pISSN 1976-913X
eISSN 2092-805X

J Inf Process Syst, Vol.8, No.4, December 2012 http://dx.doi.org/10.3745/JIPS.2012.8.4.539

Evaluation of an Abstract Component Model for Embedded Systems Development

Christian Bunse*, Yunja Choi** and Hans Gerhard Gross***

Abstract—Model-driven and component-oriented development is increasingly being used in the development of embedded systems. When combined, both paradigms provide several advantages, such as higher reuse rates, and improved system quality. Performing model-driven and component-oriented development should be accompanied by a component model and a method that prescribes how the component model is used. This article provides an overview on the MARMOT method, which consists of an abstract component model and a methodology for the development of embedded systems.
The paper describes a feasibility study that demonstrates MARMOT's capability to alleviate system design, verification, implementation, and reuse. Results indicate that model-driven and component-based development following the MARMOT method outperforms Agile development for embedded systems, leads to maintainable systems, and higher than normal reuse rates.

Keywords—Software Development Management, Software Reusability, Modeling

1. INTRODUCTION

Embedded systems are increasingly being built by following the model-driven and component-based development (CBD) paradigms. The motivation is to move reuse onto levels of abstraction higher than the code level, and to be able to reason about a system without having to compose and implement it. Components should be described in a way that they can be composed independently of a specific underlying runtime platform. Additionally, they should be organized in a way that they can be easily deployed in different execution, and hardware contexts. Model-Driven techniques are often part of an entire development framework, providing a component model, as part of general product and a process models. Example methods are KobrA [2] or the Rational Unified Process (RUP) [23].

In general, a *component model* is a wiring standard [34] provided with an execution environment, and it determines how components are composed and executed. Examples of such component models are J2EE, JavaBeans, CORBA, DCOM, .NET, Koala, etc., and they require specific implementation technologies, such as programming and interface definition languages, and

※ This work was partially supported by National Research Foundation of Korea Grant funded by the Korean government (2010-0017156)
Manuscript received October 6, 2011; first revision June 22, 2012; accepted July 6, 2012.
Corresponding Author: Yunja Choi
* University of Applied Sciences Stralsund, Stralsund, Germany (Christian.bunse@fh-stralsund.de)
** School of Computer Science and Engineering, Kyungpook National University, Daegu, Korea (yuchoi76@knu.ac.kr)
*** Software Engineering Research Group, Delft University of Technology, The Netherlands (h.g.gross@tudelft.nl)

 Copyright ⓒ 2012 KIPS

specific bindings for supporting specific programming languages.

In contrast, this paper makes use of an *abstract component model* in order to abstract concrete bindings, focusing on abstract component descriptions rather than on concrete component implementations. Abstract component models deal with system design at a higher level of abstraction than executable units, and they enable engineers to plan and reason about a system architecture or its behavior, without having to implement it.

This paper presents an evaluation of the MARMOT method and its abstract component model [9] with respect to its capability to support component reuse, and to alleviate verification and implementation/composition. It demonstrates how an abstract component model facilitates early verification and assessment of composition. It also compares the effects of higher-level modeling with the Agile approach often used in the industry. A primary concern of this article is to demonstrate how an abstract component model facilitates the reuse of concepts and models, rather than merely executable assets, so that the abstract system design can be instantiated for verification, and, at the same time, realized in terms of concrete executable components. That way, an abstract component model can be used to detach the system design from the implementation, so that the system can be refined and realized in a range of platforms, using various implementation technologies.

The article is structured as follows: Section 2 gives a short overview on the Agile method used throughout the context of this paper. Section 3 briefly introduces the MARMOT methodology including its abstract component model and its process model. Section 4 outlines a system used as demonstrator case study. Section 5 presents the evaluation of the MARTMOT method in comparison with the Agile development approach. Section 6 presents related work about component-based development methods that apply modeling techniques for system design. Finally, Section 7 summarizes and concludes this paper.

2. AGILE DEVELOPMENT OF EMBEDDED SYSTEMS

In principle, Agile software development is a method that accumulates a set of software development methods, based on iterative and incremental strategies. According to [3], the key of Agile development is that requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. A typical representative of this group is an approach known as Extreme Programming (XP) that was developed by Kent Beck, Ward Cunningham and Ron Jeffries. XP is based on a formal set of rules about how one develops functionality such as defining a test before writing the code or to never design more than is needed to support the code that is written. Other core practices are simple design, pair programming and delivering small releases frequently.

When it comes to embedded and real-time systems, Agile practices are not always the best choice due to the focus on software development and the neglecting of documentation and up-front-design. One solution is hybrid approaches that adapt Agile practices to the development of embedded systems. The approach presented in [18] aims at mitigating this problem by defining a process that is based on XP but that also contains modeling aspects using the UML profile for real-time systems (e.g., regarding timing and precision) as well as activities for specifying hardware-related aspects and requirements (e.g., interfaces, distribution, etc.). In contrast, to other Agile processes, such as XP [3], Agile development for embedded systems has a significant

modeling proportion. It makes use of structural and dynamic UML diagrams in order to address the specific needs of embedded and real-time system. In detail, the method is divided into three stages:

1. The *Product & System Development Phase* organizes all of the activities that are done in preparing the development of a product. The goal is the specification of hardware and software requirements as well as of boundary conditions. Based on these the phase concludes by sketching and deciding upon the initial system architecture. Since the phases address a system as a whole, this phase is typically carried out by joint teams of hardware and software developers.
2. The *Software Development Phase* is exclusively addressing the software parts of the final system. Within this phase software requirements and boundary conditions are specified and are then refined to a functional (system structure) and a technical (integration into larger frameworks, etc.) software architecture.
3. The *Integration, Test, and Deployment Development Phase* focuses on the development of hardware and software artifacts. In contrast to the prior two phases that are largely based onto specification and modeling, this phase aims at mapping those artifacts to physical or software artifacts.

3. MARMOT METHOD

The MARMOT method [9] provides an abstract component model and a process for composing and integrating embedded systems on an abstract level. The MARMOT builds on the principles of KobrA [2]. It assumes KobrA's abstract component model, and extends it towards the development of embedded systems. In contrasts to the more lightweight Agile method, the MARMOT method proposes for the end-to-end use of the Unified Modeling Language (UML) as a modeling notation and it prescribes how models should be ideally used in the development of embedded systems. Whereas in Agile development methods, engineers quickly aim at low-level designs in the form of executable code, the MARMOT method prescribes how high-level designs in the form of UML diagrams, representing different views and concerns of a system, can alleviate the reuse of artifacts in other, related development projects.

The MARMOT method advocates the principles of encapsulation, modularity and unique identity that most component definitions put forward [2, 6]. Component communication relies on interface contracts, which are realized through software abstractions for hardware components. A hardware wrapper turns the hardware communication protocol into a software component communication contract. Encapsulation separates the description of what a software unit does (specification), from how it does it (realization). The specification is a collection of (UML) models that defines the external interface of a component so that it can be assembled into or be used by a system. The realization is a set of models that define a component's realization in terms of subcomponent specifications and implementation. Following these principles, each component is described through a suite of models as if it was an independent system in its own right.

3.1 Abstract Component Model

Each artifact of the abstract component model[1] represents a distinct view on the subject, and

1 In contrast to the standard definition of a component model providing rules about component semantics or intercon-

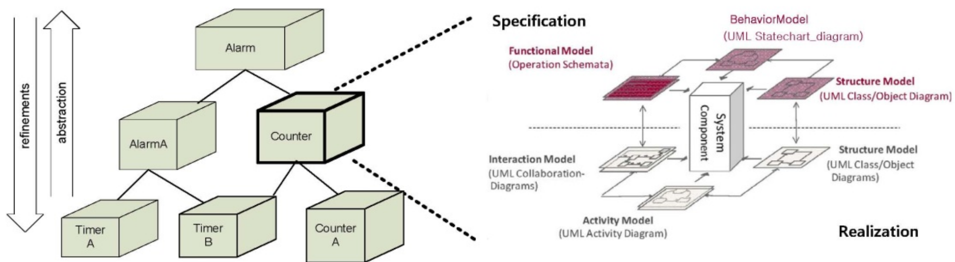

Fig. 1. Conceptual view of MARMOT components

thus only concentrates on a particular aspect of what a component can do, or what it is. Fig. 1 illustrates the concept of abstract components in the MARMOT method with an example. The abstract component *Alarm* may be implemented using the existing abstract components *AlarmA* and *Counter,* which can be implemented using *TimerA*, *TimerB*, and *CounterA*. Each abstract component (e.g., *Counter*) is organized into the two parts of specification and realization, acting as if it is an independent system. The top level abstract component, *Alarm*, is implemented through the successive decomposition and realization process.

Component Specification A specification contains everything necessary to fully use a component and for understanding its externally visible behavior. It defines the provided and required interfaces, including the contract through UML diagrams and other specification elements, such as natural language, or Object Constraint Language (OCL). The specification comprises structural elements (through UML class and object diagrams and other associated components that the subject component requires), its provided and required functionality, and its external behavior (through state diagrams, with pre and post conditions and observable transitions).

Component Realization While the externally visible behavior of a component can be uniquely specified according to the functionality provided and required by the component, there may be various ways of realizing this functionality depending on the design decisions and the available (hardware) components to be used. A component realization is a specification of the design of a component through decomposition and collaboration among the sub-components contained, e.g., underlying runtime middleware, or hardware drivers implementing the communication with hardware components.

Context Realization The context realization is a specific component realization that represents the specification of the physical context in which the system is going to operate. It contains information about the environment in which the system will be embedded, and how the system affects this environment, in terms of sensors and actuators.

3.2 Process Model

At its core, the MARMOT method is based on a tree shaped containment hierarchy with nested abstract component representations. The composite pattern defines that the group of objects

nection rules at the implementation level, an *abstract component model* defines the concepts, rules and constraints that underpin the method, on a meta-model level. [2] provides a detailed overview on the KobrA/MARMOT component meta-models.

shall be treated in the same manner, as a single instance of an object. Within the MARMOT method any component can be a containment tree in its own right (through realization), and any system can be a component within another system. Thus, MARMOT follows a divide-and-conquer strategy by decomposing a system into smaller parts. In the end, a system is represented by a tree of components (containment tree). Whether these components are hardware or software is irrelevant in the abstract component model, since all components are treated in a uniform way. In addition to the top-down orientation of divide-and-conquer strategies, the MARMOT method offers a bottom-up composition to actively support the reuse of components. A system can thus, be assembled according to the containment model.

The final implementation of the system, which involves developing components and integrating them together with reused ones in the final system, is supported in two ways. First, the MARMOT method uses refinement and translation patterns for mapping abstract models to the source-code, following the basic ideas of the MDA paradigm. Second, it supports validation to check whether the concrete representations are in line with the abstract ones. The MARMOT method advocates testing as being the single most important technique for validation [14] and uses model checking for the verification of composition behaviors [9].

3.3 Verification Framework

One goal of verification is the correct coordination of the component containment relationships in order to maintain a correct component containment graph. Every MARMOT component is realized by specifying its internal structure and internal behavior (component realization in the meta-model). The internal structure is reflected in the external structure of its contained sub-components, whose collective behaviors realize the services provided by the super-ordinate component (component specifications in the meta-model). The coordination of the models can then be assessed by model checking tools [31]. The MARMOT iterative verification framework focuses on verifying that subcomponents are coordinated correctly. In this framework, the information contained in the models is extracted using XMI-export, which is provided by modeling tools, is then transformed into the formal language of the model checker SPIN for verification [9].

4. Description of the Case Study

An exterior mirror control system [9] (automotive domain) has been used for our case study to evaluate the usability of the MARMOT abstract component model, and to demonstrate reuse at a higher level of abstraction.

The system is composed of electrical and mechanical components and software control logic, allowing the mirror to be adjusted horizontally and vertically into the desired position. Cars supporting different driver profiles can store and recall the mirror position as soon as the profile is activated. The system is comprised of a microcontroller, a button, two potentiometers, and two servos. The micro controller accesses the two servo-drives via two potentiometers, and shows their movement on a small LCD panel. It reads values from the potentiometers, converts them to degrees, and generates the needed servo control signals.

Figs. 2 to 5 show the primary specification artifacts that have been created according to MARMOT's abstract component model.

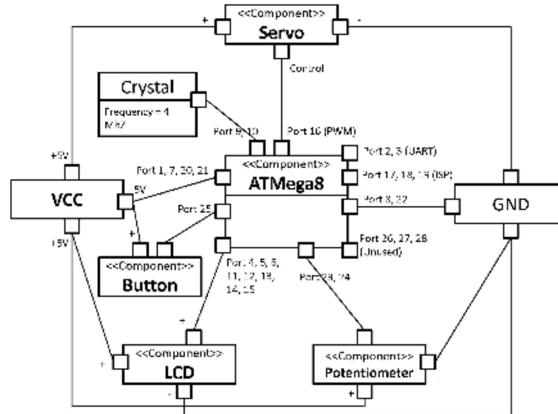

Fig. 2. Model of the deployment platform (ATMega8)

Requirements Modeling Use cases describe the requirements in a graphical (Fig. 3) and a textual representation (not shown). The actor *User* initiates control of the mirror aptitude rotation, and stores and recalls positions through the button. Activity diagrams describe the general flow of control and component diagrams show the UML representation of the target platform, as displayed in Fig. 2.

System Architecture The models shown in Fig. 3 represent part of the *context realization* of the mirror system. The context is like a pseudo component realization at the root of the development tree that embeds the system as a regular component.

Component Modeling Component modeling creates the specification and realization of all software components by using class, state, interaction, and activity diagrams. Modeling starts at the root of the containment hierarchy, and the top-level component is specified by using the various diagrams that the MARMOT method defines in its abstract component model. For example, Fig. 4 shows excerpts from structural and behavioral models of an exterior mirror control application. This top-level abstract component is then successively refined into sub-abstract

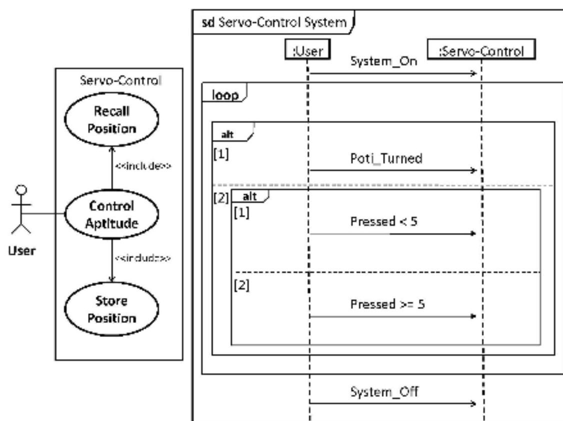

Fig. 3. Context realization use case model and sequence diagram

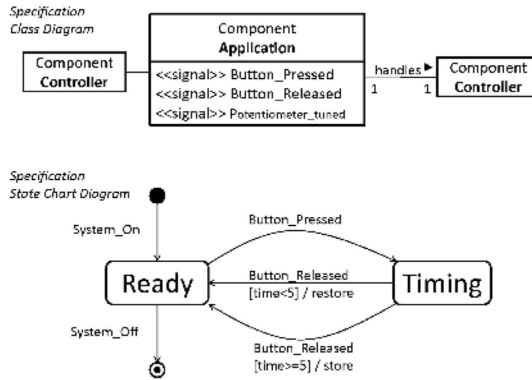

Fig. 4. Structural (class diagram) and behavioral (state diagram) model

components while their interaction consistency is verified at each refinement step. For example, the left side of Fig. 5 displays the structure of the hardware driver component that is used to realize the main component of the exterior car mirror control system. In addition, on the right side of Fig. 5, how the lower-level components interact to achieve the functionality defined in the component specification is shown.

Implementation Iteratively devising specifications and realizations is continued until an existing component is found, thereby targeting existing abstractions (for increasing reuse), or until it can be implemented (no reuse). Coming to a concrete implementation from the models requires the reduction of the level of abstraction of the descriptions. First, the containment hierarchy is simplified according to the technical restrictions of the implementation technology used, i.e., through refinement and mapping it to a UML model with the source code structure of the resulting system. Second, the models are mapped to the source code, either through a code generator, or manually as described in [22].

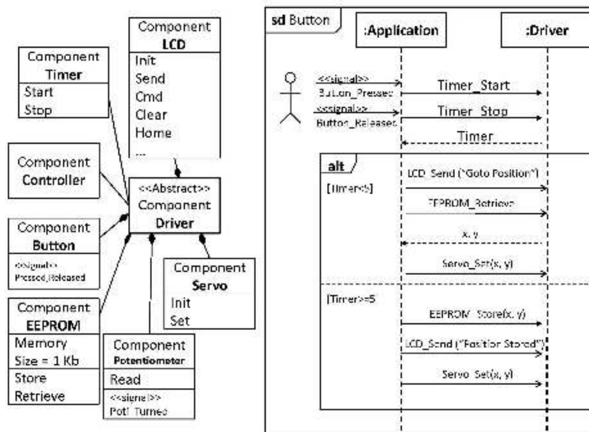

Fig. 5. Realization: structural and interaction model

5. EVALUATION

This section describes an empirical case study for evaluating the efficiency of the abstract component model, w.r.t. composition and reuse, and a qualitative evaluation of the MARMOT formal verification framework w.r.t. reusability and reliability of composition. A more detailed description of this evaluation can be found in [7].

5.1 Empirical Evaluation

The research questions focus on two key sets of properties of model-driven development with components.

1.Questions in the first category are related to the process and decision:
- Q1.1 Which process was used to develop the system?
- Q1.2 Which specification artifacts have been used?
- Q1.3 Are all UML diagram types required, or is there possibly a specific subset sufficient for this domain?
- Q1.4 How are models transferred to the source code?

2.Questions in the second category are more toward specifics of development:
- Q2.1 What is the model-size of the systems?
- Q2.2 What is the defect density of the code?
- Q2.3 How long did it take to develop the systems?
- Q2.4 How is this effort distributed over the requirements, design, implementation, and test phases?
- Q2.5 What is the system footprint?
- Q2.6 How much reuse can be achieved?

Model size follows the metrics as defined in [25]. Effort saving (a promise of MDD and CBD) is measured for all development phases. Memory footprint is considered, which is an important issue for embedded systems, and so is reuse, which is the central property of CBD.

Since it is expected that the benefits of MDD and CBD are only visible during follow-up projects [10], one initial system was specified and used as basis. There were five follow-up projects (R1-R5), namely:

1. *R1/R2*: Ports to different hardware platforms while keeping functionality. Ports were performed within (i.e., ATMega32) and to a different processor family (i.e., PICF). Implementing a port within the same family might be automated at the code level, whereas, a port to a different family might affect the models.
2. *R3/R4*: Evolving system requirements by (1) removing the recall position functionality, and (2) adding a defreeze/defog function with a humidity sensor and a heater.
3. *R5*: The mirror system was reused in a door control unit that incorporates the control of the mirror, power windows, and interior illumination.

In the first experiment[2], the MARMOT method is used, whereby students received an upfront

2 The experiment was performed two times (once for each methodology), whereby every experiment was comprised of several runs (i.e., R1-R5) that represent different typical reuse situations

training in its correct application. The second experiment follows an Agile process (based on Extreme Programming) [18], adapted for embedded software development. Experimental subjects were trained in applying the approach prior to the experiment run. The experimental run was comprised of steps such as preparing system requirements by defining user stories and case studies. This also included the definition of hardware requirements and a specification of the physical context. The next step was modeling the system architecture in order to identify subsystems, etc. Based on requirements and the architecture system, production was started. This composed the modeling of selected system properties (processes, behavior, deployment, etc.), the definition of test-cases and programming.

Subjects were graduate students from the Department of Computer Science at the University of Kaiserslautern (1st experiment) and the School of IT at the International University (2nd experiment). Subjects knew that data would be collected and that an analysis would be performed, but were unaware of the hypotheses that were being tested.

All projects were organized according to typical reuse situations in component-based development, and a number of measures were taken to address the study questions: Model-size was measured with the absolute and relative size measures proposed in [25]. Relative size measures (i.e., ratios of absolute measures) addresses the UML's multi diagram structure and it deals with completeness. Absolute measures are the Number of Classes in a Model (NCM), Number of COmponents in a Model (NCOM), Number of Diagrams (ND), and the normalized LOC for code size. NCOM describes the number of hardware/software components, while NCM represents the number of software components. System size is measured in KBytes of the binary code. The number of reused elements is described as the proportion of the system which can be reused without any changes or with small adaptations (i.e., (re-)configuration but no model change). Defect density is measured in defects per 100 LOC (collected via inspection and testing). Development effort and its distribution over development phases are measured in hours by daily effort sheets.

Overall When reviewing the resulting documentation of the MARMOT method, as well as of the Agile approach, it appears that the amount of modeling naturally differs significantly between these two. However, it shows that both make use of a comparable subset of the UML although the level of detail and refinement differs: Both approaches make use of UML class-, object-, state-, sequence-, collaboration-, component- (package), and timing diagrams while others such as the composite structure diagram remain unused. Thus, as a response to research questions Q1.2 and Q1.3 one immediate conclusion is that the modeling subset for embedded systems might be limited to the aforementioned selection. Interestingly, although the approaches make use of a comparable set of models their strategy for implementing them differs significantly (Q1.4). While the MARMOT method makes use of a code generators and the SORT technique [5], Agile approaches depend on the experience of developers when transferring models in an ad-hoc manner to code. This might also be the reason for the lower overall quality of the Agile system.

The MARMOT method Porting the system (R1) requires only minimal changes to the models, because the MARMOT method supports the idea of platform-independent modeling through the abstract component model of the method (see Fig. 6). Platform specific models are created in the embodiment step. Ports to different processor families (R2) are supported by MARMOT's reuse mechanisms [2]. Concerning the adaptation of existing systems (R3 and R4), data show

		Original	R1	R2	R3	R4	R5
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

Fig. 6. Results following the MARMOT method

that large portions of the system could be reused. Compared with the initial development project, the effort for adaptations is quite low (26hrs vs. 3hrs and 10hrs). The quality of the system profits from the quality assurance activities of the initial project. The promises of CBD concerning time-to-market and quality can be confirmed. Interestingly, the effort for the original system corresponds to standardized effort distributions over development phases, whereby the effort of follow-up projects is significantly lower. This supports the assumption that CBD saves on effort for subsequent projects. Porting and adapting an existing system (R1-R4) implies that the resulting variants are highly similar, which explains why reuse works well. It is interesting to look at larger systems that reuse (components of) the original system (i.e., R5). 60% of the R5-system can be reused without requiring major adaptations of the reused system. Effort and defect density are higher than those for R1-R4, due to the additional functionality and hardware extensions that are required. When directly compared to the initial effort and quality, a positive trend can be seen that supports the assumption that the MARMOT method allows embedded systems development at a low cost but with high quality (Fig. 6).

The Agile method Although, the amount of modeling is limited in the Agile approach (Fig. 7), the original system is developed quickly and with a high quality. This observation does not hold for follow-up projects. Reuse with the Agile approach requires a substantially higher effort than the effort required for the first experiment of applying the MARMOT method, which is attributable to the fact that the development team was different. Due to missing documentation and abstractions, reuse rates were low. It is worth noting that the source-code is of a high quality.

Threats to Validity The authors view this study as exploratory, thereby limiting generalization of the research. Reuse is a problematic concept to measure, but it is argued that the defined metrics are intuitively reasonable. In a single controlled study it is unlikely that every single aspect of that concept can be captured. A maturation effect may be caused by subjects learning as the study proceeds. So, the threat is that subjects learned enough from the single runs to bias their performance in the following ones. An instrumentation effect may result from differences in the materials which may have caused differences in the results. However, this can be addressed by keeping the differences to those caused by the applied method. The subjects were

		Original	R1	R2	R3	*R4*	R5
LOC		280	290	340	300	*330*	550
Model Size (Abs.)	NCM	14	15	15	13	*17*	26
	NCOM	5	5	5	4	*7*	12
	ND	3	3	3	3	*3*	3
Model Size (Rel.)	$\frac{NumberofStateCharts}{NumberofClasses}$	0	0	0	0	*0*	0
	$\frac{NumberofOperations}{NumberofClasses}$	3.21	3.3	3.3	3.15	*3.23*	4.19
	$\frac{NumberofAssociations}{NumberofClasses}$	3.5	3.3	3.3	3.46	*3.17*	2.57
Reuse	Reuse Fraction(%)	0	95	93	93	*45*	25
	New (%)	100	5	7	7	*55*	75
	Unchanged (%)	0	85	75	40	*54*	85
	Changed (%)	0	14	15	40	*36*	10
	Removed (%)	0	1	10	20	*10*	5
Effort (h)	Global	18	5	11.5	6	*13.5*	37
	Hardware	6	2	4	1	*2*	8
	Requirements	0.5	0	0	0.5	*1*	1
	Design	2	0	0	1	*1.5*	3
	Implementation	7	2	5	2	*6*	18
	Test	2.5	1	2.5	1.5	*3*	7
Quality	Defect Density	7	0	2	1	*5*	7

Fig. 7. Results following the Agile method

students, who are unlikely to be representatives for software professionals. However, the results can be useful in an industrial context if one considers that industrial employees often do not have more experiences in MDD and CBD. Laboratory settings allow the investigation of a larger number of hypotheses at a lower cost than field studies. The hypotheses supported in the laboratory setting can be further tested in industrial settings.

5.2 Assessment of Quality Improvement

The MARMOT verification framework and its prototype implementation introduced in [9] is applied to assess the coordination of the mirror control system design in its refinements process. Although it is a long-term project for quantitatively measuring the impact of using the MARMOT verification framework, the 3 major potential improvements that are listed below are anticipated through this case study:

Early detection of problems Fig. 8 illustrates a representative counterexample generated by the SPIN model checker, showing a potential interaction error in the early design model. The counterexample trace shows that the environment of the application model generates signals *button_pressed*, *butten_released*, *poti_tuned*, and *poti_tuned* in that exact order. All of which go through the input channels of the application process. After receiving the *button_released* signal, the application sends the *store* signal to its output channel and waits for the result from its environment. At that moment, the environment sends the *poti_tuned* signal, which cannot be processed by the application since it is busy waiting. Then, the environment tries to send another *poti_tuned* signal which cannot be processed since the channel is already occupied. This is a potential process-deadlock situation where both input and output channels are occupied but cannot proceed. Using the MARMOT verification framework, such communication-related errors are identified before they are implemented.

Reliability Being able to verify the behavioral consistency between a super-ordinate component and its sub-components, the reliability of an abstract component is assured early in the design process. The MARMOT verification framework is currently focused on checking for the

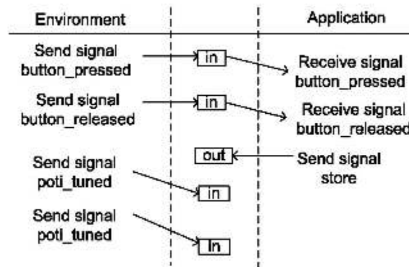

Fig. 8. A counter example trace

absence of the process-deadlock which is a major concern in embedded software. An extension to general property checking is technically possible, but its usability depends on the level of support for property specifications and analysis for verification results.

Reusability The use of the MARMOT verification framework improves the reusability of existing verified abstract components. Once a coordination of an abstraction component with its subcomponents is verified, reuse of the abstract component only requires one time verification between the specification behavior of the abstract component and the realization behavior of its super-ordinate component. This enhances the reliability of reusable components at a low cost.

6. Related Work

6.1 Empirical Studies

This work relies on research that is linked to both quality measurement [15] and empirical software engineering. Good references for empirical software engineering are [21, 36], and our studies were conducted according to common empirical principles, [4, 36]. A detailed description of the chosen empirical approach, experimental design, dependent and independent variables, etc. can be found in [7].

Quality assessment benefited from models such as McCall's, Boehm's or Dromey's. Unfortunately, design- or model-related quality is not explicitly addressed. Software design quality still benefits from its own research. For instance, [4] summarizes many object-oriented design measures and studies the relationships between them and software quality. Nevertheless, we still identified difficulties when conducting empirical studies in software engineering in general, and more particularly in Model-Driven Engineering. This situation even gets worse when it comes to embedded systems [20] that only have a few published studies.

6.2 Model-Driven Development of Embedded Systems

The growing complexity and short release cycles of embedded systems stimulated the transfer of model-driven development techniques to the domain of embedded systems. There are two research routes: formal modeling languages for embedded system design, and non-formal approaches using notations, such as UML. Initially, formal languages such as [26], functional decomposition [32], or state-based notations [16] were used, but these approaches lack reuse mechanisms on higher levels of abstraction. Newer developments such as MATLAB or MODELICA provide tool and (additional) methodological support, but they lack effective reuse strategies and adaptation mechanisms. Recently, the Unified Modeling Language (UML) was

adapted for modeling embedded and real-time systems, but it still lacks precise semantics, and guidelines about its usage. OMEGA [37], HIDOORS [35], FLEXICON [28], or the works presented in [10, 11, 27, 30, 33] define the development methods for real-time and embedded systems using the UML. Although this is 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. In contrast, the MARMOT method fully supports (applies) UML 2.0 and its process and product model (e.g., uniformly and encapsulated components) specifically address the problem of software (component) reuse. Another problem is the inadequate support for mapping UML (2.0) models to code [22] for code generation.

The development of embedded systems would benefit from the advantages of model-driven component-based development as demonstrated in this article if the technologies could be integrated into existing development processes (i.e., keep C as a target language). Most approaches and tools map models to sophisticated languages (e.g., Java, which results in runtime performance, memory, or timing problems [22]), or they use straightforward mapping strategies (UML to C) that neglect concepts such as inheritance or dynamic binding. The MARMOT method uses a pattern-based and MDA-related mapping approach [2] that specifically addresses the relationship between model transformations and non-functional requirements.

There are approaches that aim to combine the two previously mentioned routes. They follow the idea of translating component models to formal specifications so that formal checking can be performed on the translated component models (e.g., [8, 13, 19, 38]). Nevertheless, most existing approaches are limited to language-to-language or artifact-to-artifact translation, and lack the systematic integration of verification activities into the development process. With the MARMOT method, verification is systematically integrated into the process model, due to the iterative nature of MARMOT's abstract component model.

Further, [12] provides a good overview on model-driven development approaches.

7. Summary and Conclusions

Developing software for embedded systems with prefabricated components is appealing, and often, companies have no alternative but to reuse existing assets or they must purchase third party units. The vision of developing once, and reusing often, as well as the possiblility of achieving projected savings in development effort and quality assurance, allows organizations not only to quickly react to changing market requirements but to also manage the inherent complexity of modern systems.

This paper introduced the abstract component model of the MARMOT method, and its process model, and it demonstrated how a meta model supports the reuse of concepts and models on a high level of abstraction. The advantage of using an abstract component is that system specification and design can be devised in models and then instantiated in different formalisms according to the requirements of a project. These formalisms can be specific component models of execution platforms, distinct programming languages, or verification notations.

The paper presents a case study in which MARMOT's abstract component model has been applied according to typical reuse scenarios for the development of an exterior rear-view mirror system. The results of the case study indicate that using an abstract component model in tandem with component-based development has a positive impact on reuse, development efforts, and

quality.

In summary, the MARMOT method provides an easy-to-use development framework for embedded systems, facilitates the development of high-quality components or systems, and fosters systematic reuse by supporting high-level modeling and the instantiation of components. Although, the MARMOT method requires significant upfront investments (similar to product line engineering approaches) and training, it provides a positive return on the investment as soon as follow-up projects are conducted. The MARMOT method is not suited for systems that will not be reused in the future, though, this is unlikely in the embedded systems world.

Our greatest concern for the future is the application and evaluation of this work when it is presented in a larger-scale industrial context. Of particular interest here is the evaluation of the cost effectiveness of this method (i.e., initial investment vs. trade-off in reuse) as compared with the development approach used in an organization.

REFERENCES

[1] R. Alur and D.L. Dill. *A theory of timed automata*, Theoretical Computer Science, No.126, 1994.

[2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J.Wüst, J. Zettel. *Component-Based Product Line Engineering with UML*, Addison-Wesley, UK, 2001.

[3] K.Beck, C.Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*, Addison-Wesley Professional, 2004.

[4] L.C. Briand, J. Wüst, J.W. Daly, D.V. Porter. *Exploring the relationships between design measures and software quality*. Journal of Systems and Software, 51(3), pp.245-273. 2000.

[5] C.Bunse, CAtkinson. *The Normal Object Form: Bridging the Gap from Models to Code*, Proceedings of UML'99: The Unified Modeling Language – Beyond the Standard, Second International Conference, Fort Collins, USA, pp 691-705, CO, 28-30, 1999, Springer, Lecture Notes in Computer Science

[6] C. Bunse and H.-G. Gross. *Unifying Hardware and Software Components for Embedded System Development*, Architecting Systems with Trustworthy Components, Lecture Notes in Computer Science, Vol.3938, Springer, 2006.

[7] C.Bunse, C.Peper and H.-G.Gross. *Embedded System Construction - Evaluation of Model-Driven and Component-Based Development Approaches*, Proceedings of Workshops and Symposia at 11th Int. Conf. Model Driven Engineering Languages and Systems. LNCS 5421, Springer, 2008.

[8] J. Carlson, J. Håkansson and P. Pettersson. *SaveCCM: An Analysable Component Model for Real-Time Systems*, Electr. Notes Theoretical Computer Science, Vol.160, pp.127-140, 2006.

[9] Y. Choi and C. Bunse. *Design Verification in Model-based micro-Controller development using an Abstract Component*, Software and Systems Modeling, Vol 1, No 1, pp.91-115, 2011.

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

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

[12] R. France and B. Rumpe. *Model-Driven Development of Complex Software: A Research Roadmap*. In Proceedings of the 29th Intl Conference on Software Engineering (ICSE 2007), Future of Software Engineering, pp.37-54, Minneapolis, 2007.

[13] N. Guelfi and A. Mammar. *A formal semantics of timed activity diagrams and its PROMELA translation*. 12th Asia-Pacific Software Engineering Conference, Seoul, South Korea 2005.

[14] H.-G. Gross. *Component-Based Software Testing with UML*. Springer, Heidelberg, 2005.

[15] N. Habra, A. Abran, M. Lopez, A. Sellami. *A framework for the design and verification of software measurement methods*. Journal of Systems and Software, 81(5), pp.633-648, 2008.

[16] D. Harel, M. Politi. *D. Harel and M. Politi, Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.

[17] G. J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley, 2003.

[18] P. Hruschka and C. Rupp.*Agile SW-Entwicklung fuer Embedded Real-Time Systems mit UML*. Hanser, 2002.
[19] J. Jürjens and P. Shabalin. *Tools for Secure Systems Development with UML*. International Journal of Software Tools and Technology Transfer, 9, pp.527-544, 2007.
[20] J. Jürjens, D. Reiss, D. Trachtenherz. *Model-Based Quality Assurance of Automotive Software*, 11th International Conference on Model Driven Engineering Languages and Systems, Toulouse, France, 2008.
[21] N. Juristo, A.M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers. 2001.
[22] M.U. Khan. *Model-Driven Development of Real-Time Systems with UML 2 and C*. 3rd International Workshop on Model-Based Methodologies of Pervasive and Embedded Software, pp.33-42, 2006.
[23] P. Kruchten. *The Rational Unified Process - An Introduction*. Addison-Wesley, 2003.
[24] K.G. Larsen, P. Pettersson, W. Yi, *Uppaal in a Nutshell*, Int. Journal on Software Tools for Technology Transfer 1 (1997), pp.134-152.
[25] C.F. Lange. *Model-Size Matters*. Workshop on Model Size Metrics, co-located with ACM/IEEE MoDELS/UML Conference, October, 2006.
[26] K. Lano. *Formal Object-Oriented Development*. Springer, 1995.
[27] L. Lavagno, G. Martin and B. Selic (Eds.). *UML for Real Design of Embedded Real-Time Systems*. Kluwer, 2003.
[28] M.Marcos. *UML Modeling of Industrial Distributed Control Systems*. 6th Portuguese Conference on Automatic Control, Portugal, 2004.
[29] R.C. Martin. *Agile Software Development*. Principles, Patterns, and Practices. Prentice Hall, 2002.
[30] P.Marwedel. *Embedded System Design*. Springer, 2006.
[31] Edmund M. Clarke, Orna Grumberg, Doron Peled. *Model Checking*, MIT Press, 1999.
[32] H.D. Mills, V.R. Basili, J.D. Gannon, R.G. Hamlet. *Teaching principles of computer programming*,15th Conference on Computer Science, St. Louis, Missouri, USA, February, 16-19, 1987.
[33] B. Selic, G. Gullekson and P.T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.
[34] C. Szyperski. *Component Software - Beyond Object Oriented Programming*. Addison-Wesley, 2002.
[35] J. Ventura, F. Siebert, A. Walter, J.J. Hunt. *HIDOORS-A High Integrity Distributed Deterministic Java Environment*, 7th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002), 7-9 January, 2002, San Diego, CA, USA.
[36] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.
[37] J. Hooman. *Towards Formal Support for UML-Based Development of Embedded Systems,* 3rd PROGRESS Workshop on Embedded Systems, Technology and Foundation, STW, 2002.
[38] G. Naeser and K. Lundqvist. *Component-Based Approach to Run-Time Kernel Specification and Verification*. 17th Euromicro Conference on Real-Time Systems, 2005.

Christian Bunse

Christian Bunse studied computer science with a minor in medicine at the Universities of Bochum and Dortmund in Germany and received his PhD degree in computer science from the Technical University of Kaiserslautern, Germany. Currently he is an Associate Professor (Software Engineering) at the University of Applied Sciences Stralsund in Germany. Prior to that he was responsible for the component engineering department at the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany. His research interests are in software methodologies, modeling, component- and service orientation, resource awareness, software reuse, and empirical software engineering.

Yunja Choi

Yunja Choi is an associate professor at the Kyungpook National University, Republic of Korea. Before that, she was a research scientist at the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany. She received the B.S degree and the M.S degree in mathematics from Yonsei University in Seoul, Korea, and the Master's degree and the PhD degree in computer science at the University of Minnesota, USA. Her research interests include light-weight formal methods, software verification, and model-driven component engineering.

Hans-Gerhard Gross

Hans-Gerhard Gross received an MSc in Computer Science (1996) from the Beuth University of Applied Sciences, Berlin, Germany, and a PhD in Software Engineering (2000) from the University of Glamorgan, Wales, UK. Following his PhD, Dr. Gross joined the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany, where he was responsible for a number of public research projects, and consulting projects with German software organizations. Since 2005, Dr. Gross is employed as Assistant Professor at Delft University of Technology, The Netherlands. His research interests encompass all phases of software development, and software evolution and software testing, in particular.