Model-based Middleware for Embedded Systems

Christian Salzmann, Martin Thiede

BMW Car IT GmbH
München, Germany
christian.salzmann@bmw-carit.de
martin.thiede@bmw-carit.de

Markus Völter

voelter – ingenieurbüro für softwaretechnologie
Heidenheim, Germany
voelter@acm.org

Abstract: In this paper we describe the advantages of a model-based approach to embedded component middleware. Component infrastructures such as Enterprise JavaBeans, Microsoft's COM+ and CORBA Components have become a de-facto standard for enterprise applications. Reasons for this success are the clean separation of technical and functional concerns, COTS containers (applications servers), and the resulting well-defined programming model and standardization. To benefit from these advantages in the domain of embedded systems, the same concepts can be used, but a different implementation strategy is required. First we describe the characteristics of automotive software and explain why the implementation strategies used in enterprise systems can not simply be applied to the automotive domain. Then we present a brief outline of the design and implementation of a model-based embedded component middleware.

1 Introduction

Why is software development for embedded Electronic Control Units (ECU) so much more complicated than for example developing software for a PDA? Why do we have a factor of 50 concerning development time and cost? One reason is that the software in the automotive domain is more or less developed from scratch each time around. There is very little reuse compared to desktop IT. But why is this? What differentiates automotive software and the way it has to be developed from other domains, such as business software or aerospace? In order to answer these questions we first outline some of the characteristics of automotive software that lead us to proposing a different approach for automotive middleware implementation. The different kinds of characteristics of automotive software, which include technical, organizational as well as economical issues, also serve to illustrate the broad variety of challenges found.

Heterogeneity: Automotive software is very diverse ranging from entertainment and office-like software to safety-critical real-time control software. We cluster automotive software into three groups: *Infotainment & Comfort Software* (including system services with soft real-time requirements), *Safety Critical Software* and *(hard) Real-time Software.*

Emphasis on Software Integration: As a result of the collaborative and distributed development process, OEMs have to integrate independently developed systems or software. While this is always a difficult task the situation for automotive software engineering is even worse, because suppliers usually have a lot of freedom in how they realize the required functionality. (In business IT the client would often strongly constrain the technologies that may be used by a supplier to facilitate integration and maintenance.) Furthermore, the OEM usually only has black-box specifications of the subsystems to be integrated which makes successful testing and error localization more difficult. Besides that, the distributed nature of the development effort causes difficulties. OEMs often experience problems when they try to modify parts of the subsystems in order to localize errors.

Unit based cost structure & resource optimization: The unit based cost structure is primarily a business characteristic. For automotive software engineering it is problematic, because it drives system engineers to design hardware that is as cheap as possible for the required functionality. This usually means that performance and memory is severely limited. Thus, the software engineer is confronted with hardware that is barely sufficient for the functionality to be implemented requiring a highly optimized implementation. In particular this means that abstraction layers are sacrificed in order to achieve the required performance, and state-of-the-art development methodologies and programming languages often cannot be used. This of course makes developing complex software that is correct and delivered in time a very difficult task. In the recent past important automotive system development projects were late because of software problems.

An important part of software engineering is structuring a system in a way that helps to cope with the inherent complexity, making it easier to handle, cheaper to produce and faster to develop and adapt. In business IT component middleware has proven to be useful in this context. So the goal must be to make this approach applicable to the automotive domain. This requires an adaptation of the implementation strategy to fit the constraints of automotive software development.

2 Components & Middleware

Component infrastructures such as Enterprise JavaBeans, Microsoft's COM+ and CORBA Components [OMGb] have become a de-facto standard for enterprise applications. Reasons for their success are the clean separation of technical and functional concerns, COTS containers (applications servers), and the well-defined programming model. To benefit from these advantages in the domain of automotive embedded systems, the same basic concepts can be used, but a different implementation strategy is required: monolithic application servers are not suitable because of the limited resources regarding computing power, memory, etc. on the device. Instead, the container needs to be customized exactly to the needs of the ECU and the application. Model-based code generation is an efficient means to do this.

2.1 Our understanding of Components

Our definition of components and component infrastructures is based on the *Server Component Patterns* book [VSW02]. The following paragraph summarizes the essential building blocks.

A *component* encapsulates a well-defined piece of the overall application functionality. Component instances execute in a *container* which handles technical, typical cross-cutting concerns for the components.

A system is assembled from collaborating components as well as one or more containers. Components access each other through a well-defined *component interface*. Components can be reused in several applications. Since the functionality of a component and the way to access it is well-defined and self-contained, the preexisting interface is technically separate from the *component implementation* which can be exchanged without affecting clients.

The strict separation of interface and implementation allows the container to insert *component proxies* into the call chain between the clients and the implementation. On behalf of the container, these proxies handle technical concerns. The *lifecycle callback interface* of a component is used by the container to control the lifecycle of a component instance. This includes instantiating components, configuring instances, activating and passivating them over time, checking their state (running, standby, overload, error, …) or restarting one in case of severe problems. Because all components are required to have the same lifecycle interface, the container can handle different types of components uniformly. *Annotations* are used by the component developer to declaratively specify technical concerns (i.e. which of a container's services are needed by a component and in which way). A *component context* is an interface passed to the component implementation that allows it to control some aspects of the container (e.g. report an error and request shutdown).

A component is not allowed to manage its own resources. It has to request access to *managed resources* from the container, allowing the container to efficiently manage resources for the whole application (i.e. several components). These resources also include access to other component interfaces (*required interfaces*). All the resource a component instances wants to use at runtime must be declared in the annotations to allow the container to determine if a component can correctly run in given context, and prepare accordingly.

When operations are invoked on instances, the invocation might carry an additional *invocation context* that contains, in addition to operation name and parameters, data structures which the container can use to handle the technical concerns (such as a security token). Last but not least, a component is not just dropped into a container; it has to be explicitly installed in it, allowing the container to decide (based on the annotations and required resources) if it can host the component in a given environment.

2.2 Benefits & Drawbacks

The approach outlined above has the following benefits:

- *Portability:* Components are developed against the interfaces of the container, the container can adapt this to different environments (such as operating systems, databases or transaction monitors in the enterprise world).
- *Container-based Optimization:* Within the boundaries specified by the specifications of the container and the lifecycle interface, the container is free to optimize different aspects of the application.
- *Standardized, Simplified Programming Model:* Because the environment in which components execute is well-defined, and because the developer does not need to deal with low-level implementation details of the technical concerns, the programming model for application developers is simplified and consistent over the family of applications implemented for the same container.
- *Clearly defined developer roles:* Because application developers can focus on their specific application requirements, and because infrastructure experts deal with the implementation of the container, both aspects can be implemented by people who are experts on their respective field, improving the quality of the software.

On the other hand, traditional implementations of component infrastructures also have a couple of drawbacks:

- *Performance Overhead:* Because requests are intercepted by the container, and because its services are implemented generically to be reusable, performance of component-based applications is impacted.
- *Loss of control:* Some people feel that handing over control over technical aspects to the container limits their control over what is actually happening. While this is true, in most scenarios this is not a liability, however, because the container can handle most of these aspects better and more reliably than code handcrafted by the average developer.
- *Large and heavy:* Most of today's implementations are large and heavy software monsters. Installing, configuring or (re-) starting them can take a while.
- *Complexity:* Of course, by providing a reusable solution to a recurring problem, component infrastructures imply a lot of accidental complexity. This might be a problem for safety-critical applications.

These drawbacks prevent a traditional approach from being used in the automotive domain. In order to remedy these drawbacks we propose using model-based software development.

2.3 Concepts of Model-based Software Development

Model-based Software Development (MDSD, see [OMGa, Sa02]) aims at automatically constructing software programs from domain specific, abstract models. The “intent” of the application developer is captured in specifications (or models) that consist of concepts related to the problem domain; they are thus based on a domain-specific language (DSL).

A generator then reads the specification/model and verifies the model against the domain meta model available to the generator. In a second step, source code for the respective runtime platform is generated. It is important to understand that the focus of MDSD is not to generate dumb class skeletons from UML class diagrams. Rather, the generator automatically creates all of the infrastructure code needed to run a piece of pure application logic on a certain runtime platform. Additionally, it is responsible for realizing domain-specific optimizations for the respective platform.

2.4 Benefits of Using MDSD to Develop Component Middleware

Generated containers implemented using model-based software development techniques can even improve some of the benefits while reducing most of the drawbacks of component middleware as described above:

- Optimizations can be implemented directly in the generated code. Since domain-specific models are more expressive than code, more domain-specific optimizations are possible.
- The programming model can be simplified even further, since generated code and the development process based on modelling guides developers when implementing application logic. Using code generation in combination with a traditional compiler (for a language such as C) even allows to *enforce* some aspects of the programming model, violations of which could otherwise not be prevented.
- The tradeoff between footprint and performance overhead can be adjusted over a wide range since the generator is free to implement features statically (faster, but typically more footprint) and dynamically (usually slower, but smaller).
- By adding only those features to a container that are actually necessary for a given scenario, the overall footprint and performance overhead of a system can be significantly reduced.

3 Our Approach to Model-based Middleware Development

The authors are developing a model-based approach to embedded component middleware that satisfies the above requirements. The following illustration shows the approach in a nutshell.

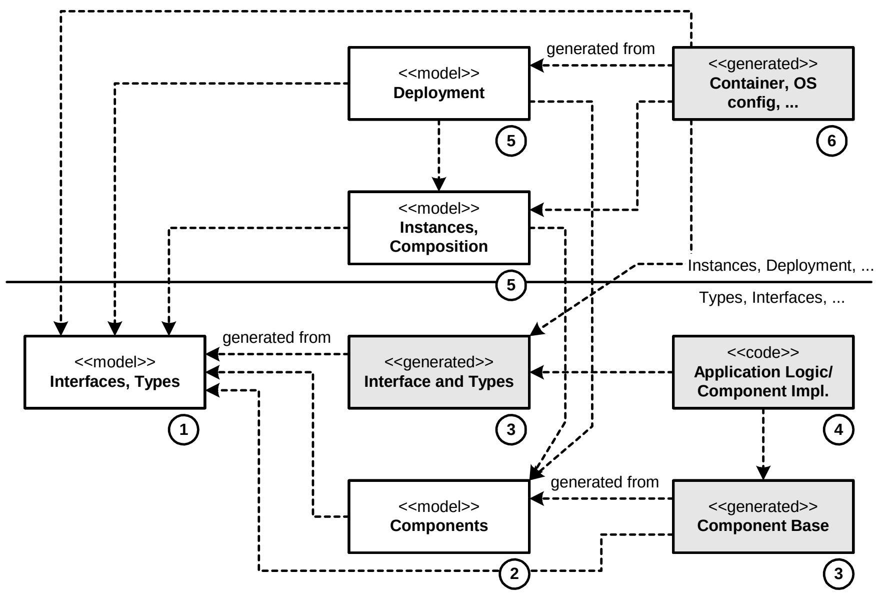

Illustration 1: The proposed solution in a nutshell

In the first step, we define interfaces and complex types. Since an interface can be used by various components, it is crucial to define them first. In a second step, we define components and their communication ports. Here, we also define some of the communication parameters; for example, whether a communication in a port should happen synchronously or asynchronously. Since this affects the API against which the implementation code is developed, these definitions have to be made before the application logic is implemented (manually) in step four. Before doing this, we generate these interface APIs as well as component base code (for example, base classes in OO systems), see step 3. This concludes component definition. In step five we define which component instances we will use, how their ports will be connected; and on which hardware devices these instances will be deployed, as well as other system constraints. All these models together will then be used by the second generation step that creates all the infrastructure code: the container, the communication implementations, the OS configuration files, build scripts, etc. In a final step (not shown) all the generated code will be compiled and linked using the generated build script, resulting in the final system.

The models mentioned above are joined. The resulting overall model is then forwarded to the generator tool which works as shown in the following illustration.

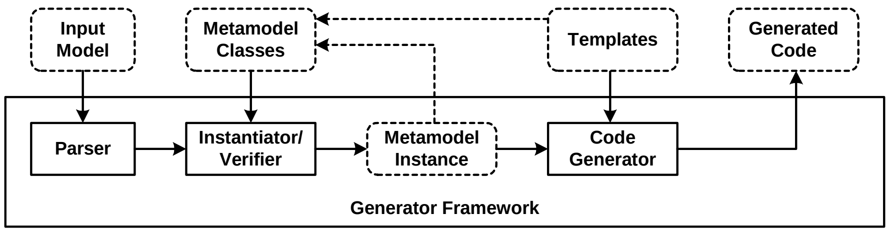

Illustration 2: Generator tool workflow

The overall model is read by a *parser*. Then, the model is *instantiated* and *verified* against the *metamodel*; the corresponding part of the generator acts as a configuration validator (or buildability checker) that checks if the container will be able to work correctly (as far as this is possible at this early stage). If the model is correct, the last stage uses *code generation templates* to generate container implementation code as well as any other artefacts that are required to build the system and run it on the target ECU. All this is supplied to a normal programming language compiler/linker/make tool for the respective target. Optionally, it can be analysed statically to verify its correctness (as far as such static code analysis is feasible – this is no different from static analysis of hand-written code).

3.1 Technical Concerns: Container Features

The selection of what constitutes the technical concerns in a particular family of applications depends on the specific requirements of the domain. Unlike in enterprise systems, where all applications typically consist of some database/transaction related logic, embedded systems are more diverse and it is thus not feasible to decide once and for all on what constitutes the technical concerns. This is the reason why we do not propose one specific embedded container in this paper, but rather an approach, or an architecture, to construct such containers for a specific software system family.

However, there are some candidate aspects that lend themselves to being implemented as container features.

- *Scheduling:* Controlling of thread and task priorities, creation and maintenance of thread pools, deadlock detection
- *Interrupt Handling:* A high-level notification interface in case interrupts occur can be provided by the container. It can invoke previously configured operations on component instances.
- *Simple Event Propagation:* The propagation of events from one component to another, synchronously or asynchronously can be supported by the container.
- *Timer:* Time-based events can be triggered by the container
- *Remote Communication:* In case communicating components reside in different containers on different devices (boards, controllers, computers, …) in a

distributed system, the container can take care of remoting (using CORBA [OMGb], CAN 0, etc.)

- *Generic Driver Interface:* Accessing low-level drivers can be simplified. The container can provide more abstract, higher level access to lower lever drivers, or convert data structures depending on the device. In addition, it can control concurrent access to shared hardware devices from several components.
- *Lifecycle Control:* In case determinism is not absolutely important for a system, the container can control the lifecycle of component instances. This includes restarting instances in case errors are detected, lazily instantiating instances only when they are actually needed, passivating instances when they have not been used for a specified period of time, etc.
- *Resource Control:* The container can control resources, or manage pooling of critical resources. Also, it can enforce quota allocated to component instances (e.g. memory or disk quota) and coordinate concurrent access to shared resources.
- *Advanced Error Detection:* In embedded systems with more or less strict timing constraints, many errors are an indirect consequence of a (seemingly unrelated) timing problem or illegal invocation sequences of component operations. If interfaces are annotated with state charts that include timing constraints, a container can be generated that controls these timing and state constraints and report the root-cause. In general, the concept of “programming by contract” can be used effectively because the container can contain code to actually check pre- and post conditions, as well as invariants, specified for interfaces of the components.
- *Management Facade:* The container can provide a coherent, homogeneous external interface for the management of embedded applications. For example, it can provide an SNMP MIB or issue SNMP traps for the container itself and the components inside. Management of devices will be simplified, and application programmers need not bother with specifics of management protocols such as SNMP.

3.2 Interface specifications

Interfaces play a central role in component infrastructures. Interfaces define contracts between components, as well as between the container and the components it hosts. In traditional systems, interfaces are typically defined as a set of operations including typed arguments, as well as a return type. In order to achieve a more reliable system composition, more details must be given on interfaces. These are all specified in the respective models and include

- services required from the container to allow the component to run
- other component’s interfaces required by a component
- timing constraints regarding interface operations

- pre- and post conditions for operations, or a state machine that defines legal invocation sequences
- data published by a component, or data consumed (required) by a component

Interface definitions as outlined above are logical definitions of what a component provides or requires. It does not say anything about how these interfaces are implemented. The realization of the interfaces can be supported by the generated container. For example

- operations can be called directly if the caller and the callee are collocated in the same process, or can include proxies and some kind of remoting infrastructure (e.g. over CAN) for remote calls.
- published or required data items can be stored to/retrieved from a shared memory area or it can be put on/taken from a CAN bus.
- timing constraints or pre/post conditions can be checked by the container and errors can be reported

3.3 Applicability of the solution

Considering the different architectures for embedded systems the question is: in which architecture can the proposed approach be used sensibly? Let's look at each of these architectures in turn.

- *No operating system:* In these very small systems, the proposed architecture is very suitable. First of all, software on these devices typically is very static, not featuring dynamic aspects. Efficiency and small code size is important, while we still need some flexibility regarding different hardware platforms/devices (because there is no OS). Also, because there is no OS, there is a lot of use for reusable, cross-cutting technical concerns handling of the container. The container thus serves as an efficient implementation of an abstraction layer – providing flexibility while still being efficient.
- *With (real time) operating system:* real time operating systems (as any operating system) typically provide APIs on a very low level. Also, there is no handling of domain- (or software system familiy-) specific technical concerns. Containers can provide this higher-level abstractions. The container can also serve as a means of integrating different tools, systems, middlewares, etc. For example, the container can provide remoting based on CAN or Flexray.

4 Conclusion: Embedded Middleware – more than reuse

In this paper we sketched the basic concepts of a model-based embedded component middleware for automotive systems. We showed that a middleware-based approach reduces complexity of the systems which makes software cheaper to produce, faster to develop and more flexible to adapt. Traditional middleware approaches such as those used in enterprise systems are not applicable in the automotive domain, due to unit based cost structures and resource constraints. Here the approach of model-based software development is a promising approach.

However, once a middleware technology has been established, more use cases than those outlined here might arise. Thus we present two application scenarios for automotive middleware.

- **Communication based specifications**: In the automotive world functional networks [Be01] are used to specify logical system behaviour independent of software modules, ECUs and busses. A model-based middleware can respect the constraints given in the functional networks and generate a system that enforces conformance to the specified functional network [SS03]. This results in important advantages concerning safety and issues such as feature interaction [Za99].

- **Handling of cross-cutting concerns**: The generated middleware might provide ECU-wide services, such as lifecycle management, diagnostics or safety. Each software component may use these services without caring about how they are delivered.

5 Related Work

The AUTOSAR standard [AS] aims at standardizing a communication middleware and reference architecture for automotive ECUs. While it does not prescribe a specific implementation strategy, the requirements stated in the standard suggest an approach in the spirit of what has been described above.

Code generation in general has been used in the embedded systems domain for a long time, however focussing chiefly on the efficient implementation of “application logic” from flow charts or state diagrams (using tools such as [Ma]). The automatic generation of infrastructure code, i.e. a component middleware is not a widely used approach, though.

Traditional middleware such as CORBA (or the CORBA Component Model) cannot directly be used in the embedded domain. While there are versions for resource-limited and real time systems (MinimumCORBA and RTCORBA, respectively [OMGb, OI01]) these are not useful for really small and optimized systems; also, transport implementations for bus systems found in the automotive domain such as CAN, Flexray or MOST are not readily available.

6 ACKNOWLEDGEMENTS

Many of the concepts described above were made concrete by the embedded middleware team at BMW Car IT, namely Paul Hoser and Michael Rudorfer.

REFERENCES

[AS] AUTOSAR GbR: Automotive Open System Architecture, http://www.autosar.org

[Be01] von der Beeck; Braun; Rappl; Schröder: Modellbasierte Softwareentwicklung für automobilspezifische Steuergerätenetzwerke, VDI Tagung Elektronik im KFZ, BadenBaden, VDI Berichte Nr. 1646, 2001

[CiA] CiA: Controller Area Network (CAN), an overview, http://www.can-cia.de/can/

[Ma] Mathworks: Matlab / Simulink, http://www.mathworks.com/products/tech_computing

[OI01] Objective Interface: Realtime and Embedded CORBA discussion forum, http://www.realtime-corba.com/

[OMGa] OMG: Model-Driven Architecture, http://www.omg.org/mda

[OMGb] OMG: Minimum CORBA Specification, http://doc.ece.uci.edu/CORBA/formal/02-08-01.pdf

[Sa02] Salzmann, C.: Modellbasierter Entwurf spontaner Komponentensysteme, PhD Thesis Munich University of Technology, 2002.

[SS03] Salzmann; Schätz: Service-Based Systems Engineering: Consistent Combination of Services In: Proceedings of ICFEM 2003, Fifth International Conference on Formal Engineering Methods. Springer LNCS 2885, 2003

[VSW02] Voelter; Schmid; Wolff: Server Component Patterns - Component Infrastructures illustrated with EJB, Wiley, 2002

[Za99] Zave, P.: Systematic Design of Call Coverage Features technical Report AT&T Labs 1999.