, which are in the low to medium hundreds in current infotainment projects. The description would allow here to automatically create a variable for each message and to identify the senders and receivers of each of those messages and register them to the data pool accordingly.

This description does not contain non-functional requirements like the maximum delay for messages, the required bandwidth, etc. These are normally the concerns of the developers, which need to choose a framework based on their experience and the expected communication amount for a system. The description instead allows specifying the communication in parallel to these development efforts and allowing discussing between the different stakeholders in such a project.

3 Integration into the embedded hardware

After adopting the standardized description in a project, the glue code between the UI and the applications could be generated automatically. Unfortunately, in real projects a combination of frameworks, low-level hardware and UIs created by different tools are combined into the infotainment software. Different frameworks, hardware components and UIs lead to a combinatorial explosion that would require to a new code generator for each combination. To solve this problem, an approach for the automatic combination of glue code parts was created, which will be described in the following chapter.

4 A model-based approach for communication design

Developing embedded systems in the automotive industry is very different from standard PC software development. First, embedded systems development involves in almost all cases software as well as hardware development. High cost pressure and very specific non-functional requirements usually do not allow the use of standard hardware platforms. For example, in the automotive industry often extended temperature operating ranges are required by law, but are not guaranteed by standard chips. The selection of hardware components is even more complex as high performance requirements for some components collide with heavy cost pressure. For some use cases, e.g. safety relevant display situations (speed in an instrument cluster), a high display update frequency is required. For other applications the component is chosen which provides just enough resources that are needed. All these considerations lead to the design of irregular hardware platforms and thus have significant influence on the software development for the system.

On the software side, it would be ideal if standard technology could be used throughout the system. Unfortunately, due to the custom hardware design also custom frameworks, middleware and operating systems have to be used. While they are based on standard technology, e.g. CORBA[Sie98] resp. Real-Time CORBA[Sch00] or OSGi[Alv11], they have to be ported to the new hardware platform and implement additional non-functional requirements, such as a complex resource management or guaranteed system responsiveness. The latter example is based on the high quality standards that have to be met in automotive development. Longevity of the addressed systems results in an extended maintenance phase. Additional features have to be built in while at the same time a stable runtime behavior of the system has to be preserved under the given resource constraints. Therefore, supposed little additions may rapidly lead to significant changes throughout the whole system. Long development cycles and the various pressures in terms of budget and deadlines in combination with technical restrictions result in a progressive divergence between the specification and the implementation. This contradicts the high quality standards and at some point forbids further feature development without risking system stability.

In order to face the various problems mentioned above, a standard procedure evolved within the industry. In hardware and software development there is a tendency to build reusable components that can be hierarchically composed to large scale systems. This

enables the use of custom but reusable frameworks for application and HMI development.

One major task of the frameworks is to realize the communication between components. For that purpose they provide a standardized application programming interface (API) that can be used by any component. In order to minimize recurring development tasks that are distinguished only by a handful of parameters and to enable task-specific development tools such as HMI-design-tools to access relevant information, interface description languages (IDL) are widely used, from which the API is automatically generated.

However, they are usually part of the framework and the associated tool chain (see [Sie98]) and therefore described interfaces are not directly transferable between different frameworks. Furthermore IDLs focus on syntactic description of the interface and do not contain any additional information that is vital for accessing a component via that interface. Since components usually are highly runtime-configurable in order to maintain flexibility, performing a supposedly simple interaction potentially involves several complex configuration steps in advance. For that kind of information separate API-documentations provide the required details in textual form. High manual effort is needed to keep separate specification documents from diverging from the corresponding implementation.

The upper half of figure 2 gives a coarse overview over the reference system architecture with a focus on the communicating HMI and services. HMI and services are implemented according to established development methods and by the use of established tools. These tools generate code and provide a tool specific programming interface for communication access. The adaption layer is implemented in order to ensure the HMI and services to work together properly. In current projects adaption layers are implemented manually based on textual descriptions of communication details. They have to implement the specific interface towards the communication end point on one side and the middleware API on the other side. The latter one usually is generated from an IDL depending on the middleware framework that is used.

A model based approach for designing the communication between components that equally addresses syntactic aspects needed for code generation as well as semantic aspects to compute a holistic communication model between the components can be a solution to the problems above. It may as well reduce the manual development effort and error proneness of current methodologies. In an iterative process, from formal descriptions the holistic communication model is deduced which again can be used for automatic code generation.

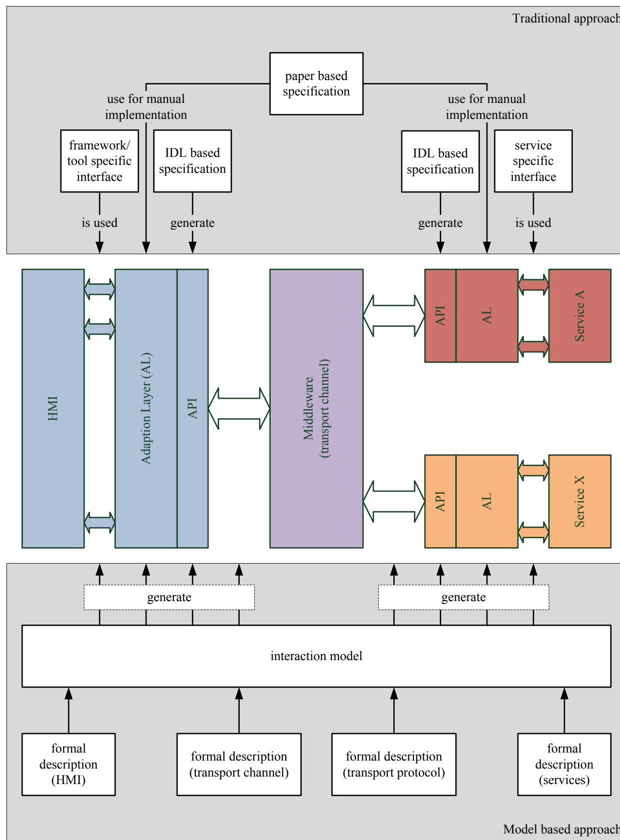

Figure 2: System architecture and development methods

Communication elements are described from a local point of view. A system contains several communication element types such as the interfaces of the HMI and the various connected services, a transport protocol, the transport channel, and additional actions (e.g. for data manipulations). The approach is briefly sketched in the lower part of figure 2.

The key to a reusable component is to minimize the mutual dependencies to other components. If described properly the in fact existing dependencies between communicating components can be automatically deduced and reflected in the composed model. Unfortunately, a composed model may not be deduced if the global dependencies are not described which is usually the case if elements are modeled from a local point of view. However, the leaks can be located and the missing links identified. That information enables the developer to enrich the description with the specific data. Once a complete model can be computed the glue code for each component can be generated. In the following sections the description and composition is described in greater detail.

In our approach we use Petri nets[Rei85] for behavioral description. Petri nets are a common tool for process modeling or modeling of concurrent behavior. They provide a well-founded formal semantics which is needed for automatic deduction. Communication between components is usually asynchronous because the components operate concurrently. On the other hand there exist explicit synchronization points to ensure data availability. Petri nets provide these features inherently. Several extensions have been made to Petri nets over the time that perfectly fit into this use-case: continuous time[Pen87], waiting queues[Bau89], transition priority, or the ability to be modeled hierarchically[Hub90]. These extensions allow us to model real, large scale communication systems.

For our purpose, code generation and semi-automatic model composition, we need a mechanism for accessing the real components and matching abstract transitions in the model to implemented methods in the system. On the model side, colored Petri nets (Petri nets with attributed tokens) [Jen97] are used. They allow tokens to carry data of a specific type and transitions to perform computations on that data. Colored Petri nets bring an own functional language that can be used for data computations as well as entry conditions. For expressiveness and performance reasons we do not use this language. Rather we use abstract data types and transition actions and associate them with manually implemented methods which are described using an IDL.

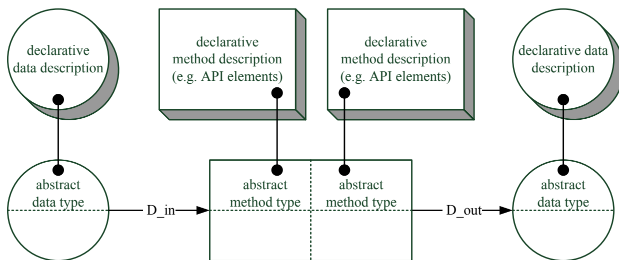

Figure 3: Binding of abstract model elements to the implementation

Locally describing communication elements is the first step towards obtaining the glue code of component interaction. Next, the loosely coupled elements in the collection have to be combined to an overall communication model. One additional reason for using Petri nets is that they are easily combinable without violating global correctness properties. [Zho07] proposes a composition algebra that we adopt for our purposes since it provides all required operations for our approach.

Unlike in [Zho07] we do not follow a goal driven composition algorithm. While communication tasks can be broken down to a similar hierarchical structure in our case the single composition steps require much more additional logic than performing basic composition operations. Pattern orientation has been proven to be more beneficial since it allows to explicitly model the additional logic as well as to break down complex tasks into simpler ones. This approach also best fits into a higher-level engineering process. Each development role has its own view of the elements it models. While describing their components they make use of other elements at their specific abstraction level and parameterization.

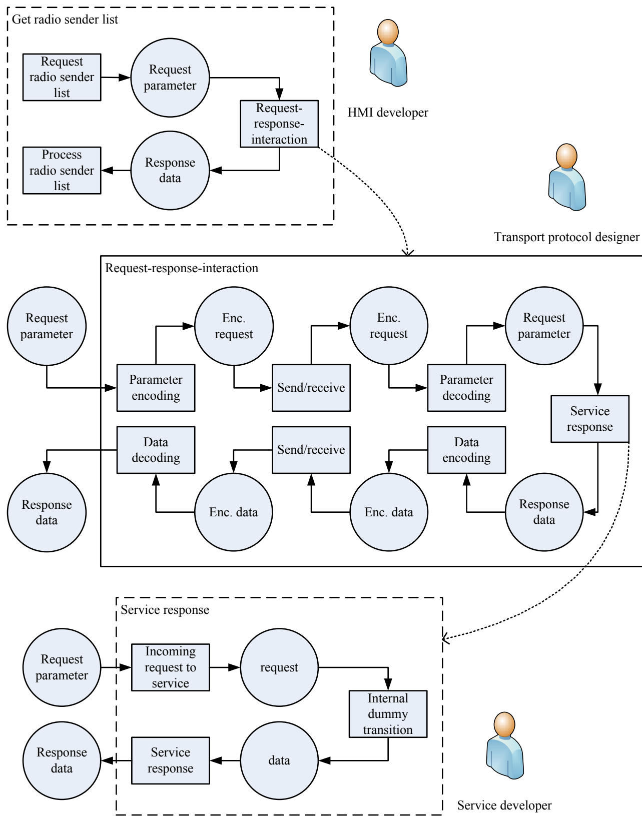

Figure 4: Role specific views and hierarchical design

In a top down approach the composition algorithm tries to find matching concrete elements for each abstract element used in components of higher abstraction. In detail, for each abstract transition a transition is identified that corresponds in terms of incoming and outgoing data, abstract action description and action specific parameters.

Under specific circumstances the search and binding within the model can be done fully automatically. Precisely, the algorithm requires the single components of the collection to be described based on a common understanding. There has to be a unique naming within the respective project for data types, content of data, and action behavior. There also has to be a defined set of parameters that are valid for an abstract action.

Since the systems are usually very different from each other, complete coverage of a common understanding of all occurring abstract elements cannot be achieved from the start of development. Therefore, in our approach the algorithm gives feedback to the developer in case an exact match was not found. In that case it provides a list of closest matches and the developer has to take action, either by implementing a new element, or adapting existing elements, or assuring that elements that are already contained in the collection have unique naming. That way, the complete communication model is

deduced to an increasing portion eventually leading to the complete system being automatically composed.

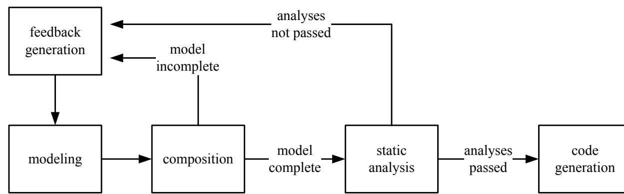

Figure 5: Iterative communication development process

Simultaneously, analyses of the model help improving the quality by identifying possibly error prone parts of the model or by pointing out optimization potential. This is another reason for the choice of Petri nets as modeling technique since analyses of important properties such as lifeness, boundedness, and reachability are already available and well-founded[Mur89]. Lifeness of certain transitions and reachability specific markings of the net are required in order to guarantee responsiveness of the system. If for example the transition representing the end point of a communication is not alive there may exist a dead lock resulting in a non-responsive system. Boundedness properties are used to decide whether the communication can get by with the restricted memory resources. Based on the results of lifeness and reachability analyses model optimization is possible as they allow a safe elimination of unreachable parts of the net provided they are not mandatory.

In the subsequent code generation phase there are three subtasks to perform: model splitting, transformations, and the generation of program code.

In modern systems components are split between several runtime structures, e.g. processes or tasks, which are not reflected by the composed communication model. The model thus has to be partitioned into different parts which are associated with their proper runtime structure. This step is necessary because code generation backends are runtime-structure-specific. Additionally, separated parts of the model define contexts that explicitly run concurrently which again restricts the freedom of optimizations that aim at minimizing concurrency. For example within the context of one component for three parallel transitions a single execution sequence may be stated as long as there are no contradictions, e.g. external triggers for the transitions.

The next step consists of transformations that are targeted at the specific execution environment. In resource constrained embedded systems, for performance reasons, it cannot be assumed that Petri nets can be executed directly. The execution model of Petri nets consists of a repeated sequence of identifying active transitions (transitions that may be executed under the current marking), choosing one active transition from the list of possible, executing the transition, and updating the marking of the net for the next iteration. Since the activation of a transition is dependent on the marking and additional, manual coded entry conditions are allowed, the activation of each transition in the net would have to be verified in each iteration resulting in a huge computation overhead.

Fortunately, bounded Petri nets can be transformed into equivalent state machines. Each possible marking of the net corresponds to a state in the state machine and transitions between states exist where corresponding markings can be transferred in the net by firing an active transition. On the one hand state machines can be executed more efficiently since identification of active transitions is performed at compile time, i.e. during the transformation process. On the other hand the resulting state machines are substantially larger. Eliminating concurrency, wherever it is not required in the net before transformation, greatly reduces the state space of the resulting state machine.

Finally, the glue code for further integration in the system is obtained by running a code generator that produces code fragments that fit into the specific target architecture. These backend code generators are adapted to and highly optimized for the concrete technique used to implement the target component in order to take full advantage of the features provided by the environment of the respective target component.

Besides benefits from directed developer guidance through the feedback system during the communication model composition process and the resulting saving of development time there are other advantages of the proposed approach. First the progressing diversion of specification and implementation can be reduced. The local descriptions of communication elements representing the specification of the interaction between the components are directly used for the automatic generation of the glue code. Provided tools performing transformations and code generation work correctly, only the small manually coded fragments, e.g. for realizing data manipulation, remain as a source for inconsistency.

Furthermore, many important correctness properties can be assured by formal proofs at design time because the modeling technique is based on a formal, well-founded, mathematical method. This way, the effort for traditional testing can be reduced while at the same time improving quality. For those cases where no formal analysis methods are available, the approach provides other ways of simplifying test routines. Since the glue code is generated automatically, it is easy to implement a design for test interface allowing test procedures to access system internals for enhanced error detection and localization.

In times of increasingly shorter development cycles special attention has to be given to the reusability of components. To a high degree, the presented approach is independent from project specific target architecture and implementation details. Exceptions are the backend code generator and the interface descriptions that are bound to project specific component implementations.

5 Conclusion

In this paper, we presented an approach for the specification and generation of the so-called "glue code" between the user interface, the middleware and the services. In our approach, software components are first specified as high-level models. The models

describe the different components of an infotainment system in a hardware-independent way, allowing for reuse or reification into more concrete models.

These models are then semi-automatically transformed into a model of the glue code needed for the interaction between the components. Finally, the glue code is generated by connecting code fragments to fit in the gap between components.

Acknowledgments

The research described in this paper was conducted within the project automotiveHMI. The project automotiveHMI is funded by the German Federal Ministry of Economics and Technology under grant number 01MS11002 resp. 01MS11007.

References

[aHMI] http://www.automotive-hmi.org/

[Alv11] Alves, A. de Castro. OSGi in Depth (1st ed.). Manning Publications Co., Greenwich, CT, USA, 2011.

[Bau89] Bause, F. and Beilner, H. Eine Modellwelt zur Integration von Warteschlangen- und Petri-Netz-Modellen. Messung, Modellierung und Bewertung von Rechensystemen, 5. GI/ITG-Fachtagung, Günther Stiege and J. S. Lie (Eds.). Springer-Verlag (1989), London, UK, 190-204.

[Hub90] Huber, P., Jensen, K., and Shapiro, R.M. 1991. Hierarchies in coloured Petri nets. Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets, Grzegorz Rozenberg (Ed.). Springer-Verlag (1990), London, UK, 313-341.

[Jen97] Jensen, K. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 3. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Kra88] Krasner, Glenn E. and Pope, Stephen T. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80 In J. Object Oriented Program., SIGS Publications(1988), 26-49

[Mur89] Murata, T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77, 4 (1989), 541-580.

[Pen87] Peng, D. and Shin K.G. Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control. IEEE Trans. Comput. 36, 4 (1987), 500-516.

[Pfa83] Pfaff, G.E.(editor) User Interface Management Systems In Proceedings of the Seeheim Workshop, Springer Verlag (1983)

[Rei85] Reisig, W. Petri Nets: an Introduction. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[Sie98] Siegel, J. OMG overview: CORBA and the OMA in enterprise computing. Communications of the ACM 41, 10 (1998), 37-43.

[Sch00] Schmidt, D. C.; Kuhns, F. An Overview of the Real-time CORBA Specification. The Computer Journal 33, 6 (2000), 56-63.

[Zho07] Zhovtobryukh, D. A Petri Net-based Approach for Automated Goal-Driven Web Service Composition. Simulation 83, 1 (2007), 33-63.