Fraunhofer Institut
Experimentelles
Software Engineering

UML-based Development of Embedded Software Systems

Author:
Christian Bunse

Submitted for Publication at
UML 2004

IESE-Report No. 026.04/E
Version 1.0
February 2004

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Sauerwiesen 6
67661 Kaiserslautern

Abstract

Model-driven development, using the UML, has become the most dominant development paradigm, particularly in business and web application engineering, due to their many advantages over traditional procedural approaches. However, Model-driven and UML-based development methods are still inferior to conventional software development approaches when it comes to embedded and real-time system development. Most such methods provide only weak systematic and methodological support for system development in the embedded domain. The most fundamental problems in this domain stem from the fact that individual techniques for embedded system development only acknowledge and address the particularities of object and component technologies insufficiently, and more importantly, that individual technologies are mostly treated in isolation. One important aspect is the heterogeneity of embedded systems, containing both, hardware and software components. A systematic approach for the model-driven development therefore has to address this fact by providing a uniform but systematic way for modeling hardware and software parts of the system. This paper discusses how software and hardware components of an embedded system can be uniformly described with the UML, and it presents a small case-study.

Table of Contents

1 Introduction

The success of a company depends directly on its ability to adapt to changing market requirements. This is particularly true in the domain of embedded software systems, where the rising decentralization and the increasing portion of software in technical systems require development methods for correct, safe, cheap, and flexibly adaptable control systems [9]. Unfortunately, despite their popularity, current model-driven methods have serious weaknesses, which limit their effectiveness for embedded software engineering. Most prominent among these is the lack of systematic methodological support for the integrated and uniform modeling of embedded systems based on a "standard" modeling language, such as the UML [14, 20]. This results not only in an early decoupling of hardware and software development that leads to inconsistencies and integration problems, but also to heterogeneous descriptions resulting in systems with uncertain and usually low quality.

The idea of building systems by stepwise and hierarchic decomposition and refinement of larger building blocks into smaller building blocks has been around since the days of structured analysis and design [1]. However, when each building block (i.e. component), be it hardware or software, is described by its own suite of diagrams, it is important that each diagram fits into the overall hierarchical organization scheme and focuses on the component it is intended to describe [2]. This requires that every development artifact is oriented towards the description of a single architectural component. Having a concise, focused description of an individual component also simplifies the task of the development teams charged with implementing/realizing the component.

Many approaches to embedded system development [9,21] change the way in which they represent a particular kind of abstraction or concept during the development process. Moreover, the choice of representation is often based solely on the level of granularity, or phase of development, in which the abstraction is used rather than on the properties of the artifact being described. To avoid this unnecessary redundancy, it is essential that a given component (either hardware or software) is represented and manipulated in the same way in all phases of a project [2]. This also requires that all components are modeled through the same basic concepts regardless of their level of granularity or position in a development project. In other words, the nature of the models describing a component should be dictated only by its properties.

Following this principle not only simplifies the task of the developers, since it reduces the number of concepts and associated processes that they have to learn and differentiate, but also inherently supports scalability, since big com-

ponents are treated in the same way as small ones. Furthermore, the principle enhances reusability, since a uniform representation approach simplifies the task of incorporating existing components in new applications.

The remainder of this paper is structured as follows: Section two briefly goes through current approaches for modeling embedded software systems, and outlines why they are only insufficiently addressing the problems of embedded system development. Section three presents a uniform way for modeling hardware and software components with UML, whereby section four describes how the modeling approach is integrated into a larger method, including tool support, for embedded system development. Section five illustrates the principles through a case study, an autonomous vehicle. Finally, section six summarizes and concludes this paper.

2 UML-based Development of Embedded Software Systems

Developing embedded systems with the UML, on the one hand, requires notational elements to capture the specific properties of embedded systems, such as timing and precision, or even hardware connectors. Section 2.1 discusses how such elements can be integrated into the UML. On the other hand, the UML, like any other notation, is only a tool. Therefore, section 2.2 gives an overview on UML-based development methods which exactly define which, when, and how specific UML elements have to be used.

2.1 Notations

Before the advent of object technology formal languages such as Z, Z++, and discrete or continuous functions were often applied for the description of technical systems [13]. Further examples for such techniques are functional decomposition as proposed by Mills [15], state-based notations, MATLAB, a tool used in the automotive industry, or the table notation SCR proposed by Parnas [7]. However, most of these languages are purely function-oriented which makes the application of the object-oriented paradigm difficult if not impossible.

Since the introduction of the object paradigm, the Unified Modeling Language (UML) [17] has become a de-facto standard notation for modeling software systems graphically. However, the UML does not provide precise semantics, or explicit means for modeling the specific requirements of embedded systems. This deficiency led to a set of suggested improvements that are summarized in the following:

1. Extension or formalization of the UML by integrating other languages, such as SDL [16], that is coming from the telecommunications domain and has been extended with object-oriented features. The result is a UML variant (a so-called profile) that contains and provides support for SDL features. Further examples for UML extensions are the German research projects IOSIP [11] and USE [8]. The goal of IOSIP was the integration of UML modeling techniques, as well as extending the UML by application-specific description techniques, with the idea of developing a reference model that can be validated. A goal of USE was the integration of Message Sequence Charts and UML sequence diagrams, consistency checking between these two diagram types, and their transfer to object-oriented description techniques.
2. Application of standard UML with stereotypes for real-time constructs. Unfortunately, by now, the concepts of response time and performance could not be sufficiently embedded into a development methodology. Ap-

proaches, such as that proposed by Douglas [9], are limited to the specification of non-functional requirements in the form of associated attributes. However, the impact of such characteristics on system structures and algorithms, or on their verification is not really successfully tackled by these approaches.

3. Definition of a formal semantic. The German research project InTime [3] dealt with the adaptation of UML for real-time systems. In particular the missing semantics of the UML, as well as the missing guidelines for systematically applying UML are approached through the development of a real time variant of the UML. The project uses the ROOM method [21] and the very general concept of recursive refinement as a basis.
4. Translation of UML models. Telelogic's Tau-Tool allows, in the context of the SOMT method [23], the translation of UML models to SDL descriptions. These SDL descriptions can be refined and implemented in subsequent development phases with support from the tool.
5. Definition of special UML variants (profiles), which directly allow the modeling of special characteristics of embedded systems. Examples are the profiles suggested by Martin [14] or the OMG [20]. However, without precise semantics, and without the support from a sound method it is difficult to apply such UML variants.
6. The latest version of the UML (i.e., UML 2.0 [18]) offers a number of enhanced features such as improved support for architecture, better support for component-based development, and more powerful sequence diagrams, and promises more relevance to the embedded systems community. However, the final version will be released by the end of 2004, which means that first experiences of using UML 2.0 for embedded system development still have to be made.

In summary, much research effort has been spent in order to integrate elements for the depiction of embedded systems into the UML. This has lead to the integration of modeling diagrams and elements for embedded systems into UML 2.0. However, it is now the task of development methods to systematically apply these modeling elements. The next section discusses how modern methods for UML-based development can be used for developing embedded systems.

2.2 Methods

Many observers forecasted that object and component technology will be the major driving factors for re-use in system development [22]. Classes and objects supported by mechanisms such as inheritance or polymorphism guarantee the widespread, institutionalized re-use and produce a market of multiple reusable components.

Contemporary software development methods have their roots in the first generation object-oriented methods such as OMT [19], Fusion [6], or ROOM [21]. The Object Modeling Technique can be seen as the basis for all subsequently published methods, techniques and tools that are built upon the same fundamental concepts. However, most of these methods are either based on the waterfall model or provide incomplete support, since fundamental and typical concepts (e.g., inheritance and polymorphism in the case of HOOD) are missing. Even methods specifically designed for the development of real-time systems (e.g., ROOM) have their limitations. Newer approaches, such as ACCORD/UML [24], are based on abstractions of typical system by using a UML profile to model all aspects of embedded software quantitatively and qualitatively. However, neither explicit re-use strategies nor the influence of non-functional characteristics are considered.

In summary the expectations concerning re-use, in particular within the range of technical control systems, have not yet been fulfilled. One of the key problems is that object-oriented languages are compiler-bound. Thus objects are merged at compile-time into larger systems and are only of limited benefit for widespread and platform independent re-use. The component paradigm decreases this problem, by declaring individual objects to executable units. Thus the object-oriented paradigm is extended through releasing objects from the chains of the surrounding program. The development of component-based systems corresponds more to an assembly and deployment of individual parts instead of merely constructing larger units out of individual parts as in traditional programming.

Catalysis [10] was amongst the first methods to use or integrate the UML, contemporary component technologies, and modern re-use techniques. However, Catalysis defines a large number of principles, techniques, and artifacts without systematically defining their relations, and their application and use throughout the entire development process. Developers therefore have to rely on their experiences in configuring and applying Catalysis. In addition, the development of technical systems with their specific non-functional characteristics is not addressed sufficiently.

The Unified Process [12] represents an attempt to integrate methods such as OMT, Booch, and Objectory. However the Unified Process, defined as a standard, is only vaguely applying rules and guidelines that help developers or application programmers to achieve their daily tasks. For example, they require the development and application of models without stating how to perform the modeling, how to incorporate non-functional requirements, or how to assure the overall quality of the resulting system.

The KobrA method [1] propagates the use of components throughout all phases of the software life cycle. This goal is achieved by integrating the three most important software-engineering paradigms today: Components, Product

Lines, and Model Driven Architectures (MDA). In addition, the KobrA method comes equipped with powerful means to achieve continuous, model-driven quality assurance.

In summary, existing component-based development methods provide little guidance on how to achieve their promises under stringent constraints of embedded developments. In particular, quality requirements are often completely ignored during the development, and they are later burdened upon the testing phase, or it is simply taken for granted that the component-based methods, by definition, lead to high quality software units right from the beginning of a project. Such practices and assumptions are utterly detrimental. Quality must be built into the components on purpose, and this principle must be followed right from the very start of the project. In doing so, existing techniques, methods, and tools need to be tailored and used for achieving this overall goal. However, quality-enhancing technologies are often limited to conventionally structured development methods.

3 Modeling Hardware/Software Components with the UML

3.1 Modeling Principles

Most existing component-based methods only regard an entity as a component if it is implemented through a specific construct (e.g. a Java Bean), or modeled by using a particular abstraction (e.g. a component icon). In other words, being a component is regarded as an absolute property. In fact, being a component is a relative term rather than an absolute one. The term “component” indicates that one artifact (the component) may be a part of another artifact (another component), and certainly not that it is described in some particular abstraction. Composability may therefore be regarded as a key feature, and composition as a key activity in component-based development. Methods such as KobrA or MARMOT [4] recognize this fact in that they advocate composition as the single most important engineering activity. A system can thus been viewed as a tree-shaped hierarchy of components, in which the parent/child relationship represents composition (i.e. a super-ordinate component is composed out of its contained sub-ordinate components).

Another, long established principle of software engineering is the separation of the description of what a software unit does (e.g., "specification", "interface" and "signature") from the description of how it does it (e.g., "realization", "design", "architecture", “body”, and "implementation"). This facilitates a "divide and conquer" approach to modeling in which a software unit can be developed independently. It also allows new versions of a unit to be interchanged with old versions provided that they do the same thing.

This principle is as important when modeling architectural components as it is when implementing them [5]. A component modeled according to this principle is essentially described at two levels of detail - one representing a component's interface (what it does) and the other representing its body (i.e., how it fulfills the specified interface). Following this principle each component of a system can be described by a suite of UML diagrams as if it was an independent system in its own right. This is shown in Figure 1. This separation allows developers who want to use an existing component or to replace one component with another to concentrate on the interface, neglecting the details of the body.

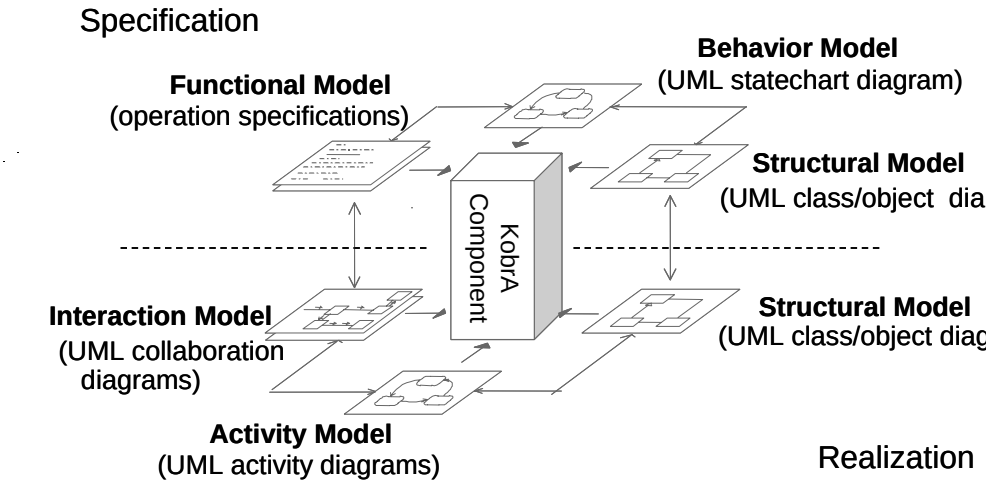


Figure 1 Component Model

3.2 Embedded Components

The idea of modeling the components of a system using a standard suite of models is general applicable in that it can also be applied to non-software components. In detail this means, that software and hardware components are treated in the same logical way. Thus, components can be either hardware or software. In general, this simply means that the concept of a component (see Figure 1) is extended by defining additional stereotypes: <<Electronics>>, <<Mechanics>>, <<Mechatronics>> in order to indicate the respective feasible device types, and in order to provide the correct set of component specification artifacts accordingly (see Figure 2).

On the specification level a simplified view on all types of components, except the distinction between software and various hardware components can be used. (i.e., all components use the same suite of UML diagrams as depicted in Figure 1). However, at the realization level this view has to be specialized.

Here we have to distinguish between new in-house developments and component reuse (i.e., a decision has to be made if there is a product on the market that fulfills the component specification). In its simplest form, (re-)using an existing component in the realization of another component is just a matter of instantiating it, and using its services in a way that conforms to their client-ship rules, defined through the specification.

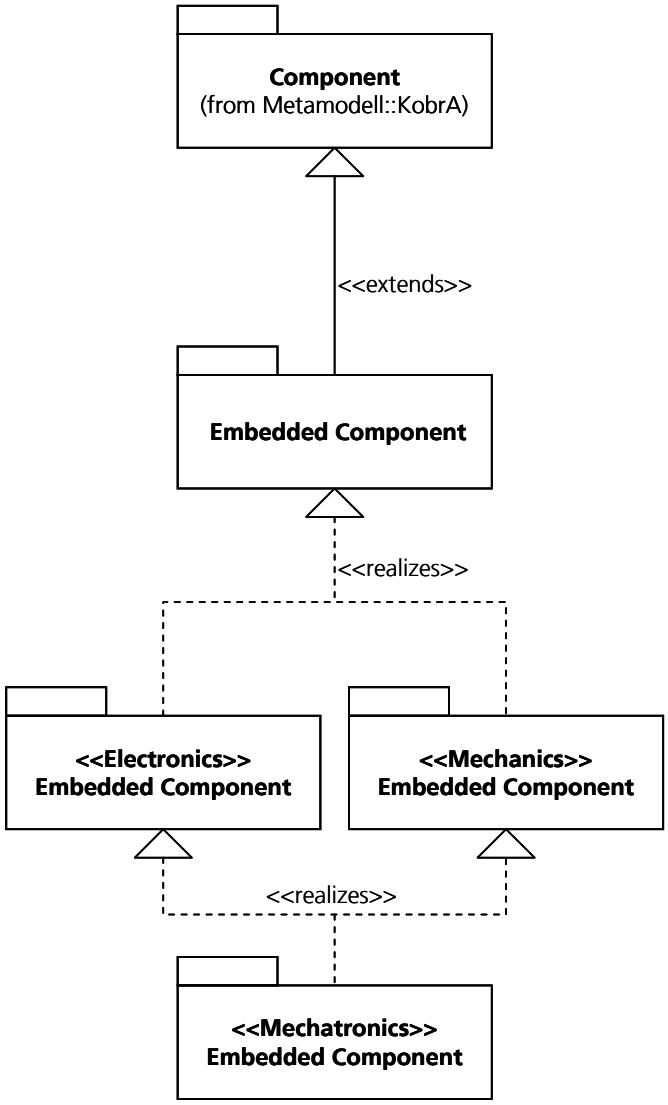


Figure 2 Embedded Components

Sometimes it is not possible to realize a hardware-component through reuse of an existing COTS-component. Instead a specialized piece of hardware has to be developed in order to fulfill the desired specification. The previously described principle offers means to combine hard- and software development in a unique way. In detail, this means that specific realization artifacts can be used to describe the realization of different hardware components.

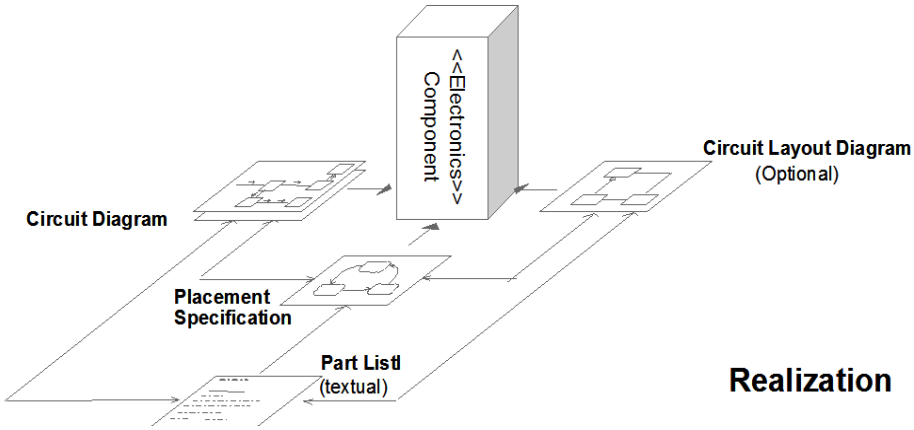


Figure 3 Realization of a <<Electronics>> component

A good example for this is the realization of an <<Electronics>> Component. Basically this is a component which realizes its interface in form of some kind of electronic circuit and hardware parts. Thus, the realization of such a component can be described using standard electrical engineering descriptions (see Figure 3). In detail, an <<Electronics>> component uses four realization artifacts, in order to use standard procedures from electrical engineering for manufacturing:

1. The *Circuit Diagram*, which describes the logical paths between the building blocks of the system through which an electrical current or signal can be carried. In practice, these paths can be implemented using wires or printed connections on a specifically designed board (see Figure 4).

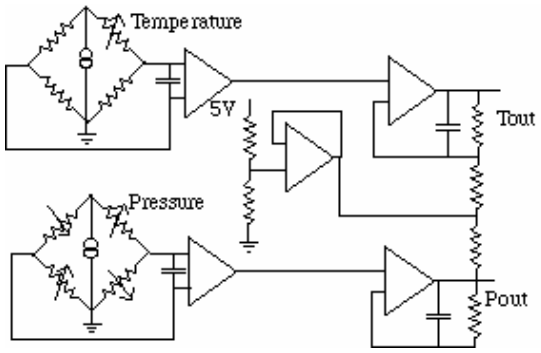


Figure 4 Circuit Diagram

2. The *Part List*, which describes the needed building parts in a textual form by giving the quantity, manufacturer and a unique identifier for of each part (e.g., 1 x; Capacitator; HARDparts Inc.).

3. The *Circuit Board Layout* (see Figure 5), an optional artifact based on the decision if the building parts are connected by wire or by a printed board layout needed for mass production. It is needed during production and the layout can then be created automatically from the circuit diagram by using tools such EAGLE™ offered from CADSOFT.

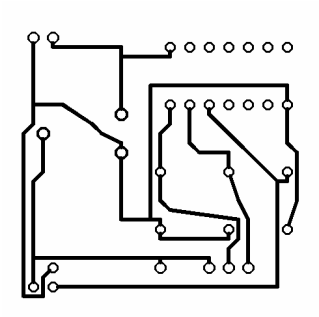


Figure 5 Circuit Layout Diagram

4. The *Placement Specification* (see Figure 6) describes which part (of the *Part List*) has to be placed on a specific location (of the *Circuit Layout Diagram*) or how this part is connected to other parts (of the *Circuit Diagram*).

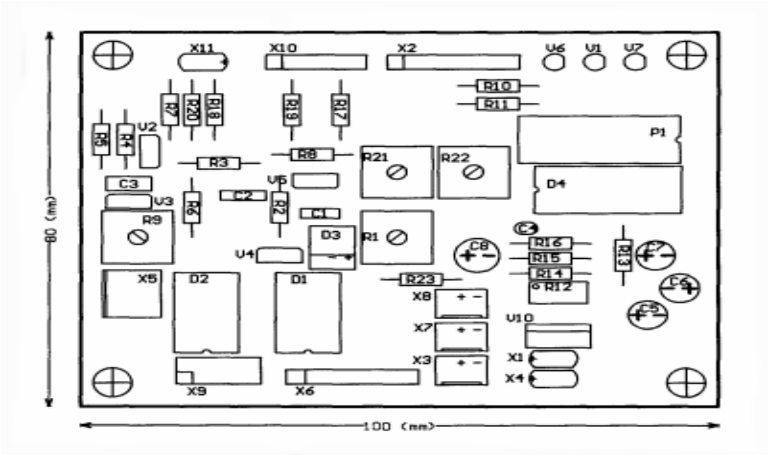


Figure 6 Placement Specification

The realization of a <<Mechanics>> component, a device represented by an assembly of mechanical parts (e.g., transmissions, gears, etc.) follows the same principle (see Figure 7). In contrast to <<Electronics>> and <<Mechanics>> components <<Mechatronics>> components represent an assembly of mechanic and electronic parts. It aggregates the characteristics of <<Electronics>> and <<Mechanics>> components. A typical example for a mechatronic device is a limited-slip-coupling which uses mechanics to realize the coupling and elec-

tronics to control the behavior. Thus, the realization of a <<Mechatronics>> component has to reflect this dual nature by using the realization artifacts of both <<Electronics>> and <<Mechanics>> components. Optionally, artifacts describing the interaction between mechanic and electronic parts may be added.

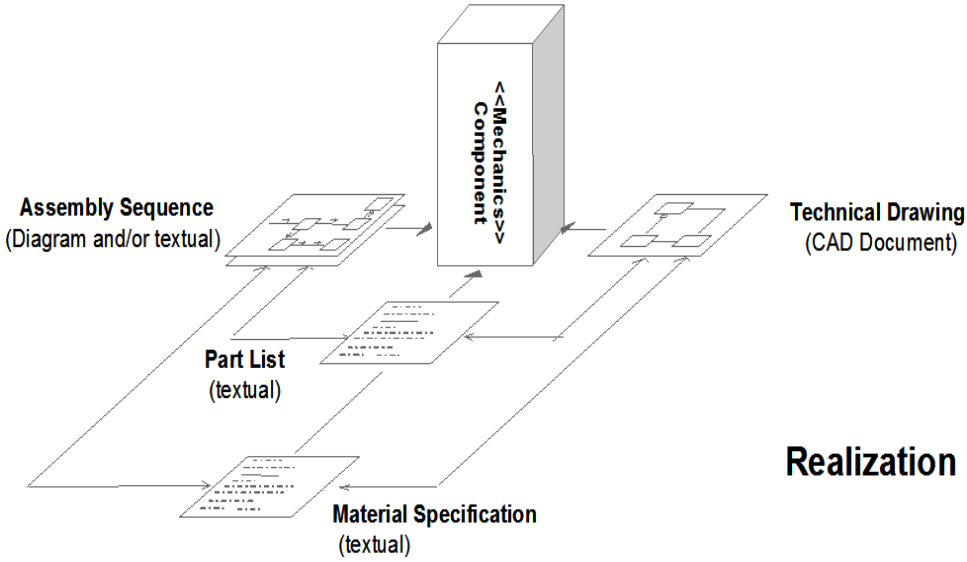


Figure 7 Realization of a <<Mechanics>> Component

3.3 Methodological Support

By now it is defined how software and hardware components of an embedded system can be modeled using the UML. However, as mentioned before, the UML like any other notation is just a tool, which needs methodological support for the systematic development of such a system. This is the task of the MARMOT approach. MARMOT (**M**ethod for Component-B**a**sed **R**eal-Ti**m**e **O**bject-oriented Development and **T**esting) [4], is specifically geared towards embedded and real-time system development in an object and component-oriented context. It subsumes the powerful principles of the KobrA method, but provides additional features, that are particularly important in embedded, real-time application construction. MARMOT is based upon the following fundamental principles that are fully in line with the KobrA method's meta-model:

- MARMOT is completely based upon the KobrA method, and fully subsumes all of the method's principles and artifacts: The MARMOT component meta-model simply extends the KobrA method component meta-model.
- Software and hardware components are treated in the same logical way. Hardware components are defined through KobrA specifications. Only the

realizations define how the KobrA specification is implemented in terms of a HW/SW co-design.

- MARMOT is inherently aspect-oriented, it means that it supports a complete embedded system to be entirely considered from a particular perspective or point of view. Such perspectives comprise HW/SW views, performance/efficiency views, response time views, safety views, etc.
- Real-time or response-time specifications are typically derived from high-level user requirements, where a user may be a human role, the natural environment, or another associated system. MARMOT provides an iterative approach to system development and testing with real-time requirements, based on dynamic timing analysis, for non-critical real-time systems, and a combination of static and dynamic timing analysis approaches for critical systems.
- Safety requirements are considered right from the most initial development phases. Whereas traditional hazard analysis techniques are only applied at lower level abstractions (i.e. at component level) and are typically performed in a bottom up fashion. MARMOT provides a top down approach to safety requirements through its in-built tracing facilities from user-level abstraction down to concrete designs. With that respect, MARMOT brings forward the user and the environment as primary subject of safety engineering rather than the system itself as in most traditional approaches.

3.4 Developing Systems with Embedded Components

A MARMOT project is always based upon the following fundamental activities: (1) iteratively decompose the system into finer-grained parts that are individually controllable, this is termed "decomposition", and (2) reduce the level of abstraction to create representations of the system that come closer and closer to executable formats, this is termed "embodiment". Doing this, MARMOT is inherently component-oriented. In other words, every system is organized as a tree-shaped hierarchy of logical building blocks that have class-like and package-like properties. The class-like properties allow a component to have attributes, operations and behavioral features, whereas the package-like properties allow a component to represent a name space and act as container for a wide range of documents, concepts and other components.

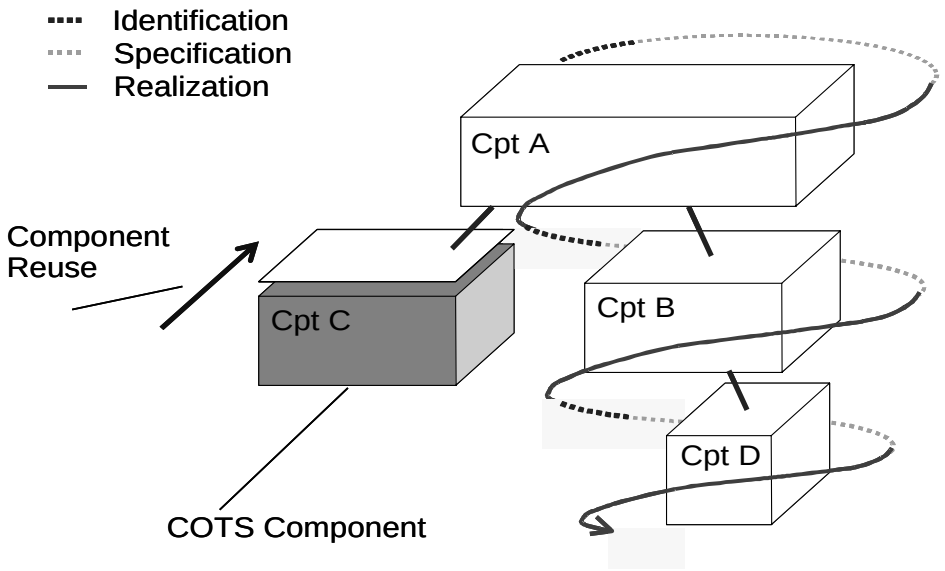


Figure 8 Development Process Overview

As demonstrated by the Cleanroom approach [15], a recursive development process is inherently iterative. Figure 8 shows how the primary component engineering activities of specification and realization, when visualized in connection with the hierarchic product that they generate, can be regarded as leading to a spiral-based process.

The final goal of the component reuse activities (see Figure 8) is to fully integrate a component that has been developed earlier outside the context of the tree (i.e., an external component). To achieve this, the specification desired of the reusing component and the provided specification offered by the preexisting external component have to be brought into agreement. When such a situation exists, the reused component realizes, and usually also implements the specification that is required by the reusing component, and the reused component is then fully integrated.

3.5 Tool Support

Tool support is critical for the approach presented in this paper. Two main categories of tool support are required: CASE tool support for UML diagram development, and configuration management support for keeping track of the many artifacts developed during a project.

Today no CASE tool provides explicit measures for all the previously discussed issues. Thus, help in diagram development can only be attained by adapting and using existing tools. Therefore a plug-in for Rational Rose (see Figure 9) has

been developed within the institute. It provides an infrastructure for organizing the required diagrams, provides specific modeling elements needed for embedded systems, and, in the future, will provide automatic checking of consistency and correctness rules.

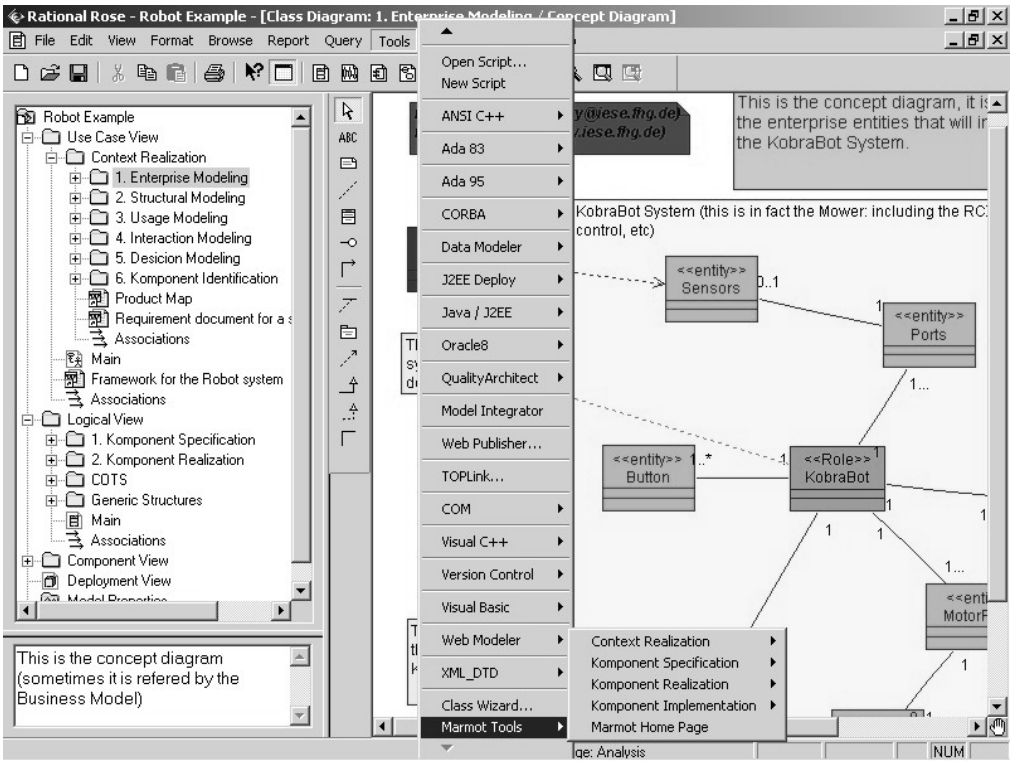


Figure 9 Tool Support

The other category of tool support required in industrial applications, is support for configuration management. This is best based on a repository that is capable of providing full life-cycle support for the various artifacts defined by the method. As part of the KobrA project a prototype component manager and configuration management based on the Enabler Repository was implemented.

4 Case Study

We have developed a case study in order to evaluate and demonstrate the applicability of our approach to embedded system development. This is a small robot or autonomous vehicle (see Figure 10), which has been exhibited at the CeBIT 2003 and CeBIT 2004 fairs in Germany. The robot is based on the LEGO™ MINDSTORMS™ system for robotics that incorporates a small microcontroller. The robot has been modeled through applying MARMOT principles and implemented with Java. The main challenges of the project were the limited number of interfaces (i.e., sensors and actuators have three channels each) and the small memory size (12K for code).

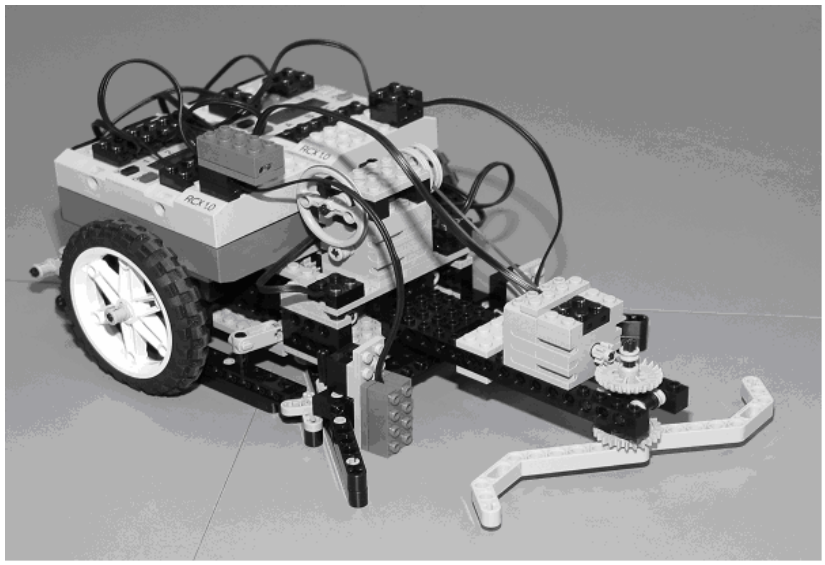
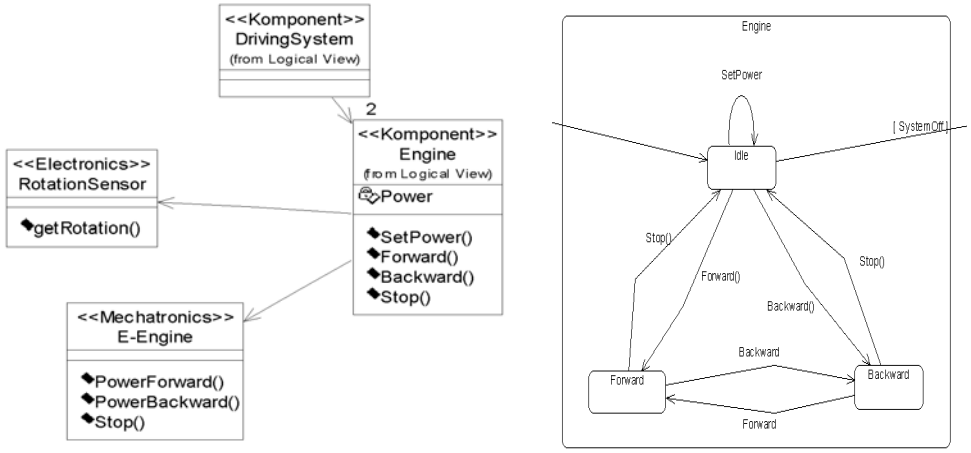


Figure 10 Example Robot

The robot represents a simple form of utility machine with several possible variations (e.g., by changing its arm it can act as a lawn mower or a road sweeper) that are represented by different components that can be easily plugged to the robot core or framework. Based on a number of requirements the robot was modeled using MARMOT and its concepts, as presented by this paper, of how a component should be described. A good example of such a component is the robot's driving system that is responsible for all movements of the robot. Figure 11 shows an excerpt of the components' specifications, whereby Figure 12 shows one of its realization models. These artifacts have been described for every component within the system prior to their implementation in Java.

The driving system is also a good example for the support of integrated Hardware/Software development. The class diagram in Figure 11 shows that the driving system component is dependent upon a <<Mechatronics>> component, an electric engine, in order to move the robot. On the specification level, the engine component can be described as any other MARMOT component since it offers specific functionality to the system (i.e., the engine can rotate forward and backward and can stop any movements). However, at the realization level we use a COTS component (i.e., a standard engine) that fulfills this interface.

One result of the case study was that the complexity of the system has been reduced while the maintainability and reusability has been increased, although this is a subjective observation. One of the major goal for future activities is to evaluate the benefits of the presented approach in the form of an empirical study which will also provide objective measures.



SetPower

Name	SetPower()
Description	The engine can have seven different rotation speeds. This operation sets the speed with which all rotations are performed.
Receives	Speed:Int
Sends	-
Rules	Valid Speeds are integers ranging from 0 to 7
Changes	Engine.Power
Assumes	The Engine is stopped
Result	The attribute Power is set to the given speed

Figure 11 Diagrams for the Robot

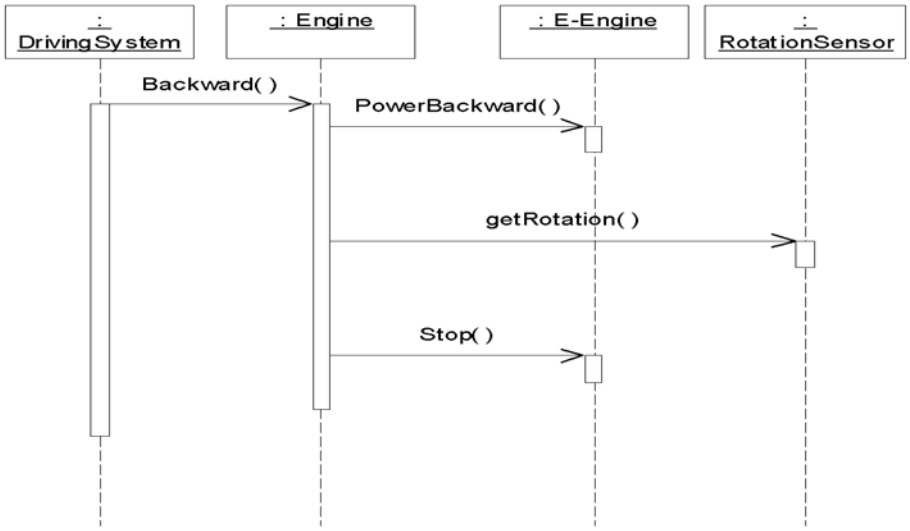


Figure 12 Driving System - Interaction Diagram

5 Conclusions and Future Work

The phenomenal interest in the Unified Modeling Language provides a unique opportunity to increase the amount of modeling work performed in the software development industry, and through this to increase quality standards. UML 2.0 promises new opportunities to apply object-oriented and model-based development in the development of embedded systems. However, this chance will be lost if developers are given no effective and practical means for handling the complexity of such systems and guidelines for systematically applying them.

This paper has outlined the UML modeling practices which are needed in order to fully leverage the component paradigm in the development of embedded software. Following the principles of encapsulation and uniformity - separating the description of what a system unit does (e.g., “specification”, “interface” and “signature”) from the description of how it does it (e.g., “realization”, “design”, “architecture”, “body”, and “implementation”) and describing both levels with a standard set of models – it is possible to uniformly model the hardware and software components of an embedded system. This facilitates also a "divide and conquer" approach to modeling in which a system unit can be developed independently. It also allows new versions of a unit to be interchanged with old versions provided that they do the same thing. The MARMOT method and its accompanying tool support this approach to modeling by providing embedded system developers with step-by-step guidelines throughout a complete development project.

We have already applied the introduced concepts in a small case study, an autonomous vehicle that has been presented at the 2003 and 2004 CeBIT exhibitions in Germany. The goal of this study is to demonstrate the feasibility of a fully model-driven and component-based approach to embedded system design. Unfortunately, exact numbers and measures to document success in an objective way have not been collected yet. Hence, it was not possible to set up a measurement program to empirically prove our statements and experiences. In future applications we will set up a small measurement program to collect data and empirically validate our goals and thus enable companies to repeat the results we retrieved in their context systematically and successfully

References

1. C. Atkinson, J. Bayer, C. Bunse, et al. Component-based Product Line Engineering with UML. Addison Wesley, 2002.

2. C. Atkinson, C. Bunse, E. Kamsties, J.Zettel. Principles for UML-based Component Modeling. In The Development of Component-Based Information Systems. M.E. Sharpe Inc., 2004 (to appear).

3. M. Broy and R. Sandner. InTime methological founded development of real-time systems. http://wwwbroy.informatik.tu-muenchen.de/proj/intime/intime.html, 2002.

4. C. Bunse, H.G. Gross, N. Mayer. Developing Embedded Real-Time Applications Systematically: The MARMOT Approach. Submitted to the Journal on Software and Systems Modeling (SoSym), 2003

5. J. Cheesman, and J. Daniels. UML Components: A simple Process for Specifying Component-Based Software. Addison-Wesley, 2000.

6. D. Coleman, P. Arnold, S. Bodoff, et al. Object-Oriented Development: The Fusion Method. Prentice Hall, 1993.

7. P.J. Courtois, D.L. Parnas. Documentation for safety critical software. In Proceedings of the Fifteenth International Conference on Software Engineering, 1993.

8. W. Damm. Use-case driven specification of engineering applications, 2002.

9. B.P. Douglass. Real-Time UML: Developing Efficient Objects for Embedded Systems. Addison-Wesley, 1999.

10. D. D'Souza, A. Wills. CATALYSIS practical rigor and refinement - extending OMT, fusion, and objectory. Technical report, ICON Computing Inc., 1995.

11. H. Ehrig, E. Westkämper. IOSIP. http://tfs.cs.tu-berlin.de/SPP/iosip.html, 2002.

12. P. Kruchten. The Rational Unified Process. An Introduction. Object Technology Series. Addison-Wesley, 1998.

13. K. Lano. Formal Object-Oriented Development. Springer, 1995

14. G. Martin, L. Lavagno, J.L.Guerin. Embedded UML: a merger of real-time UML and co-design. Unpublished (avail. at http://www.gigascale.org/pubs/101.html), 2001.

15. H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet. Principles of Computer Programming: A Mathematical Approach. Allyn and Bacon Inc., 1987.

16. A. Mitschele-Thiel. Systems Engineering with SDL. Addison-Wesley, 2001

17. Object Management Group. OMG unified modeling language specification, version 1.5. Technical report, OMG, 2001

18. Object Management Group. OMG unified modeling language specification, version 2.0. Technical report, OMG, to appear

19. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. Object-Oriented Modeling and Design. Prentice Hall, 1991.

20. B. Selic. Turning clockwise: Using UML in the real-time domain. Communications of the ACM, pages pp 46–54, October 1999.

21. B. Selic, G. Gullekson, and P.T. Ward. Real-Time Object-Oriented Modeling. John Wiley & Sons, 1994.

22. C. Szysperski. Component Software: Beyond Object-Oriented Programming. Addisson-Wesley, 1999

23. Telelogic. SDT 3.2 Methodology Guidelines – Part1: The SOMT Method. Telelogic, 1997.

24. S. Terrier. Intensive use of uml model transformation: the accord environment. In Proceedings of the Workshop on Transformations in the Unified Modelling Language (WTUML01), 2001. satellite event to ETAPS2001.

Document Information

Title:	UML-based Development of Embedded Software Systems
Date:	February
Report:	IESE-026.04/E
Status:	Final
Distribution:	Public