Model Level Coupling of Heterogeneous Embedded Systems

Clemens Reichmann[1], Daniel Gebauer[2], and Klaus D. Müller-Glaser[1]

[1] ITIV, University of Karlsruhe, 76128 Karlsruhe, Germany
{reichmann, mueller-glaser}@itiv.uni-karlsruhe.de
[2] FZI, Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
gebauer@fzi.de

Abstract. The integration platform GeneralStore is a tool that assists an integrated development process starting with a model and ending with executable code. The software features coupling of subsystems from different modeling domains on model level. From the coupled model it generates a running prototype by code generation. In addition to object-oriented system modeling for software components in embedded systems, it supports time-discrete and time-continuous modeling concepts. The Unified Modeling Language (UML) metamodel is used for storing CASE data in a Meta Object Facility (MOF) object repository whereas XML Metadata Interchange format (XMI) is used to interchange this data with UML-CASE-tools. The CASE-tool chain we present in this paper further supports concurrent engineering including versioning and configuration management. It provides adaptors for the tools MATLAB/Simulink/Stateflow and various UML CASE tools such as MagicDraw, as well as an importer and exporter of UML/XMI. Utilizing the UML notation for an overall system design cycle, the focus of this paper lies on the coupling of heterogeneous subsystems and on a new code generation and coupling approach.

1 Introduction

Today's developments of embedded electronic systems are driven by a rapidly increasing complexity. A common medium-class car today contains 15 different electronic control units (ECU's); in luxury-class cars more than 70 different units work together.

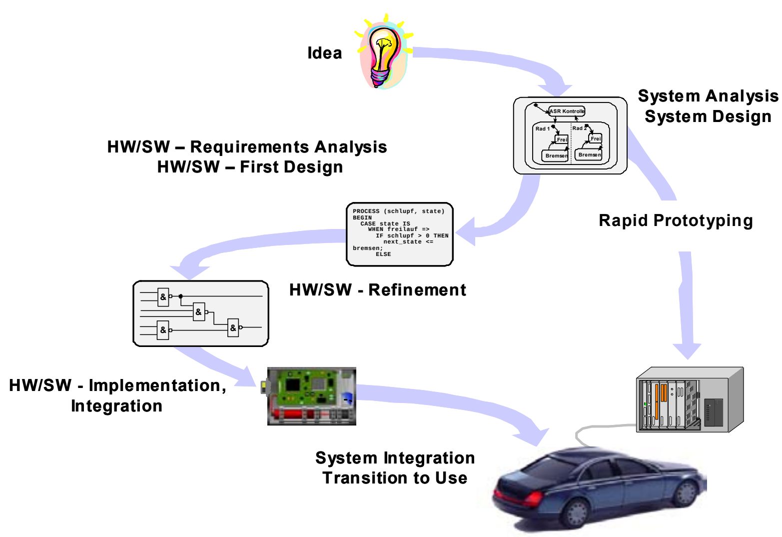

Fig. 1. Concept-oriented rapid-prototyping

This ever-increasing number of embedded systems is not the only factor challenging the development. Also the tasks they handle are manifold, ranging from classical control devices to high-end multimedia applications, automation, and biomedical engineering.

With increasing complexity one can observe a shift from electronic and pure control based systems towards software-based systems. The object-oriented paradigm is becoming more and more accepted in the domain of embedded software development and is supported by a growing number of CASE-tools for UML-based modeling [11]. Additionally, embedded software is not only characterized by pure software design problems but also event-driven subsystems. Often they control subsystems in the time-continuous domain. Thus, this need for heterogeneous system descriptions increases their complexity and error-proneness even more.

Contrary to the steep increase in complexity is the requirement of shorter product cycles while keeping safety and quality of the product at least the same level. One possible answer to this challenge is concept-oriented rapid-prototyping [1][2] as shown in Figure 1, which allows direct validation of system designs and algorithms in a real world environment.

Methods and tools are needed for such approaches, which allow modeling of heterogeneous systems. Each typically consists of different modeling domains. This approach must also provide a transparent way to model the interfaces between various notations.

For rapid validation of the model one has to be able to automatically generate a running prototype out of the model, which includes the various sub-models and their interfaces.

We address the above-mentioned challenges with a design approach as described in this paper. After evaluating the existing work in Section 2 we introduce the concept of metamodeling and describe a solution for model coupling of different notations and design concepts in Section 4. Our approach implies certain constraints on the development process that are discussed in Section 5. In Section 6 we introduce our concept to reflect the model coupling in the generated compilable source code. Obviously this concept can only be successful if there is a tool supporting it. Therefore, our CASE-tool integration platform *GeneralStore* is presented in Section 7. Finally, this paper will finish with a discussion of the results, a conclusion, and an outlook of future work.

2 Related Work

Commercial solutions that support object-oriented modeling of embedded electronic systems are rare. However a wide range of CASE-tools for object-oriented analysis and design are available which mainly support pure software modeling. Software CASE-tools for the modeling of embedded systems are often new to market. Only three well known CASE-tools support object-oriented analysis and design using UML notation in addition to time-discrete modeling (usually with state charts): ARTiSAN Real-Time Studio, i-Logix Rhapsody, and Rational Rose Realtime are tools explicitly classified for this domain. A major drawback of these CASE-tools is their lack of modeling concepts for control system engineering. For an overall system design users will have to include source-code as external C-code into these CASE environments. This approach is expensive and very rigid. Communication between both modeling domains (soft-ware to time-discrete/ -continuous system parts) has to be done via source code coupling. Hence, changing the system architecture during the development process, in large-scale systems is very susceptible to errors. Therefore, model-based coupling of software components and signal-flow/event-driven system-parts is desirable.

3 Metamodel

In our approach the whole system is described as an instance of one particular metamodel in one notation. The related metamodel has to cover all three domains: time-discrete, time-continuous, and software. The Unified Modeling Language is an Object Management Group (OMG) standard [11] which we use as a system notation and metamodel. It is a widely applied industry standard to model object-oriented software. The abstract syntax, well-formedness rules, Object Constraint Language (OCL), and informal semantic descriptions specify UML. As we will point out later, we use this notation to store the overall model while the developer still designs with the adequate and best-fitting modeling language for each particular subsystem (e.g., signal flow diagram, state charts, UML, etc.).

The UML specification provides the XML Metadata Interchange format (XMI) [8] to enable easy interchange of metadata between modeling tools and metadata repositories in distributed heterogeneous environments. XMI integrates three key industry standards: the eXtensible Markup Language (XML), a W3C standard, the UML, and the Meta Object Facility (MOF) [7], an OMG metamodeling standard which is used to specify metamodels.

One key aspect of UML is the four layered metamodeling architecture for general-purpose manipulation of metadata in distributed object repositories (see Figure 2). This makes it suitable for our universal object-oriented modeling approach. Each layer is an abstraction of the underlying layer with the top layer (M3) at the highest abstraction level. The bottom layer (M0) comprises the information that we wish to describe, e.g., the embedded system, a bridge, the data, or the execution code of the program (see Figure 2, bottom). In our application domain this is typically the source code in different languages, e.g., JAVA or C++, which executes on the target machine. On the model layer (M1) there is the meta-data of the M0 layer, the so-called model. Object-oriented software is typically described on the M1 layer as a UML model. The metamodel on the M2 layer consists of descriptions that define the structure and semantics of meta-data (e.g., the UML model). These are the metamodels, e.g., UML 1.4, or UML 1.5, and define the language and notation for describing different kinds of data respectively (M1). Finally, on the M3 layer there is the meta-metamodel MOF. It is used to describe metamodels and define their structure, syntax, and semantic. It is an object-oriented language for defining meta-data. MOF is self-describing. In other words, MOF uses its own metamodeling constructs.

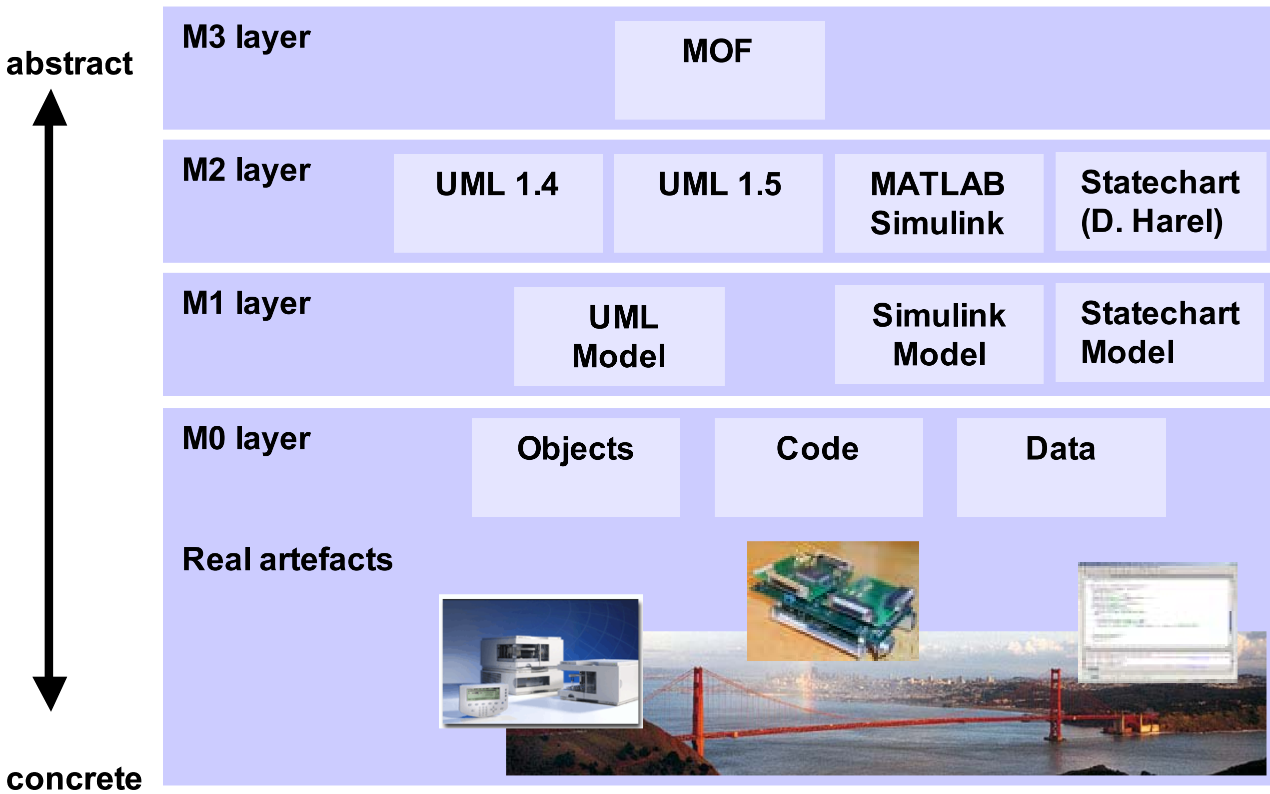

Fig. 2. Four-layer metamodeling architecture

XMI was partially influenced by the ideas for a tool-independent CASE data inter-change format called CDIF [3], which is based on entity-relationship (ER) descriptions. CDIF addresses the problem of model data interchange between CASE-tools.

Without a standardized interchange format for integrating more than one CASE-tool, proprietary import/export filters have to support the exchange of model data. In addition, new interfaces have to be implemented for tool integration. Despite the fact that today there remain incompatibilities because of different interpretations of UML and XMI, it is already supported in a wide range of industry applications.

In the current version 1.5 of the UML standard, it is possible to completely interchange model information. Nevertheless, it is not yet possible to interchange the graphical views of the model in terms of diagrams, which will be supported in the forthcoming UML version 2.0.

4 Integration on Model Level

When we capture current requirements in the design process of embedded real-time systems it is necessary to describe closed-loop-control systems (e.g., time-continuous algorithms), event-discrete, and software design techniques (Figure 3).

Besides the problem of using a large number of description notations/methods, an enormous amount of design data has to be handled. Another problem in many commercial CASE-tools is the lack of supporting concurrent engineering. Only a project file based datastore is offered, which can only be edited by one user at a time.

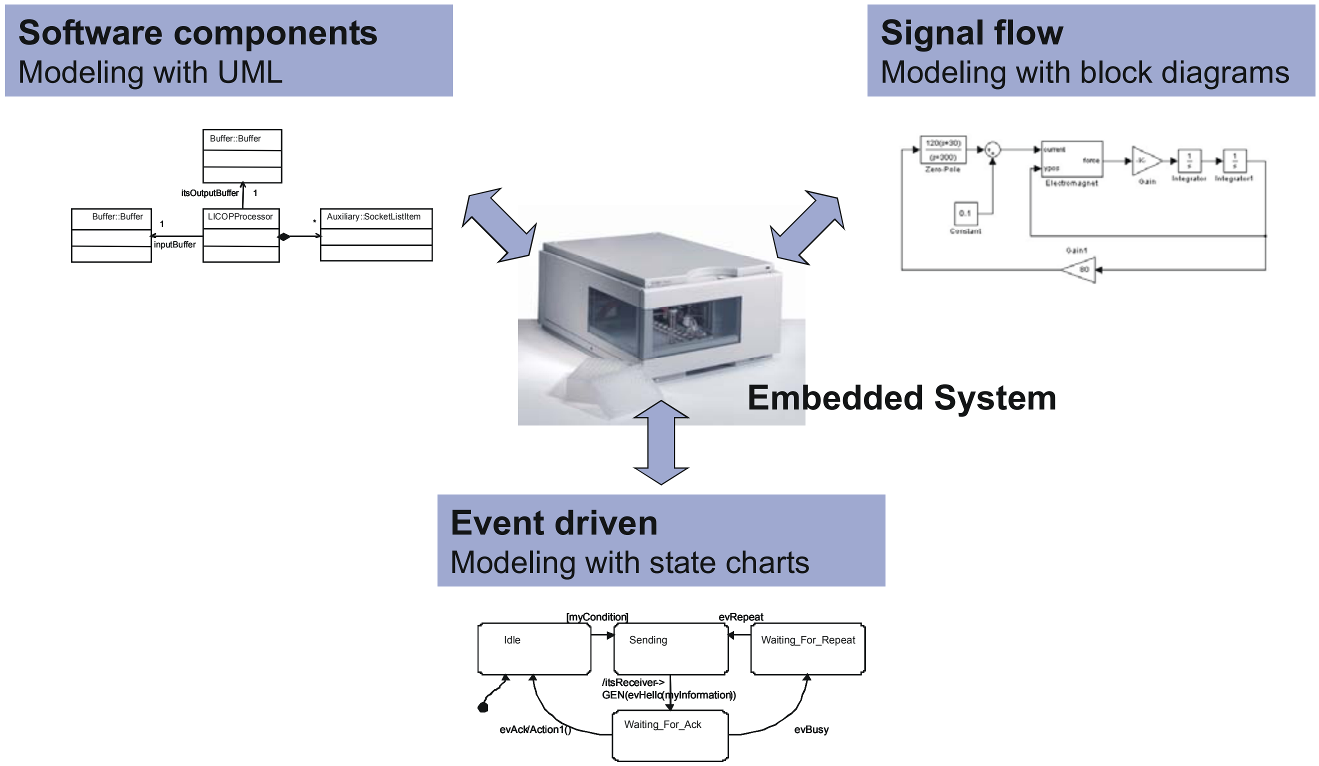

Fig. 3. Modeling domains

Transformation rules are necessary to couple different notations of various domains. In our approach we support different modeling notations used by various CASE-tools in specialized domains. The designer will choose the best tool/notation for a sub-problem and integrate the solution into the UML top-level metamodel. The software domain is modeled in UML. Therefore, no transformation is needed. In the time-discrete domain UML provides a notation but with the lack of well defined semantics. Here, we use the semantics of David Harels concurrent hierarchical state charts [12] implemented in the CASE-tool Statemate. It provides a XMI interface and can interchange a time-discrete model with our integration platform *GeneralStore* (GS) (see Section 7).

The main difficulty when using different description domains in a complex embedded systems design is the integration of control subsystems. There are two possible solutions to overcome this problem:

1. Integration of the time-continuous subsystem using the reverse engineering mechanism of modern CASE-tools for software engineering. This case is called "subsystem coupling on source code level". One major problem here is that the control subsystem is shown as a black box subsystem encapsulated inside a class with the loss of information for other designers (see Figure 4, top).
2. A bidirectional transformation rule (see Figure 4, bottom) enables the synchronization of the time-continuous subsystem to a representation in the UML notation that uses simple object diagrams and class diagrams (see Figure 5, bottom). With an automated CASE-tool-coupling layer, this transformation is reversible. This technique is called "subsystem coupling on model-layer".

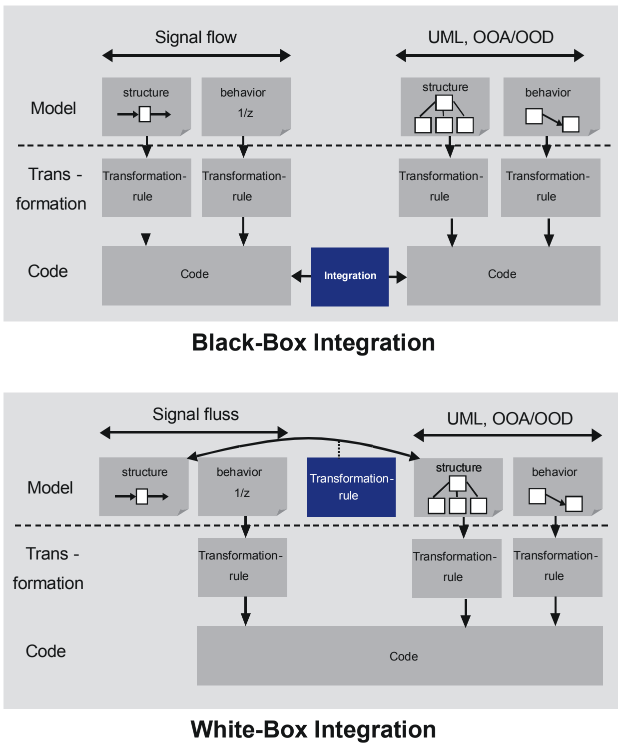

Fig. 4. Subsystem-coupling on source-code- level (top)

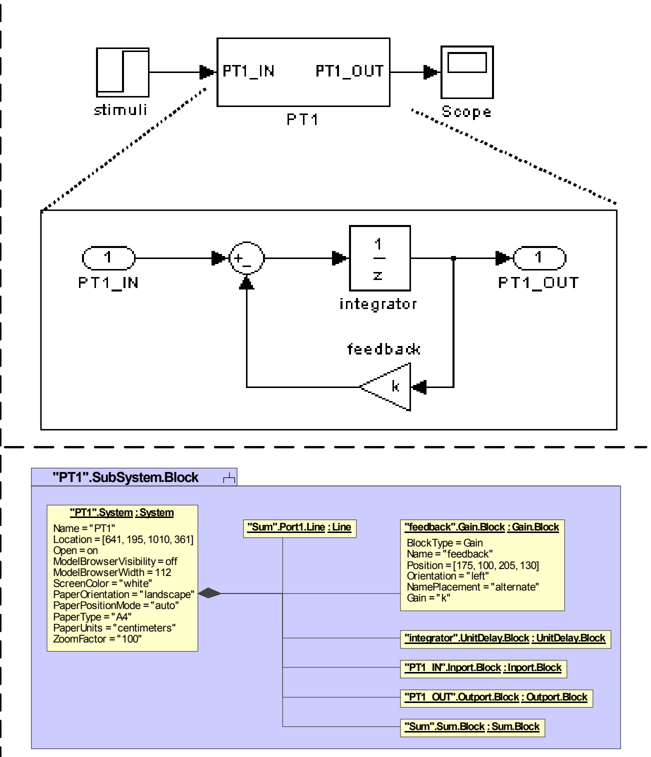

Fig. 5. Transformation of a block diagram to UML and back and on model-level (bottom)

Being more transparent, the second approach is the more suitable way for a convenient integration of both modeling domains. Figure 5 shows a small blueprint of the design process, which is formed by a so-called bidirectional transformation rule. Simplified: blocks in the block diagram are translated to objects in the UML metamodel. Connections represented by lines are also realized as UML objects. The hierarchy in the block diagram is achieved by UML subsystems and links between the objects. Parameters of blocks represent slots on objects. The classifications of objects are classes, whereas links are instances of associations. These elements for type information are located in a separate UML package. We have developed a CASE-tool integration environment (*GeneralStore*, see [4][5] and Section 7) that provides the necessary underlying design process support, which will be introduced next.

5 Development Process

In our eyes, defining a complete and potentially complex design process for our approach would be too rigid, especially because of the various application fields outlined in Section 1, which all require different process models. The appropriate engineering process for automotive development is in many cases different from the conditions in medical engineering. Differing provisions of law and other legal issues (e.g., certification) play an important role.

For object oriented software development, a wide range of methods exist, like the V-Model, the Rational Unified Process [14], the Model Driven Architecture (MDA) [15], or agile methods [16]. Each of these methods can be preferred depending on the project task, organization size, and the organization culture.

Nevertheless, the presented approach puts constraints on the development process, which can be incorporated in different process models.

The process is outlined in Figure 6. The design process starts from a textual specification document, which is the basis for the following requirements specification phase. Usually we model UML use-cases to catalog the requirements and show dependencies. Afterwards, each requirement of the system specification can be translated to a scenario. For these the UML provides sequence diagrams, collaboration diagrams, state diagrams and activity diagrams. Additionally, Object Constraint Language (OCL) expressions and comments can be used to enrich the analysis model.

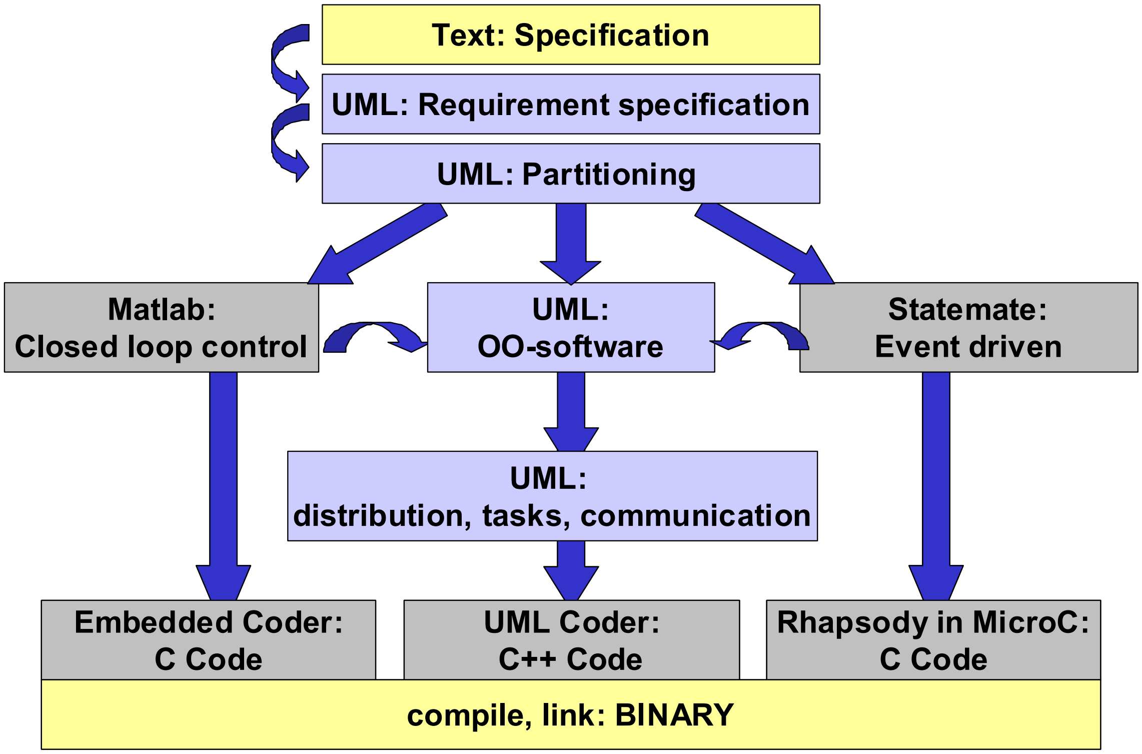

Fig. 6. Development process

Starting from there a preliminary and roughly outlined class model can be extracted. In the next step the developer arranges the class model in various class diagrams, which will then be enhanced step-by-step. Generalizations will be identified and similar classes have to be combined. Furthermore, dependencies and associations will be revealed. By performing these steps, a so-called analysis-model is achieved.

Subsequently the model of the embedded system is divided into different domain sub-models, where each sub-model is a software component, a subsystem in the control-domain, or a state chart in the time-discrete domain (see "UML: Partitioning" in Figure 6).

Now the developers can use their CASE-tool of choice in a notation familiar to their area of expertise. The already mentioned integration platform *GeneralStore* applies automated transformations on these notations to convert them to a top-level UML metamodel in a bidirectional way.

Specialists in control system design use their notations (block diagrams) and tools (e.g., MATLAB/Simulink) to model feedback control and signal processing components. Open loop control and event discrete automation parts can be modeled and simulated using state charts in CASE-tools such as i-Logix Statemate.

By mapping notations to and from UML, we gain complete access to their structure and properties. This includes parameters, signals, and events, but is not limited to them. Thus, we gain a white box integration of the model described in UML notation.

Despite modeling in various domains, all the subsystems are arranged inside one main UML model. Thus, we model the distribution of the subsystems on different ECU's using deployment diagrams. The mapping of functionality to operating system tasks (using object diagrams), the execution framework (class diagrams), and the modeling of the communication between the tasks and the domains are accomplished using UML class and object diagrams.

There are highly efficient commercial code generators on the market. In safety critical systems certificated code generators have to be used to fulfill the requirements. The *GeneralStore* (GS) platform allows for partitioning of the whole system into subsystems. Thus, we enable the usage of different domain specific code generators. Each code generator has benefits in specialized application fields. We follow the Model Driven Architecture (MDA) approach: Transforming a Platform Independent Model (PIM) to the Platform Specific Model (PSM). For control-systems there are commercial code generators like Target Link, Embedded Coder or ECCO. In the time-discrete domain we utilize the code generator of Statemate (Rhapsody in MicroC). In the software domain commercial code generators typically generate the stubs of the static UML model while behavioral functionality has to be implemented by the software engineer.

As we focus on a completely generated executable specification, it is necessary to generate code of the overall model. Therefore, we provide a code generator as a GS plug-in to enable structural and behavioral code generation directly from a UML model. The body specification is done formally in the Method Definition Language (MeDeLa), which is a high level action language based on Java syntax. It suits the action semantics defined by the OMG since UML version 1.5 as a concrete syntax. Our template code generator uses the Velocity engine (see http://jakarta.apache.org/velocity) to generate Java or C++ source code. Velocity is an Apache open source project focused on HTML code generation. It provides macros, loops, conditions, and callbacks to the data model's business layer. One of its strengths is its speed when rendering.

In order to reflect the mentioned design pattern for domain coupling in the generated code, a tool like *GeneralStore* has to provide a way to automatically generate wrappers for the control and time-discrete domain. The wrapper is generated on the UML model level and is transformed to code with our code generator (for more details on this step see Section 6). Finally, one can compile the source files and validate the running model.

6 Coupling the models with UML templates

The template mechanism which is defined in the UML specification, allows modeling of parameterisable model elements, like classifiers or packages. This mechanism offers modeling common, independent models which will be adapted at a later date by the usage of template arguments which are put as values for the template parameters. In *GeneralStore*, this mechanism is applied to classifiers, used by the wrapper generator which will be explained afterwards. Figure 8 shows an example of a UML templates where *type* and *size* are template parameters and *Command* and *42* are the template arguments at the binding symbol. At the bottom of Figure 8 the class is shown after the UML rendered is executed.

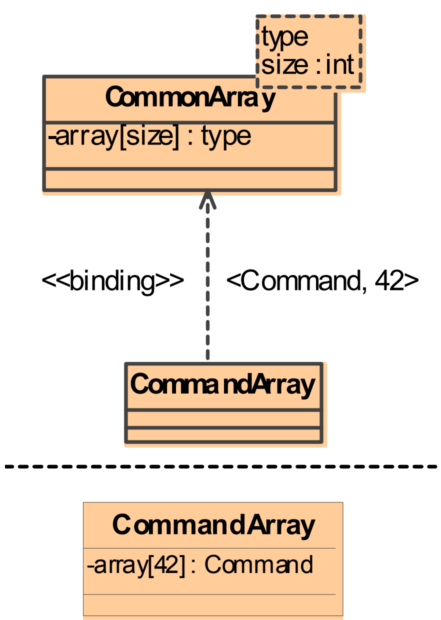

Fig. 7. UML template mechanism

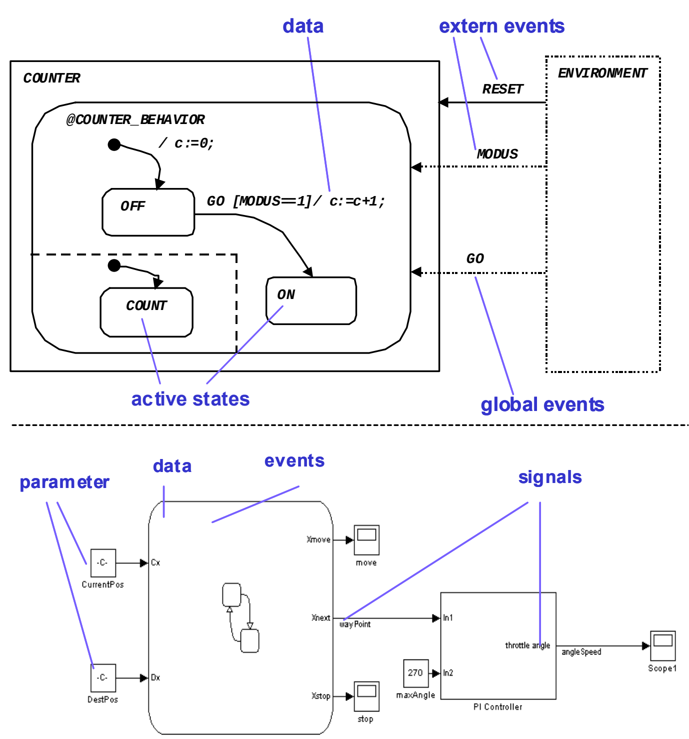

Fig. 8. Possible model interfaces

To get a completely integrated model it is required to couple the different models (modeled in different notations and different CASE tools) in a manner which enables the communication and data exchange between the models. This model

interfaces have to be defined in the used CASE tool explicitly, e.g., in MATLAB/Simulink/Stateflow they (parameters, signals, events and data) have to be marked as ”exported global“. In Figure 7 some examples for model interfaces are shown.
These interfaces will be wrapped in the design process through small automatic generated UML-Models so called wrapper, which know the model’s interfaces and offer the developer a common interface layer to access these models.
The wrapper generation process uses a UML template model labeled “wrapper template“. This template model is open to the developer; it is possible to describe different communication ways like shared memory or a CAN link. The model specific information are introduced into the wrapper template model by parameterising classes with UML templates and using these template parameters in the wrapper template model.

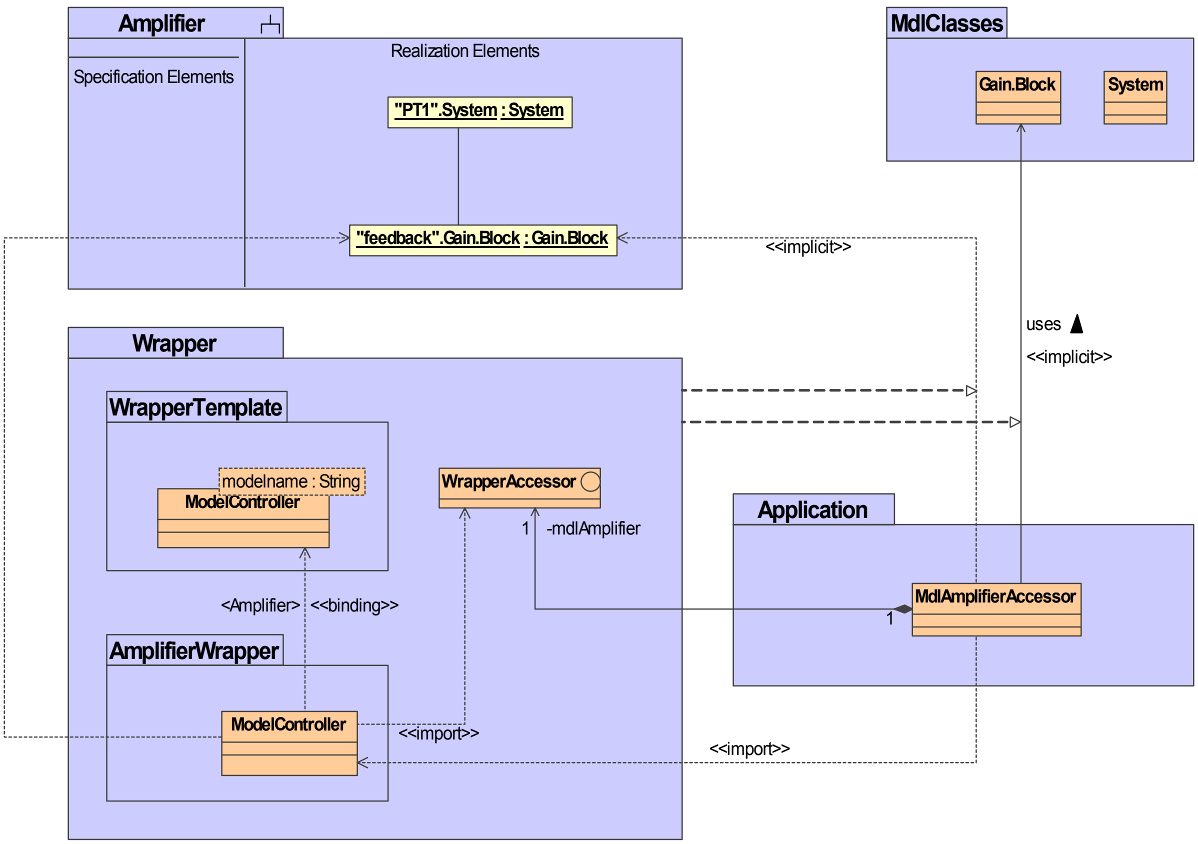

Fig. 9 Usage of the wrapper

To generate a wrapper, some specific information, e.g., data type and dimension, are needed and read out of the model to be wrapped. These information are used as template arguments for the template parameters in the wrapper template model.
It is possible that for each template class more than one parameterised class has to be generated. In this case, the rendering process on the template class has more cycles. Then the name of such a class has to consist of template parameters which ensure that no parameterised classes have identical names.
It is also possible that a template argument has to contain more than one value. Assumed a template class has the job to manage other template classes which may have parameterisations in operations. A list of values is needed as template argument.
The wrapper itself will be generated automatically, triggered by a user input. Then the model to be wrapped will be examined, it’s specific interface information are stored as template arguments and the wrapper template model will be transformed into a specific wrapper. To encourage this transformation process the wrapper generator is equipped with a couple of features:
The *copy feature*: Model elements, which have no template parameters and (in the case that it is a relation) are not connected to such elements directly, will be copied.

The *render feature*: Classifiers, which own template parameters will be rendered. During this work, all uses of the template parameters are processed and replaced by the template arguments. In the case that the template arguments are lists of values, the objects containing template parameters will be copied and then replaced by one value of the template argument. This is done for every value in the template argument once. Relations like associations, generalisations, and dependencies, which are connected to the template model element will be copied and attached to the parameterised model elements.
The *modification feature*: The activity language MeDeLa has been extended with some keywords to make iterations over template arguments possible. This is needed for template arguments which are lists of values. Then an iteration over all of the values will be inserted to make sure that all values are used.
The *association checking feature*: Assumed there are no template arguments available for a template class during wrapper generation, this class and it's relations to other classes will be disregarded. But if such a relation is needed by the *MeDeLa* description of an associated class, the model would be incorrect. Therefore *#ifdef* definitions in *MeDeLa* are possible which will be checked during the generation process.
The *relation checking feature*: May be the modeller wants to model dependencies or associations which are useful for understanding the model but are not needed for the code generation. With the stereotype <<*implicit*>> such a relation will be skipped in wrapper generation process.
The *common wrapper interface feature*: The wrapper template model can be encapsulated by an interface which will not be rendered during wrapper generation. This architecture allows using a wrapper in the user's UML model without having a model to wrap and it's generated wrapper at modelling time. After wrapper generation the developer has to attach the generated wrapper to his model.
A small example shown in Figure 9 is following to clarify the mechanism. It shows how to wrap a MATLAB/Simulink subsystem and use it in a UML application.
The UML representation of the MATLAB model consists of the subsystem itself as an object tree and the MATLAB classes which describe the MATLAB notation (also compare Figure 5).
In the UML application the developer has modelled a class named *MdlAmplifierAccessor*. This class should delegate all tasks to the wrapper to be generated. The next step is to connect it to the MATLAB parameter. Therefore an association called *"uses"* is drawn to the *Gain.Block* class to model the communication between these classes. The dependency from *MdlAmplifierAccessor* to the object *"feedback".Gain.Block* indicates which parameter has to be wrapped. Both, the dependency and the association are stereotyped <<*implicit*>> because they are realised by the wrapper to be generated. The composite association to the *WrapperAccessor* interface is the first step of the wrapper integration. These few modelling steps suffice to integrate a MATLAB model into a UML application. To get a working wrapper, the generation process has to be activated. There the wrapper is inserted into the model and some import dependencies are drawn. This workflow enables the developer to model a link-up, e.g., a MATLAB model without the need to generate a wrapper at modelling time. Directly before code generation the wrapper can be generated, when everything else has been modelled. GeneralStore supports the execution of the model to model transformation which is the UML template to UML model transformation.

7 Integration platform *GeneralStore*

To keep the system model manageable for designers, CASE-tool integration is necessary. In [1][2] an open design environment based on CDIF for the development of mechatronic systems was presented. Many of the experiences from this project have influenced our current work.

With the completion over the last three years of the prototypes design, the focus of our current work has shifted to *GeneralStore* (GS), a so-called integration platform [6]. Figure 12 shows an overview of the architecture. The model data could be stored in the commercial object-relational database management systems ORACLE and MySQL. In the core of GS we provide authentication, transaction management, object versioning, and configuration management. For managing CASE data, GS supports UML as its metamodel (see Section 2). GS uses MOF as its database schema for storing UML artifacts. Inside the database layer an abstraction interface keeps GS independent from the given database. While interim objects on the business layer enclose MOF elements, the CASE adaptor stays thin and highly reusable. These interim objects are used to enable object identity to handle object versioning and configuration management.

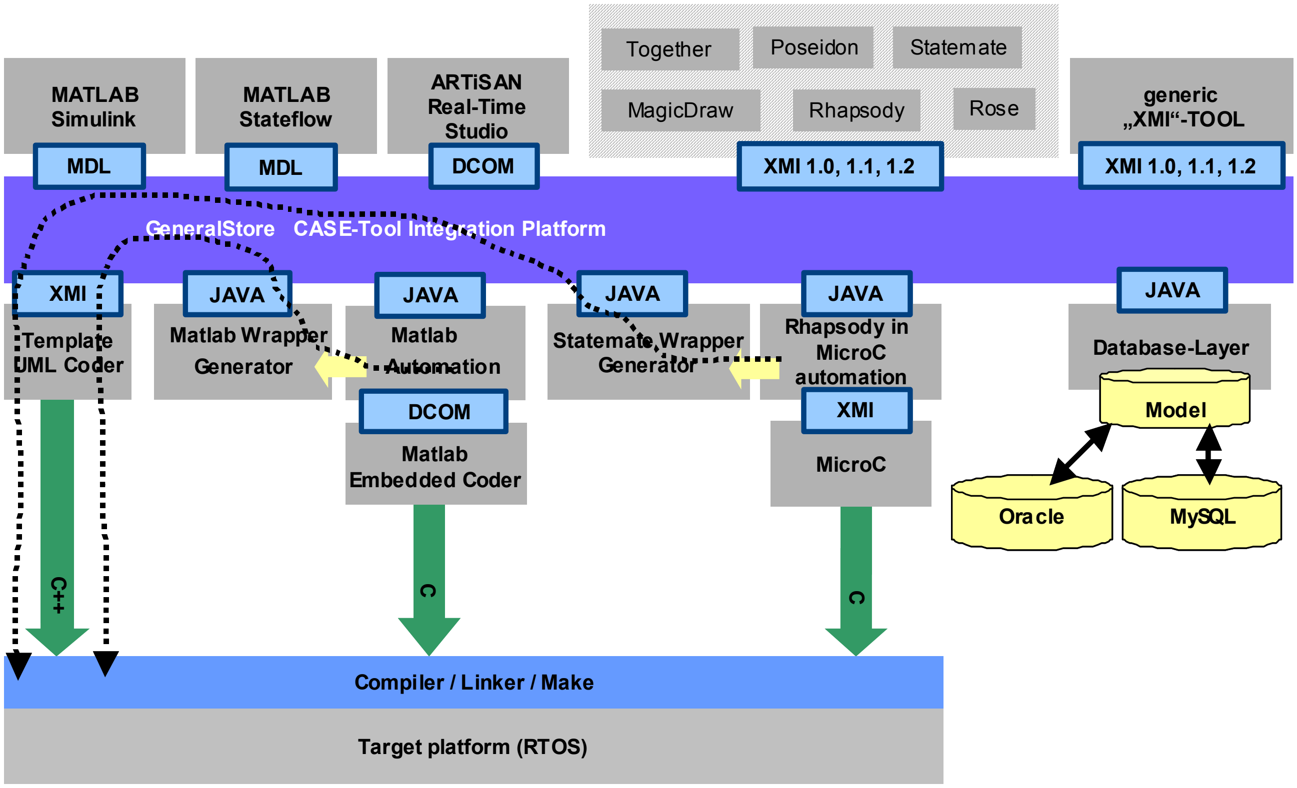

Fig. 10. Architecture of GeneralStore

To explain the versioning strategy, we consider an example where a unique identification number is added to a subsystem block from the time-continuous system domain. After the block is checked out from the repository, a designer moves this subsystem to another hierarchy layer. Despite the fact that it is the same compound object, the system controller in the time-discrete system part keeps track of this link because of the subsystem identification number.

The transformations for specific notations supported by CASE-tools are implemented using plug-ins (see top of Figure 12). GS provides three principal CASE-tool adapters:

1. MATLAB/Simulink/Stateflow was selected for the control system design domain and the integration is done using the proprietary model file (MDL).
2. Generic and specialized XMI importer/exporter filters of *.xmi files: Here we use XSLT transformations [13] to adopt the tool specific interpretation of the UML and XMI standard. The UML CASE-tools we chose are Together (Borland), Poseidon (Gentleware), MagicDraw (No Magic, Inc.), Rhapsody in C++/JAVA (i-Logix), Ameos (Aonix), and Rational Rose (IBM). Statemate (i-Logix) was chosen in the time-discrete domain.
3. DCOM based integration of ARTiSAN Real-Time Studio: This UML CASE-tool was selected because it focuses on embedded real time systems.

All tools, except Statemate, which only allows for the export of XMI files, are linked to the GS architecture in a bidirectional way.

The code generation plug-ins (Template UML Coder, Embedded Coder, and Rhapsody in MicroC) controls the transformation to the source code. Their wrapper generators are automatically building the interface to the UML model (see middle of Figure 12 and compare to Section 6).

For model management and CASE-tool control, GS offers a system hierarchy browser (compare to Figure 13). Since the internal data model representation of GS is UML, GS offers a system browser for all UML artifacts of the current design. Due to the large amount of MOF objects (e.g., the transformed PT1 subsystem needs about 10,000 XMI entities), GS offers design domain specific hierarchy browsers, e.g., a system/subsystem hierarchy view for structural or time-continuous design, or a package hierarchy view for software design.

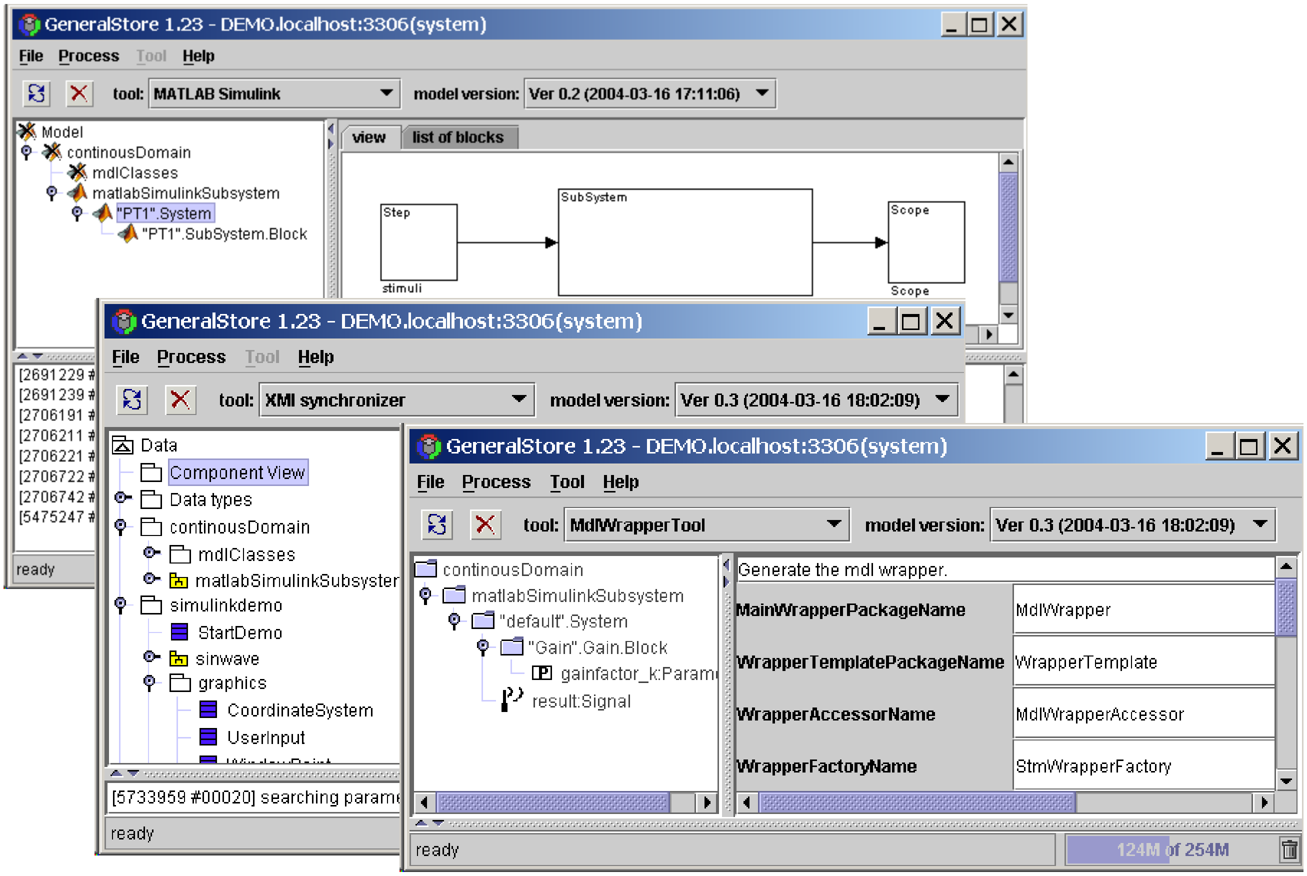

Fig. 11. GeneralStore plug-ins for MATLAB, XMI interface, and MATLAB wrapper generation

8 Conclusion

The CASE-Tool integration platform *GeneralStore* combined with our universal object-oriented modeling approach supports the concurrent development of embedded electronic systems in all design phases. We showed how heterogeneous system descriptions in different notations could work as integrated parts of an object repository based client/server CASE-tool environment. Based on an object diagram representation, time-continuous subsystems and software components can be modeled using one single metamodel. A direct link between different description domains is possible on an abstract model level. When transforming all subsystem parts to an uniform object notation, adding additional model information to time-continuous blocks will enable system designers to start with system simulation early in the design process.

Embedded electronic systems can be subdivided in event-driven, signal-flow oriented, and software domains. Each domain uses its specific notation. A highly flexible design process was described to integrate those notations, which are supported by different CASE-tools. The glue between the domains is modeled in UML. Finally, the overall system is transformed to source code with the assistance of commercial code generators in addition to our UML template based code generator.

An often needed missing feature of CASE-tools based on project files (e.g., Rhapsody, MATLAB Simulink/Stateflow) is the lack of CASE-tool assisted concurrent engineering. Using the presented CASE-tool backend *GeneralStore* together with MATLAB/Simulink, an interim project file is created each time a designer checks out a part of the model. This is possible at any specific model hierarchy point. The checked out subsystem hierarchy becomes protected. Other designers still have read access to the last version of this subsystem and can obtain read/write access to other subsystems in the project hierarchy.

9 Outlook

One major drawback of using UML/MOF from XMI as a metamodel for system description is the deficiency of a standardized graphical representation for class and object diagrams. As of today, this is one of the most requested topics for UML 2.0 and the next XMI generation. On the other side, using XMI and the UML metamodel for the description of embedded systems enables model exchange with other CASE-tools. Today, at least 10 software CASE-tools are on the market, which can handle XMI descriptions from the information point of view (import for a model without graphical description).

Future work will also have to focus on the backend side of our development environment; simulation, debugging, and emulation of system models are going to be supported.

We will examine the integration of further modeling tools to estimate the saved integration effort using UML/MOF. Especially for large systems of different design domains, our universal object-oriented modeling approach for the design of embedded electronic systems based on MOF will show its advantages.

Furthermore, it is planned to work on enhanced modeling of operations in UML for code generation, e.g., collecting information from activity charts, object collaboration diagrams, and state charts.

The UML 2.0 template mechanism could be adapted and used after the migration of the *GeneralStore* data model to the new UML 2.0 standard.

References

1. A. Burst; M. Wolff; M. Kühl; K. D. Müller-Glaser: A Rapid Prototyping Environment for the Concurrent Development of Mechatronic Systems, *ECEC*, Erlangen, Germany, 1998.
2. A. Burst; M. Wolff; M. Kühl; K. D. Müller-Glaser: Using CDIF for Concept-Oriented Rapid Prototyping of Electronic Systems, *RSP*, Leuven, Belgium, 1998.
3. EIA / CDIF Technical Committee: *CDIF / CASE Data Interchange Format. EIA Interim Std. EIS / IS- 106-112*, 1994.
4. M. Kühl; C. Reichmann; B. Spitzer; K.D. Müller-Glaser: Universal Object-Oriented Modeling for Rapid Prototyping of Embedded Electronic Systems, *RSP*, Monterey, USA, 2001.
5. M. Kühl; C. Reichmann; K.D. Müller-Glaser: Universal Object-Oriented Modeling with ARTiSAN Rts and MATLAB/Simulink, *ARTiSAN User Conference 2001*, London, UK, 2001.
6. M. Kühl; C. Reichmann; K.D. Müller-Glaser: An Overall System Design Approach doing Object-Oriented Modeling to Code-Generation for Embedded Electronic Systems, ETAPS/FASE, Warschau, 2003.
7. Object Management Group: *Meta Object Facility (MOF) V1.4*, 2001.
8. Object Management Group: *XML Metadata Interchange (XMI) V2.0*, 2003.
9. B.P. Douglass: *Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
10. M. Fowler: *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.
11. Object Management Group: *Unified Modeling Language (UML) V1.5*, 2003.
12. D. Harel: Statecharts: a visual formalism for complex systems. *Science of Computer Programming,* 8 (1987,3), 231-274
13. D. Sussman, Michael Kay: *XSLT Programmer's Reference*, WROX, 2001.
14. T. Meloche: *The Rational Unified Process*, The Menlo Institute LLC, 2002
15. A. Kleppe, J. Warmer, W. Bast: *MDA Explained: The Model Driven Architecture - Practice and Promise*, Addison-Wesley, 2003.
16. K. Beck, Martin Fowler: *Planning Extreme Programming*, Addison-Wesley, 2000.