A Model-Based Development Method for Device Drivers

Michael Kersten	Ulrich Margull	Nikolaus Regnat
Siemens AG	*1 mal 1 Software GmbH*	*Siemens AG*
Otto-Hahn-Ring 6	Maxstr. 31	Otto-Hahn-Ring 6
D-81739 München	D-90762 Fürth	D-81739 München

Abstract: The present contribution deals with the experiences of introducing a model-based software development method for device driver families used in electronic control units in the automotive domain (e.g. engine management systems). Besides the technical experiences with UML, SysML and the corresponding tools for requirements, modeling and code generation we will discuss the key benefits as well as the crucial factors for success.

1 Introduction

The major motivation for introducing a model-based software development method at Continental Automotive GmbH was to increase quality and productivity. In order to achieve maximum synergy it was decided to cover all phases of the software development v-cycle including requirements engineering, design, implementation and testing. The details of all phases will be discussed in the corresponding sections.

The better intelligibility, increased consistency and standardization of models compared with text documents improve the quality of all development artifacts. A major contribution to increased productivity was the introduction of a code generator (UML to MISRA C). Thereby the main challenge in customization is to meet the strict automotive coding requirements. In addition a dedicated document generator enables the generation of documents for all models of the v-cycle. A further positive aspect of the approach is that it contributes to reaching CMMI level 3.

2 Requirements Modeling and Specification

State of the art in the development of complex device driver is the usage of a dedicated requirements engineering tool (e.g. DOORS [Tel08]) to capture, analyze and manage the requirements. Once the requirements engineering activities are finished the requirements are transfered to a specification model using the tool Reqtify [GEE08]. In the specification model they are traced using ≪trace≫ dependencies based on the SysML standard. These dependencies allow to answer typical questions like ”are all requirements covered?” or ”what’s the impact of changing this requirement?”.

The specification model bridges the gap between the pure requirements engineering and a model based development method and provides a request for all solutions based on the requirements. The model is developed using the UML tool ARTiSAN Studio [ART08] and consists of several views. The *specification context view* shows the device driver as a black box and the interaction with the external environment in a very abstract form by using class diagrams and classes with stereotypes ≪component> and ≪hardware≫. The functional requirements of the device driver are represented in the *use case view*.

One important part of the specification model is the *variability view* which allows to model product families. Normally, a (generic) device driver is built to support a whole range of functionalities, while in a specific electronic control unit only a subset of the complete functionality is needed. This variability is modeled as a tree with the device driver (with stereotype ≪component≫) as the root element. A feature is either connected to the component or to another feature. The connection is modeled using the composite association, expressing that the owning element is "composed" of the (sub-)features. This feature model was inspired by [Gom05].

The tree is read from the root: the component has ("is composed of") one or more features. Each feature itself may be composed of other sub-features. The multiplicity defines the type of variability (mandatory, optional, alternative). A multiplicity of one (1) expresses that the (sub-)feature must be chosen if the owning feature is chosen. If the owning part is the component itself, the feature must always be chosen (mandatory feature). A multiplicity of zero or one (0..1) expresses that the feature need not necessarily be chosen when the owning feature is chosen (optional feature). A set of features that are an alternative to each other are modeled with multiplicity zero or one (0..1) and an additional $\{xor\}$ constraint.

Figure 1 shows a typical variability diagram. The central element is the ADC (Analog to Digital Converter) device driver component. There's one mandatory feature "normal conversion". In addition there are two features that exclude each other and only one may be present at a time (alternative features): "SW triggered conversion" and "HW triggered conversion". The features "burst conversion" and "edge triggered conversion" are both optional, however the latter one can only be chosen if "HW triggered conversion" is also selected.

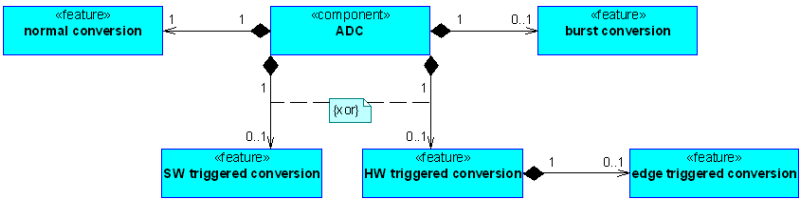

Figure 1: Variability diagram example

Additionally it was decided to include an interface view in the specification model to enforce a stable hardware abstraction to upper software layers. This is necessary due to special constraints in the development of electronic control units where different μC families are used (e.g. TriCore, MPC55XX ...). The device drivers are strongly dependent on the hardware but the applicative (upper layer) software should be independent. Therefore

stable interfaces across all supported platforms are needed.

3 Design

Next step in the development method is the design. Creating a good design is still intellectual work and automatic model to model transformation between specification and design is not possible. However by sharing (importing) the specification model into the design model the developer is able to re-use model elements (e.g. interfaces, data types ...). This relieves the developer of cumbersome work and provides consistency between models. Elements are traced between the design and specification model by using SysML based ≪satisfy≫ dependencies. These dependencies allow to answer important questions like "what's the concept to support this feature?" or "what behavior does support this use case?".

The goal of the design model is to document all design decisions of the driver in an abstract, conceptual way. The design is not intended for code generation and therefore does not focus on details, instead the important structural and behavioral decisions are described. An object-oriented design is recommended for modeling, however, a structural approach can also be used.

Similar to the specification model the design model contains several views. The *design context view* describes the interaction of the device driver with external environment: interaction with other components, processor peripherals or external hardware by using stereotyped classes. In the *static structure view*, the software partitioning shows the break-down of the device driver into sub-parts, and the interfaces and features each part realizes. The features are directly linked to the specification, and the different types (e.g. mandatory, optional and alternative) are realized correspondingly. In addition first sketches of important data structures are shown, e.g. for describing the configuration or the underlying hardware. The *dynamic view* shows the behavior of the device driver e.g. by using activity diagrams for algorithms and state diagrams for state machines. Also the mapping of the software to tasks and interrupts is shown.

4 Implementation (MISRA-C conform)

The basis for the implementation model is a finished design. Both the design and specification models are imported using the sharing technique and allow re-use of model elements e.g. interfaces. In addition model elements of other drivers are imported and re-used. A model to model transformation from an abstract design to an implementation model would support the developer but was too elaborate to implement. Tracing between design and implementation is supported by SysML based ≪realize≫ dependencies. The generated code is traced automatically to the implementation model by the code generators marker lines. As the tracing is now established from requirements to code level it is possible to answer questions like "how is this design part implemented?" or "what's the impact on changing

this requirement in our source code?".

The implementation model follows an MDA [K$^+$03] approach by dividing the model into μC independent and dependent views. This supports re-use of parts of the model especially when dealing with different μC derivates. The implementation language is C with compliance to the MISRA-C rule set [MIS08]. Since C has no object-oriented capabilities, mapping of the (object-oriented) UML model elements to MISRA-C is far from being trivial. Additionally, not all design constructs can reasonably be implemented in C, and therefore some design restrictions apply. For example, polymorphic behavior (virtual functions) or multiple inheritance is not allowed. The used UML elements like classes, attributes and operations are enriched with stereotypes dedicated for code generation e.g. an operation can be generated as an C function (external or static), or alternatively as an inline function or even as a macro function.

A customized C code generator has been developed that fulfills the special requirements for MISRA-C and Continental Automotive GmbH coding guidelines, as well as for device driver development in general (highly optimized code, small resource consumption, closely tied to underlying hardware, etc.). Code generation is currently only used for C files, e.g. not for assembler files. From the implementation model all structural C code is generated, including all files (normally two files per class: *.h and *.c), variables, data types, structures and the function declarations. The function bodies are not generated but coded manually. The generation is one-way (no round-trip engineering) but existing function bodies are kept during generation.

The main design construct is the class. In C, it can be implemented either using a set of (global) data and functions operating on the data. Alternatively, a class can be represented using a `struct`, and member functions that have a pointer as first parameter (resembling the C++ `this` pointer). Since all C functions reside in a global namespace, a naming rule is used to discriminate between the member functions of different classes: `ADC_GetResult()`, where `ADC` is the class name and `GetResult()` the member operation.

Memory allocation is an important topic for embedded systems. Within the automotive domain this becomes even more critical as resource optimization creates competitive advantages due to the high production volumes. In order to support the different μC families abstraction of the underlying hardware and used compilers is needed. The implementation model support this MDA approach by using a set of memory allocation stereotypes. The code generator then creates all corresponding memory allocation directives in the C code according to the needs of μC and compiler standards.

Behavioral aspects are currently not generated as most of the device drivers only use simple algorithms like register accesses to deal with the underlying hardware. However especially communication drivers (e.g. CAN, FlexRay ...) do use complex state machines internally and investigation is ongoing to extend the code generator to support generation of state diagrams.

5 Model Based Testing

The next step within the v-cycle is the module test. Using the sharing technique all previously created models are imported into the test models. The nature of test models allow a very high re-use as almost all elements, from specification to implementation are needed when describing test cases. In addition test cases are traced to specification, design and implementation model elements using SysML based ≪verify≫ dependencies.

Our decision to use the UML Testing Profile (UML2TP, see [S$^+$05]) is motivated by its inherent standard conformance enabled by the UML2TP to TTCN/3 mapping. Another strong argument is the relatively compact and straight-forward specification compared to other OMG profile specifications. Test contexts are modeled as stereotyped classes and composite structure diagrams show how the system under test is connected to test components. In the UML2TP a text context directly contains test cases, however we found it necessary to extend this decomposition and provide a possibility to group test cases not only by a test context. By using the concept of nested classes and introducing the stereotype ≪TestCaseGroup≫ this has been made possible. Test cases themselves are stereotyped operations and contain not only a textual description but a defined behavior that is modeled using sequence diagrams. A dedicated code generator is currently under development that enables generation of test cases (including the behavioral part) for the test automation tool Rational Test Realtime (see [IBM08]).

6 Success factors and main challenges

Good tooling is crucial for a successful introduction of model-based development. Our experience shows that developers are interested in new technologies if it has advantages for their daily work. For example, generating all documentation from the models is an important advantage which relieves the developers from doing manual documentation. At the same time, the tooling must be efficient and well-suited to their work flow, otherwise they will see the tool *and* the model-based development as a hindrance, thus trying to avoid it. Connection to the company's CM tool is also important. Storing models in CM is more difficult than storing files since a model requires a more stringent consistency across its parts. Model diff and merge is needed in daily work, too.

Sharing is another important issue. Parts of models need to be reused in other models, e.g. the exported interfaces of one device driver model need to be imported into another model. For sharing, a specific tool add-on was developed that allows to share packages easily, while at the same time keeping the consistency between the models.

Model-based software development is strongly influenced by the international collaborative development of today's global companies. The major idea here is not only to realize cost advantages but to combine different cultural ways of thinking to achieve better results. However, distributed development of complex products is non-trivial. Even for people with the same cultural background the discussion of requirement, design, test and implementation issues via phone conference is not easy. The usage of modeling techniques

helps to gain a common understanding of artifacts under development. To achieve this, it is necessary that all collaborating persons must have a consistent view on the corresponding models. Following an ARTiSAN recommendation, we use a central terminal server which also eases tool administration. With a time-shift of 5,5 hours from Toulouse to Bangalore the approach works quite well since there is enough overlap for phone conferences and also enough idle time for backups and maintenance during the night. A new challenge will be the integration of the location in USA where multiple terminal servers and database replication have to be taken into account. We are in discussion with different tool vendors on these issues but it will need a couple of years until solutions that can be used from scratch are available.

However the major success factor of the improvement project is a strong management commitment over the whole run-time of four years. This includes that all developers participate in a training before starting productive work. The management also set strong targets on the migration of existing drivers to the model-based method. For newly developed device driver it is mandatory.

7 Conclusion

During the writing of the present contribution the method was rolled out up to the implementation phase while the roll out of to the test phases was still ongoing. The method is used productively in the collaborative development for device drivers at the locations Regensburg, Toulouse, Timisoara, Iasi and Bangalore with great success. At the moment 70% of all drivers are developed using the model based methodology. It is planned that up to the year of 2009 90% of the driver development is done model based.

References

[ART08] ARTiSAN, 2008. www.artisansw.com.

[GEE08] GEENSYS, 2008. www.geensys.com.

[Gom05] Hassan Gomaa. *Designing Software Product Lines with UML.* Addison-Wesley, 2005.

[IBM08] IBM, 2008. www.ibm.com/software/awdtools/test/realtime.

[K+03] Anneke Kleppe et al. *MDA Explained.* Addison-Wesley, 2003.

[MIS08] MISRA, 2008. www.misra.org.uk.

[S+05] Ina Schieferdecker et al. UML Testing Profile. Technical report, OMG, www.omg.org, July 2005. Version 1.0 formal/05-07-07.

[Tel08] Telelogic, 2008. www.telelogic.com/doors.