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? 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. Among them are Heterogeneity, Emphasis on Software Integration, and Unit based cost structure and resource optimization.

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, Container-based Optimization, Standardized, Simplified Programming Model, and Clearly defined developer roles. On the other hand, traditional implementations of component infrastructures also have a couple of drawbacks: Performance Overhead, Loss of control, Large and heavy, Complexity.

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.

2.5 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.

3 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.

4 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.

5 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.