Delft University of Technology
Software Engineering Research Group
Technical Report Series

Embedded System Construction – Evaluation of a Model-Driven and Component-Based Develpoment Approach

Christian Bunse, Hans-Gerhard Gross, Christian Peper

Report TUD-SERG-2008-020

TUD-SERG-2008-020

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Embedded System Construction – Evaluation of a Model-Driven and Component-Based Development Approach

Christian Bunse[1], Hans-Gerhard Gross[2], and Christian Peper[3]

[1] International University, Bruchsal, Germany
Christian.Bunse@i-u.de
[2] Delft University of Technology, Delft, The Netherlands
h.g.gross@tudelft.nl
[3] Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany
Christian.Peper@iese.fraunhofer.de

Abstract. *Model-driven development, using the UML, has become an important development paradigm. It is said to have many advantages over traditional approaches, even for embedded systems. Along a similar line of argumentation, component-based software engineering is advocated. In order to investigate these claims, the model-driven and component-based method MARMOT was applied to develop and implement several variants of a small micro-controller based automotive subsystem, an electronic car mirror. Several key figures, like model size and development effort, were measured and compared to the outcome of two different approaches: the Unified Process and Agile Development. The analysis reveals that model-based, component-oriented development performs well and leads to adaptable systems and a higher-than-normal reuse rate. This is the case at least for the considered application domain.*

1 Introduction

Embedded software applications are typically more difficult to design and build because of the problem domain and the constraints placed on them by the target environment. One technique that may, at first, seem inappropriate for the embedded domain is modeling and Model-Driven Development (MDD) with components. Modeling is frequently used in other engineering domains as a way to solve problems at a higher level of abstraction and to verify design decisions before committing large resources to develop a product. Component-oriented development envisions that new software applications can be created with much less effort than in traditional approaches, simply by assembling existing parts. Although, the use of models and components for embedded software systems is still far from being industrial best practice.

One reason might be, that the disciplines involved, mechanical-, electronic-, and software engineering, are not in sync, a fact which cannot be attributed to one of these fields alone. Engineers are struggling hard to master the pitfalls of modern, complex embedded systems. What is really lacking is a vehicle to transport the recent advances in software engineering and component technologies to the embedded world.

This paper shortly introduces the MARMOT system development method. MARMOT stands for ‘Method for Component-Based Real-Time Object-Oriented Development and Testing’ and it aims to provide the ingredients to master the multi-disciplinary effort of developing embedded systems. It provides templates, models and guidelines for the products describing a system, and how these artifacts are built up throughout development. The main focus of the paper is on a series of case-studies in which we apply MARMOT to a small control system for an exterior car mirror. In a second run MARMOT is evaluated against other development approaches. In order to validate the expected effects, several aspects such as model size [22] and development effort are quantified and analyzed.

The paper is structured as follows: Section 2 gives an overview of related work. Section 3 briefly describes MARMOT, and section 4 presents the case study and Section 5 discusses the results obtained. Finally, Section 6 presents a brief summary, conclusions drawn, and hypotheses for future research.

2 Related Work

Growing complexity and short release cycles of embedded systems stimulated the transfer of model-driven development techniques to the domain of embedded software systems. There are two research routes: Formal modeling languages for embedded system design, and non-formal approaches using standard notations such as UML. Initially, formal languages such as Z [12], functional decomposition [17], or state-based notations [8] were used, but these approaches lack reuse mechanisms on higher levels of abstraction. Newer developments such as MATLAB [16] or MODELICA [7] provide tool and (additional) methodological support, but lack effective reuse strategies and adaptation mechanisms.

The Unified Modeling Language (UML) [18] was adapted for modeling embedded and real-time systems, but it still lacks precise semantics, and guidelines about its usage. OMEGA [9], HIDOORS [21], or FLEXICON [14], or the work presented in, [6], [10], [13], [15], or [19] define development methods for real-time and embedded systems using the UML. Although a step in the right direction, they often do not use the enhanced features of UML 2.0, nor do they address complexity and reuse issues. Another problem, stated by Kahn, is the inadequate support for mapping UML (2.0) models to code [11]. Developers follow traditional approaches, and a complete transition to object- or component technology is prevented by the required time and space efficiency of the product, or due to standards to be followed. Embedded system development would benefit from the advantages of model-driven development (MDD) [11], if the technologies could be integrated into existing development processes, for example, through keeping C as target language. Most approaches and tools map models to sophisticated languages, for example Java, resulting in runtime performance, memory, or timing problems [11], or they use straightforward mapping strategies (UML to C) that neglect concepts such as inheritance or dynamic binding.

3 MARMOT Overview

Reuse is a key factor in industry, and it can be seen as a major driving force in hardware and software development. Reuse is pushed forward mainly by the growing complexity of systems. This section shortly introduces the MARMOT development method for model-driven and component-based development of embedded systems, which is specifically geared toward facilitating reuse in embedded systems development. MARMOT builds on the principles of the KobrA Method [1], and extends this with techniques for dealing with developing embedded systems. MARMOT applies the key ideas of component-based software development, namely information hiding and divide-and-conquer, even in the software models. Thus, MARMOT components follow the principles of encapsulation, modularity and unique identity that most component definitions put forward.

In MARMOT, Communication between components relies on interface contracts becoming feasible in the hardware or embedded world through software abstractions. An additional hardware wrapper can realize that the hardware communication protocol is translated into a typical component communication contract. Further, encapsulation requires separating the description of what a software unit does from the description of how it does it. These descriptions are called specification and realization, respectively (see Fig. 1).

The specification is a suite of descriptive artifacts that collectively define everything externally knowable about a component. These descriptions fully specify a component in a way that it can be assembled in a system and used by the system. The realization is a suite of descriptive artifacts that collectively define how a component is internally realized. Following this principle, each component within a system can be described through a suite of models, for example UML diagrams or other textual documents, as if it was an independent system in its own right.

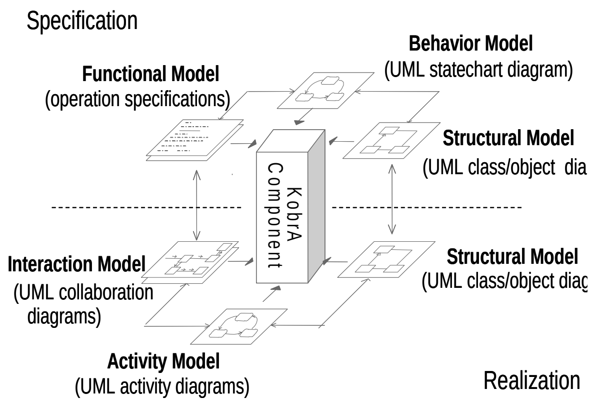


Fig. 1. MARMOT component model.

The fact that components can be realized using other components, turns a MARMOT project into a tree-shaped structure with consecutively nested abstract component representations. Therefore, a system can be 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. Such a tree is called containment tree. Any component can be a containment tree in its own right, and, as a consequence, another MARMOT project. Acquisition of component services across the tree turns a MARMOT project into a graph. The four basic activities of a MARMOT development process are composition, decomposition, embodiment, and validation as shown in Fig. 2.

Decomposition follows the divide-and-conquer paradigm, and it is performed to subdivide the system into smaller parts that are easier to understand and control. An embedded system development project always starts above the top left-hand side box in Fig. 2. The box represents the entire system to be built. Prior to specifying the box, the concepts of the domain in which the system is supposed to operate have to be determined. This comprises descriptions of all relevant domain entities including standard hardware components that will eventually appear on the right-hand side towards concretization. In embedded systems, these implementation-specific entities often determine the way in which a system is divided into smaller parts. During decomposition, newly identified logical parts of the system are mapped to existing components. Whether these are hard- or software does not play a role because all components are treated in a uniform way, as software abstractions.

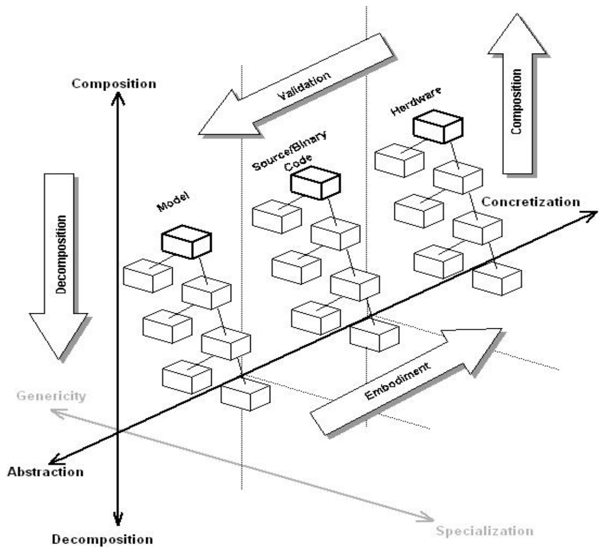


Fig. 2. Development activities in MARMOT.

Composition represents the opposite activity, which is performed when individual components have been implemented or reused, and the system is put together. After having implemented some of the boxes and having some others reused, the system can be assembled according to the abstract model. Therefore, the subordinate boxes with their respective super-ordinate boxes have to be coordinated in a way that exactly follows the component description standard previously introduced.

Embodiment is concerned with the implementation of a system and a move toward more and more executable representations. It turns the abstract system represented by models into more concrete representations that can be executed by a computer. During decomposition, the shapes of each identified individual component were defined in an abstract and logical way. The system parts can, then, be moved towards more concrete representations. This means they become platform specific. Embodiment is where MARMOT adds the most novel concepts to the KobrA development method.

Validation checks whether the concrete representations are in line with the abstract ones. It is carried out in order to check whether the concrete composition of the embedded system corresponds to its abstract description.

The described approach was designed to facilitate the interchange of new component versions with old versions provided that they do the same thing and abide by the same interface, and to foster reuse by directly supporting the typical reuse activities.

– Development for reuse, which deals with how components have to be specified and treated, so that they can be reused.

– Development with reuse, dealing with the integration and adaptation of existing components in a new application.

A more thorough introduction of MARMOT can be found in [3]. In the following section, we describe the system that was built with the help of MARMOT.

4 Description of the Case Study

We used an exterior mirror control system in our case study. It allows moving the mirror horizontally and vertically into the desired position. Mirror positions can be stored and recalled to support driver profiles. For brevity of illustration a simplified version was used, comprising a microcontroller, a button, two potentiometers, and two servos. The system controls the two servos via two potentiometers, and indicates movement on a small LCD panel.

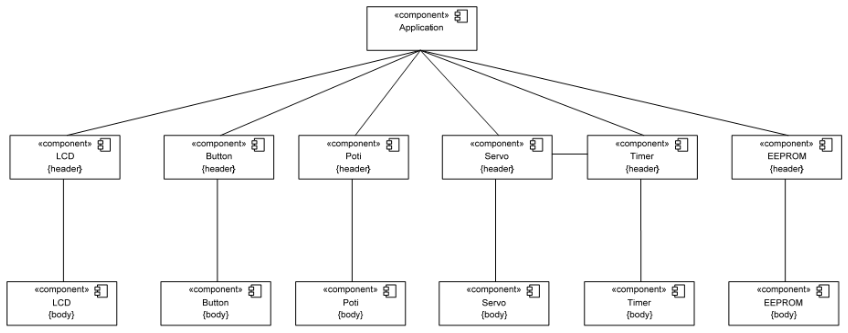
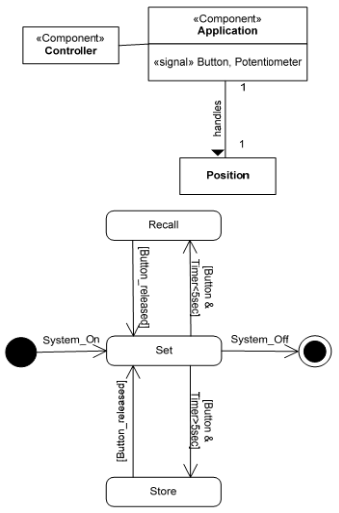

«component» Application
«component» LCD {header}
«component» Button {header}
«component» Poti {header}
«component» Servo {header}
«component» Timer {header}
«component» EEPROM {header}
«component» LCD {body}
«component» Button {body}
«component» Poti {body}
«component» Servo {body}
«component» Timer {body}
«component» EEPROM {body}

Fig. 3. Containment hierarchy (architecture).

Values are read from the potentiometers, converted to degrees, and servo control signals (PWM) are generated, while, at the same time, movement and degree are displayed on the LCD panel. The system stores a position through pressing the button for more than 5 seconds.

Requirements Modeling: Use cases describe the requirements in a textual and a graphical representation. Activity diagrams describe the general flow of control, including a UML representation of the target platform.

Component Modeling: Component modeling creates the specification and realization of all software components using class, state, interaction, and activity diagrams, as well as operation schemata. Since timing is critical in embedded systems, the component realization is extended by timing diagrams. Modeling starts at the root of the containment hierarchy (see Fig. 3) that represents the component structure/architecture of the system. The top-level component is specified using three different UML models. The component specification is further decomposed into the component realization comprising the private design of the component. Figure 4 depicts some of the specification models of the Application component, whereas in Fig. 5, the related realization models are displayed. The models are devised for every component in the containment hierarchy.



Name	Main
Description	The turning degree of two servos (i.e., for controlling the vertical and horizontal aptitude of an exterior mirror) is controlled by means of two potentiometers. In addition by the press of a button the current position (turning degrees) can be persistently stored, if the button is pressed for more than 5. If the button is pressed shorter than 5 sec the turning degrees of both servos is set to the stored position.
Constraints	--
Receives	• Signals are received from the Button and the Potentiometer. • In addition, the System_On and System_Off signals are received from the microcontroller.
Returns	Controls the Servo Position and prints messages on a display (see Use Cases).
Sends	Operations calls to the *Driver* component concerning • LCD (init, write) • Servo (init, set) • Potentiometer (init, send) • Button (init) • EEPROM (store, retrieve) • Timer (start, stop)
Reads	• The ADC value of the Potentiometer (in *Driver*). • Stored data within the EEPROM (in *Driver*).
Changes	• The turning degree of both servos (in *Driver*). • Stored position data in the EEPROM (in *Driver*)
Rules	• Pressing the Button for less than 5 sec will move the servos to the stored position • Pressing the Button for more than 5 sec will store the actual position as the default position
Assumes	--
Result	The vertical and horizontal aptitudes are controlled, whereby positions are stored and retrieved.

Fig. 4. Example component specification.

Implementation: Iteratively devising specifications and realizations is continued until an existing component is found, or, until it can be implemented. Coming to a concrete implementation from models requires reducing the level of abstraction. First, the containment hierarchy is simplified according to technical restrictions of the used implementation technology. Second, the models are mapped to source code, either through a code generator, or manual mapping 0 .

In the context of several follow-up projects (see section 4.1) the system structure (the component architecture, see Fig. 3) enabled systematic reuse. The encapsulation of hardware-related issues in a driver component facilitated the porting of the systems to other hardware platforms. Adaptations were supported by simply removing or adding new components. Furthermore, the specified component interface of the overall system allowed easy reuse in the context of another project.

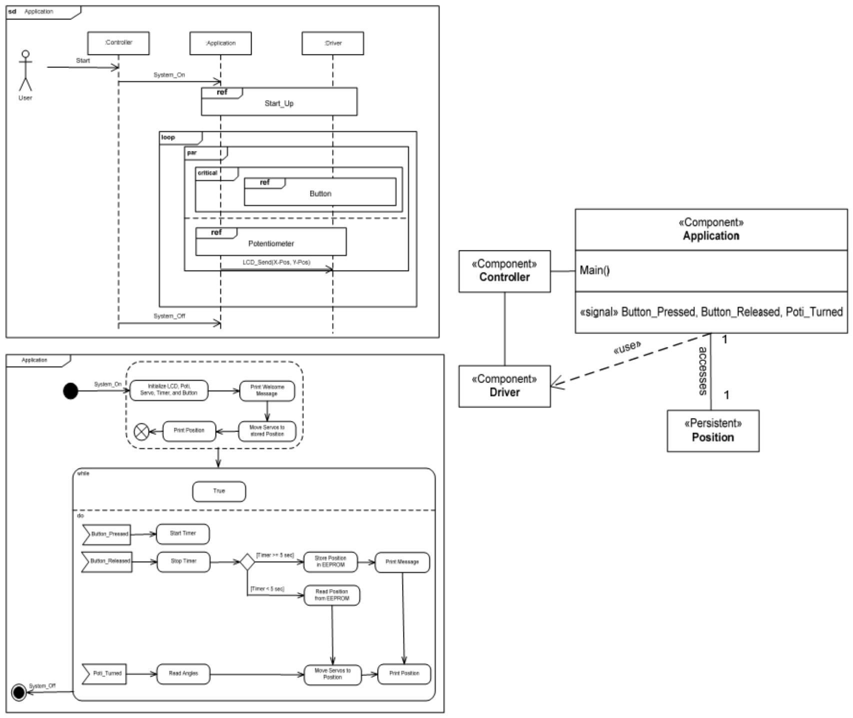


Fig. 5. Example component realization.

Follow-Up Projects: Since the effects of reuse can only be measured and analyzed in follow-up projects, a number of student projects, using the original mirror system as a basis, were defined and carried out in two series. The goal of the projects was to cover typical reuse situations: The system was ported for different processor families, adapted by removing and adding functionality, and it was reused in the context of a larger project. In the second run, the projects of the first run were reiterated using an agile approach [10] and an adaptation of the Unified Process for embedded systems [3]. The goal was compare MARMOT with two other methods.

5 Evaluation and Comparison

In the context of the first and second series of case studies, a number of measurements were performed in order to get a first impression on the maintainability, portability, and adaptability of software systems. Tables 1, 2, and 3 provide data concerning model and code size, quality, effort, and reuse rates. In detail, the following data was collected:

Table 1 Case-Study Results – 1st Series

		Original	ATMega32	PICF	Adapt-	Adapt+	Door
LOC		310	310	320	280	350	490
Model Size (Abs.)	NCM	8	8	8	6	10	10
	NCOM	15	15	15	11	19	29
	ND	46	46	46	33	52	64
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	1	1	1	1	0.8	1
	$\frac{NumberofOperations}{NumberofClasses}$	3.25	3.25	3.25	2.5	3	3.4
	$\frac{NumberofAssociations}{NumberofClasses}$	1.375	1.375	1.375	1.33	1.3	1.6
Reuse	Reuse Fraction(%)	0	100	97	100	89	60
	New (%)	100	0	3	0	11	40
	Unchanged (%)	0	95	86	75	90	95
	Changed (%)	0	5	14	5	10	5
	Removed (%)	0	0	0	20	0	40
Effort (h)	Global	26	6	10.5	3	10	24
	Hardware	10	2	4	0.5	2	8
	Requirements	1	0	0	0.5	1	2
	Design	9.5	0.5	1	0.5	5	6
	Implementation	3	1	3	0.5	2	4
	Test	2.5	2.5	2.5	1	2	4
Quality	Defect Density	9	0	2	0	3	4

Table 2 Case Study Results – 2nd Series (Agile)

		Original	ATMega32	ATMega32	PICF	PICF	Adapt-	Adapt-	Adapt+	Adapt+	Door	Door
LOC		280	280	290	300	340	270	300	310	330	450	550
Model Size (Abs.)	NCM	14	14	15	14	15	11	13	16	17	23	26
	NCOM	5	5	5	5	5	4	4	7	7	12	12
	ND	3	3	3	3	3	3	3	3	3	3	3
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	0	0	0	0	0	0	0	0	0	0	0
	$\frac{NumberofOperations}{NumberofClasses}$	3.21	3.21	3.3	3.21	3.3	3.18	3.15	3.18	3.23	4.08	4.19
	$\frac{NumberofAssociations}{NumberofClasses}$	3.5	3.5	3.3	3.5	3.3	3.54	3.46	3.3	3.17	2.69	2.57
Reuse	Reuse Fraction(%)	0	100	95	95	93	100	93	50	45	30	25
	New (%)	100	0	5	5	7	0	7	50	55	70	75
	Unchanged (%)	0	90	85	89	75	65	40	70	54	90	85
	Changed (%)	0	10	14	8	15	15	40	20	36	10	10
	Removed (%)	0	0	1	3	10	20	20	10	10	0	5
Effort (h)	Global	18	4	5	9	11.5	3	6	11	13.5	29	37
	Hardware	6	2	2	4	4	0.5	1	2	2	8	8
	Requirements	0.5	0	0	0	0	0	0.5	1	1	1	1
	Design	2	0	0	0	0	0.5	1	1	1.5	3	3
	Implementation	7	1.5	2	3	5	1	2	4	6	11	18
	Test	2.5	0.5	1	2	2.5	1	1.5	3	3	6	7
Quality	Defect Density	7	0	0	1	2	1	1	4	5	7	7

Table 3 Case Study Results – 2nd Series (Unified Process)

		Original	ATMega32	ATMega32	PICF	PICF	Adapt-	Adapt-	Adapt+	Adapt+	Door	Door
LOC		350	350	340	340	340	300	320	370	400	470	500
Model Size (Abs.)	NCM	10	10	10	10	10	7	8	12	12	12	13
	NCOM	15	15	15	15	15	11	11	19	19	29	29
	ND	59	59	59	59	59	43	45	60	60	66	68
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	1.5	1.5	1.5	1.5	1.5	0.7	0.72	1.33	1.33	1.16	1.07
	$\frac{NumberofOperations}{NumberofClasses}$	4	4	3.5	3.5	3.5	3	3.25	2.83	3	3.5	3.46
	$\frac{NumberofAssociations}{NumberofClasses}$	2.5	2.5	2.3	2.5	2.3	2.57	2.5	2	2.16	1.83	1.76
Reuse	Reuse Fraction(%)	0	100	100	95	94	91	88	87	86	59	58
	New (%)	100	0	0	5	6	9	11	13	14	41	42
	Unchanged (%)	0	96	92	84	80	73	70	88	85	90	86
	Changed (%)	0	4	4	16	15	6	6	12	15	10	14
	Removed (%)	0	0	4	0	5	21	24	0	0	41	41
Effort (h)	Global	34	7	8	11	12	5	5.5	11	13	27	29
	Hardware	10	2	2	4	4	0.5	0.5	2	2	8	8
	Requirements	4	1	1	1	1	1.5	1.5	2	3	4	4
	Design	12	1	1	1	2	1	1	5	4	6	7
	Implementation	5	1	2	3	3	1	1.5	2	2	5	6
	Test	3	2	2	2	2	1	1	2	2	4	4
Quality	Defect Density	8	0	1	2	2	1	0	3	3	4	4

- System Size. Memory is a sparse resource in embedded systems (e.g., often below 10 Kbytes), and program size is extremely important. MDD for embedded systems will only be successful if the resulting code size, obtained from the models, is small. Therefore, we collected the lines of code (without pretty printing, comments, etc.) for each system.
- MDD is often believed to create a large overhead of models, even for small projects. Therefore, we collected data concerning the model-size for all systems absolute and relative size measures proposed in [22]. In addition, figures on the number of classes in a model (NCM), number of components in a model (NCOM), and the number of diagrams (ND) were used.
- Reuse is central for MDD and CBD and it must be seen as an upfront investment paying off in many projects. Reuse must be examined between projects and not within a project. Therefore, we collected data concerning the amount of reused elements as the proportion of the system which can be reused without any changes or with small adaptations. Measures are taken at the model and the code level and are normalized using the system size (model, LOC).
- Effort saving is one promise of MDD and CBD [20] though, it does not occur immediately (i.e., in the first project), but in follow-up projects. Effort was measured (in hours) for all development phases to identify where savings could be realized. Since all projects were quite small, development hours were used, coming from daily effort sheets.
- Another promise of MDD and CBD is to support the development of high-quality systems. The vision is that the quality of a system will benefit from reusing “good” components. To evaluate if this effect occurred we collected the defect density, computed per one hundred lines of code, for all systems.

5.1 First Run

It is interesting to see that porting the system to another hardware platform required only minimal changes to the models (e.g., UML hardware representation, ports, etc.). Thus, MARMOT supports the MDA idea of platform independent modeling. Only in the embodiment step models become platform specific. The ease of porting a system to different platforms is also supported by the high amount of reuse with minimal changes, the low effort, and the low number of defects.

Concerning the adaptation of existing systems by adding or removing functionality, the data reveal that MARMOT provides sufficient support. First of all, a large proportion of the systems could be reused from the original system. Second, in comparison to the initial development project (i.e., ‘Original’), the effort for adaptation is low (26hrs vs. 3/10hrs). In addition, the quality of the system profits from the quality assurance activities carried out in the initial component development. Thus, the promises of component-oriented development concerning time-to-market and quality could be confirmed in this case-study.

Interesting to note is that the effort for the initial system corresponds to standardized effort distributions over development phases as used by common cost estimation methods, whereby the effort for the variants is significantly lower. In addition, this supports the assumption that component-oriented development has an effort-saving effect in subsequent projects.

In general, porting and adaptation of a component-based system takes place during developing system variants. Thus, the systems are highly similar, which, in turn, explains why reuse works that well. It would, therefore, be interesting to look at larger systems (of the same domain) that reuse the original system as a whole and/or some of its components. The 'Door'-controller project is such a project. 60% of the overall system was reused in form of the mirror system, which itself did not need major adaptations. Effort- and defect density are higher than those of the mirror system variants, due to the development of new additional components, major hardware extensions, and intensive quality-assurance. Thus, when directly compared to the initial effort and quality (i.e., the mirror system), a positive trend can be seen that supports the assumption that with MARMOT, embedded systems can be quickly developed at a low cost but with high quality.

5.2 Second Run

The second series of case studies (see Table 2 and Table 3) replicated the development projects but used different development methods. Every project was carried out twice for each method by independent teams in order to obtain more data points.

Concerning the limited amount of modeling in the agile approach, at the first glance, it appears that the initial development is truly faster while maintaining high quality. However, this does not hold as soon as it comes to adaptations. The more complex an adaptation becomes, the more grows the effort. In the worst case, the effort is much higher than in the model-based approaches, and it is even comparable to the effort for developing the system entirely from scratch. This is, especially, true in cases where the adaptation is not performed by the initial author (i.e., more adapter classes and workarounds were used). One reason might be the low reuse rate and missing documentation. However, in general, the final source-code seems to be of a good quality and highly integrated.

The results concerning the application of the Unified Process are largely comparable to those of MARMOT. However, it seems that using the Unified Process requires more overhead and documentation (e.g., number of diagrams) resulting in higher development effort. Ironically, documentation and model-size seemed to have a negative impact on quality, since defect density was quite high. This might also be the reason that adaptations needed more effort than in a MARMOT-based development, although this method even outperformed agile development with respect to complex adaptations, especially if different authors were involved.

6 Summary and Conclusions

The presented experiments show that the promises of component-oriented development concerning reuse, effort, and quality can be achieved in the context of embedded system development. However, similar to product-line engineering projects, CBSE requires an upfront investment before paying-off.

There are some threats to the validity of the ob-served results which may hinder their generalization. First, the people participating in this study were students that are not representative for software professionals. However, the results may be useful in an industrial context, since engineers in industry often have no more experience in UML-based development than students. Introducing such methodological support requires a steep training curve, even in professional organizations. Second, the use of volunteers may affect the validity of the study (i.e., selection bias). Individuals who volunteer for an activity are almost certainly different from those who do not volunteer. Volunteers are, by definition, motivated to participate and presumably expect to receive some benefit from the intervention, whereas employees often have a negative attitude towards new technologies. These differences between study participants and people in real organizations limit the ability to generalize the results beyond the research example. Finally, the systems developed in the scope of this paper are not representative in terms of their size and complexity. However, the results can be used as a trend indicating possible benefits and, therefore, encourage the performance of more industrial-scale case-studies.

The growing interest in the Unified Modeling Language provides a unique opportunity to increase the amount of modeling work in software development, and to elevate quality standards. UML 2.0 promises new ways to apply object/component-oriented and model-based development techniques throughout embedded systems engineering. However, this chance will be lost, if developers are not given effective and practical means for handling the complexity of such systems, and guidelines for applying them in a systematic way.

This paper outlined the UML modeling practices, which are needed in order to fully leverage the component paradigm in the development of software for embedded systems. Following the principles of encapsulation and uniformity, and describing both levels with a standard set of models – it becomes feasible to model hardware and software components of an embedded system with UML. 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.

To validate MARMOT, a series of case-studies has been performed. Quantitative and qualitative results of these studies indicate that MARMOT supports systematic reuse and thereby reduces development effort, and improves the quality of a software system. However, these results are only a starting point for more elaborate validation and generalization of the results. Therefore, a controlled experiment with a larger system is currently planned in order to obtain more objective data. A further step towards enabling technology will be the provision of tools.

References

[1] Atkinson, C., Bayer, J., Bunse, C., and others. Component-Based Product-Line Engineering with UML, Addison-Wesley, UK, 2001.

[2] Bunse, C., Gross, H.-G., Peper, C., Applying a Model-based Approach for Embedded System Development, Proc. of the 33rd Conference on Software Engineering and Advanced Applications (SEAA), Lübeck, Germany, 2007.

[3] Bunse, C., Gross, H.-G., Unifying Hardware and Software Components for Embedded System Development, In: Architecting Systems with Trustworthy Components, Reussner, Staffort, Szyperski (Eds), Lecture Notes in Computer Science, Vol. 3938, Springer, Heidelberg, 2006.

[4] Cantor, M., Rational Unified Process for Systems Engineering, the Rational Edge e-Zine, 2003, http://www.therationaledge.com/content/aug_03/f_rupse_mc.jsp.

[5] Crnkovic, I., Larsson, M. (Eds.), Building Reliable Component-Based Software Systems, Artech House, 2002.

[6] Douglass, B.P., Real-Time Design Patterns, Addison-Wesley, 2003.

[7] Fritzson, P., Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley, 2004.

[8] Harel, D., Lachover, H., Naamad, A., and others, Statemate: A working environment for the development of complex reactive systems, IEEE TSE, 16(4), April 1990.

[9] J. Hooman, Towards Formal Support for UML-based Development of Embedded Systems, Proc. of the 3rd PROGRESS Workshop on Embedded Systems, Technology Foundation STW, 2002.

[10] Hruschka, P., Rupp, C., Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML, Hanser, 2002.

[11] Khan, M.U., Geihs, K., Gutbrodt, Model-Driven Development of Real-Time Systems with UML 2.0 and C, 3rd Int. Workshop Model-based Methodologies for Pervasive and Embedded Software, 2006.

[12] Lano, K., Formal Object-Oriented Development. Springer, 1995.

[13] Lavagno, L., Martin, G., Selic, B. (Eds.), UML for Real Design of Embedded Real-Time Systems, Kluwer, 2003.

[14] Marcos, M., Estevez, E., Gangoiti, U., and others, UML Modeling of Industrial Distr. Control Systems, Proc. of the 6th Portuguese Conf. on Automatic Control, Portugal, 2004.

[15] Marwedel, P., Embedded System Design, (Updated Version), Springer, 2006.

[16] The MathWorks, Inc., Simulink Reference, 2005, http://www.mathworks.com.

[17] Mills, H.D., Basili, V.R., Gannon, J.D., and others, Principles of Computer Programming: A Mathematical Approach. Allyn and Bacon Inc., 1987.

[18] Object Management Group, UML 2.0 Super-structure Specification, OMG document formal/05-07-04, 2005, http://www.omg.org/cgibin/ doc?formal/05-07-04.

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

[20] Szyperski, J., Component Software. Beyond Object-Oriented Programming, Addison-Wesley, 2002

[21] Ventura, J., Siebert, F., and others, HIDOORS - A High Integrity Distributed Deterministic Java Environment, Proc. of the 7th Int. Workshop on Object-Oriented Real-Time Dependable Systems, USA, 2002

[22] Lange, C.F., Model Size Matters, Workshop on Model Size Metrics, 2006 (co-located with the ACM/IEEE MoDELS/UML Conference); October, 2006.

TUD-SERG-2008-020
ISSN 1872-5392