Design-Time Verification of Component-Based Embedded Systems With Respect to Extra-Functional Properties

Juraj Feljan
Mälardalen Real-Time Research Centre
Mälardalen University
Sweden
juraj.feljan@mdh.se

ABSTRACT

When developing embedded systems, certain constraints regarding extra-functional properties have to be guaranteed. It is desirable to be able to perform early design-time verification of embedded systems with respect to their extra-functional properties, because the earlier potential design flaws are caught, the easier and cheaper it is to correct them. Employing component-based software engineering and model-driven development for the development of embedded systems can facilitate this early verification. In this paper we present our planned research on early analysis of component-based embedded systems, which enables avoiding designs infeasible with respect to constraints on timing and resource consumption.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Verification

Keywords

embedded systems, extra-functional properties, design-time verification, component-based software engineering, model-driven development

1. INTRODUCTION

Computers have become highly intertwined with our daily routine, as they are used in industry, business, communication, traffic, health, research, education, entertainment, etc. Computer systems are continuously growing in complexity and size, increasing the cost of developing and maintaining them. Most computers systems today are *embedded systems* — microprocessor based systems with a single dedicated function. A significant part of the functionality of embedded systems is realized in software. Parallel with the growth of software complexity, there is an increasing demand on it to be robust, reliable, flexible, adoptable, etc. Thus, a systematic approach to steering the whole software life cycle is necessary. The discipline addressing this is *software engineering*. IEEE defines software engineering as the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software [1]. Two new promising approaches gained a lot of attention recently, namely *component-based software engineering* and *model-driven development*. These can be considered as subdisciplines of software engineering. The former promotes building systems not from scratch, but from pre-developed software components, thus lowering time-to-market. The latter advocates shifting the focus of software development from code to models, which should enable the developers to focus more on the application logic than on the implementation details.

Embedded systems have particularities not present in "traditional" desktop- and Web applications, in the sense that the correctness of embedded systems does not only depend on correct logical functioning, but also on *extra-functional properties*, such as timing and resource usage. When developing such systems, certain constraints on extra-functional properties have to be guaranteed. It is desirable to be able to check, as early as possible in the development process, whether these constraints will hold, in order to avoid time and money loss caused by doing redesign after the implementation had been done. This is the focus of our research, and it can be facilitated by using component-based software engineering and model-driven development.

In this article we give an overview of our planned research, and it is organized as follows. Section 2 presents the background, while Section 3 details the research. Related work is surveyed in Section 4. The paper is concluded by Section 5.

2. BACKGROUND

In the background section we present the particularities of embedded systems, and then the basic ideas of component-based software engineering and model-driven development, respectively.

2.1 Embedded Systems

Today most computer systems are embedded systems. To illustrate, more than 98% of all processors manufactured in 2005 were used in embedded systems [14]. The term embedded system is not strictly defined, but usually refers to a microprocessor based system with a single dedicated func-

WCOP'11, June 20–24, 2011, Boulder, Colorado, USA.

tion (or few dedicated functions), as opposed to a general purpose desktop computer. An embedded system is usually a part of a larger system or product. Embedded systems span from small devices such as MP3 players to large systems such as factory controllers. Their complexity follows accordingly, from a single microcontroller chip, to multiple nodes connected via a network.

Embedded systems usually have limited resources in terms of processing power, memory size or battery power. This is because they are often limited in physical size, due to the fact that they have to blend in with the environment they operate in. Another reason is that some embedded systems are produced in large quantities, and thus have to be cheap to produce.

Moreover, embedded systems are often *safety-critical* and *real-time* systems. The former means that their failure can result in disaster, either loss of human life, or expensive damage to equipment. The latter means that their correct behavior depends not only on logical computations, but also on the time at which the results are produced.

Due to these requirements, limited resources on one hand and reliability on the other, for embedded systems extra-functional properties (EFPs) have significant importance, and thus have to be explicitly addressed during the development process. EFPs (or quality attributes) encompass timing properties (e.g. latency, worst-case execution time), resource-wise properties (e.g. memory consumption, power consumption), dependability, reliability, security etc.

In order for an embedded system to function properly, it is not enough for it to exhibit correct logical behavior. Additionally, it must satisfy certain constraints on EFPs. An example of such a constraint can be the following — an embedded system controlling the airbags in a car must have a response time of maximum 10ms, and cannot have a memory footprint larger than 10kb. Due to its safety-critical role, it is not enough if the system only deploys the airbag, it is also crucial when the deployment takes place. Such constraints have to be guaranteed during the development process. This is usually done by employing formal methods, which allows various analysis to be performed on embedded systems (for instance schedulability analysis which checks that component execution is scheduled in such a way that all components meet their timing deadlines). Formal analysis techniques are usually combined with stringent testing.

Having in mind all the listed particularities, it is clear that developing embedded systems differs from developing "traditional" desktop- and Web applications.

2.2 Component-Based Software Engineering

Component-based development is a concept well known in building hardware. For example, a radio can be built by connecting integrated circuits. Further, a personal computer can be built by picking and assembling pre-existing components such as CPU, memory, hard disk, etc. To be able to connect hardware components, their interfaces have to "match", i.e. they have to follow a certain standard.

Component-based software engineering (CBSE) is the software equivalent of the above. The central idea behind CBSE is that software systems are not built from scratch, but from pre-developed software components. Ideally a system can be built by browsing a repository of available components, selecting the right components, and finally connecting them together. This way, the development of systems is separated from the development of components. In practice some glue code (i.e. code that enables communication between originally incompatible components) will have to be written, and particular components (those that are not found in the repository) will have to be implemented in parallel with the system development.

One of the biggest benefits CBSE promotes is reusability — it should be possible to reuse a component across different systems. Thus, employing CBSE should:

- shorten time to market — systems are built by connecting existing components;
- facilitate management of complex systems — by dividing them into smaller, less complex components;
- increase the quality of the software — since a component is intended for reuse in multiple systems, it is repeatedly tested in various contexts;
- simplify maintenance — improving existing or adding new functionality to a system is done by replacing existing or adding new components.

A paramount concept in CBSE is the *component model*. A component model provides methods and rules for (i) component specification and (ii) component composition, and therefore corresponds to the aforementioned standard. To be able to easily connect software components together, they have to follow the same standard, i.e. they have to conform to the same component model.

Although a promising approach, CBSE still has many open questions and is the subject of ongoing research, especially in the embedded systems domain. Some of the main challenges CBSE is facing are [7]:

- component specification — still no consensus has been made about what a component is and how it should be specified;
- component models — the currently available component models have many ambiguous characteristics, they are incomplete and use inconsistent terminology (for a more detailed discussion on this issue see [8]);
- component-based software life cycle — how to define precise component requirements since it is not known beforehand in what systems the component is going to be used, how to find a particular component that fits a system that is being built, what if a component only partially fits;
- extra-functional properties — how to express quality attributes of a component, such as performance, required resources, reliability, latency, security, etc.;
- composition predictability — it is not clear how extra-functional properties of a particular component affect the corresponding extra-functional properties of a composition of components (this issue is particulary important for embedded systems with rigorous safety requirements).

2.3 Model-Driven Development

Model-driven development (MDD) aims at shifting the focus of software development from code to models. In other words, MDD promotes tackling software complexity by raising the abstraction level of software systems to a higher level than code, closer to concepts from the application domain than to algorithmic concepts. This enables the developer to focus on the application logic, without worrying about implementation details. The raise of abstraction level from code to models is equivalent to switching from assembly code to procedural languages, or from procedural to object-oriented languages.

In "traditional" software development, models usually aid in understanding the system being built. In MDD, models do not serve only as documentation, but become a formal specification of the system. In the MDD context models are the main artefact — the requirements, the system architecture, verification and validation, etc. are all models. The basic idea is that from the model(s) of a system, through the process of model transformation, implementation code can automatically be generated. However, models should not only be used for code generation, but also for generation of non-implementation artifacts such as documentation, tests, deployment scripts or other models. MDD should provide the following benefits:

- shorter time-to-market;
- early verification of a system design — the earlier potential design flaws are caught, the easier and cheaper it is to correct them;
- coherent system design and implementation — a change done on a model at one level of abstraction should be automatically promoted to all other levels, keeping all models coherent;
- application logic independent of technological changes — if the application logic is specified by a model that contains no information specific to the implementation platform, then implementations for different platforms can be automatically generated.

Similarly to CBSE, MDD is a relatively new approach with open questions that are the subject of ongoing research. For instance, how to ensure coherence between different models of a system, how to handle model evolution, how to capture extra-functional properties in models, how to separate application logic specifications from platform specific information, etc.

3. PROPOSED RESEARCH

In this section we first present the motivation for doing design-time verification with respect to EFPs, and formulate the general research question. Then we describe the context and the assumptions of the research. Finally, by outlining a research plan and listing the expected results, we give more details of what the planned research encompasses.

3.1 Motivation and Research Question

In order for safety-critical, real-time embedded systems to function correctly, in addition to performing correct logical computations, certain constraints on their EFPs have to be satisfied. It is extremely desirable to be able to verify systems with respect to EFPs early at design-time. The earlier in the development lifecycle design flaws are discovered, the easier and cheaper it is to correct them. Design-time verification with respect to EFPs can therefore prevent severe loss in time and money, caused by discovering design flaws after the implementation had been done.

Different implementation platforms imply different EFPs, and in our research we investigate how EFPs change with different implementations. The final goal of the research is a methodology for early design-time verification of component-based embedded systems with respect to EFPs, more specifically timing and resource consumption. Having this in mind, our research tackles the following general research question: *How can early analysis help avoiding infeasible component-based designs of embedded systems, with respect to constraints on timing and resource consumption?*

CBSE and MDD are two approaches that aid us with answering the research question. Conceptually, MDD is mainly a top-down approach, as it promotes incrementally transforming models of a system to code, while CBSE focuses on run-time composition of components' executable code, making it a bottom-up approach. However, looking only at design-time, which is in the focus of our research, CBSE and MDD overlap, and the border between the two is fuzzy. Here the two approaches share the notion of models. Currently the trend in CBSE is to develop models of components and their interactions to support various types of model-checking and analysis, and therefore facilitate the aforementioned run-time composition of components. In our research we address both CBSE and MDD, while trying to leverage their respective advantages, encapsulation facilitating reuse from the former, and early analysis from the latter.

3.2 Research Context

Design-time verification of component-based embedded systems with respect to EFPs answers the question whether a particular deployment of software components to hardware nodes of the implementation platform satisfies the constraints that were set on EFPs. In order to be able to perform this verification, we assume a model-driven approach in the design phase that includes:

- *an architectural model* specifying the system under development as a composition of software components;
- *a platform model* specifying available hardware nodes the components will run on, and the connections between the nodes;
- *a mapping model* specifying the mapping between the architectural model and the platform model, i.e. the mapping of software components onto hardware nodes.

Additionally, the components in the architectural model have to specify their EFP requirements, while the hardware nodes in the platform model must specify what they offer in terms of resources. Then we can perform the aforementioned verification, by applying a suitable analysis method. For example, if we want to verify if a particular design is correct with respect to static memory usage, each component in the architectural model must state how much static memory it requires, while each hardware node must specify a budget in terms of how much memory it provides. Using these values and the three aforementioned models as input, an analysis of static memory usage will output an answer showing whether the design is feasible. Static memory usage is an EFP where

composition is straightforward (summation). However, this is not the case with all EFPs, which is one of the challenges of the research.

The above implies that a mapping model is given. This mapping is then verified. However, to be more general than only answering if a particular mapping is correct, it is desirable to be able to automatically derive a mapping in which timing constraints are met and resources are used efficiently, according to certain trade-off constraints. For example, one mapping might imply a bigger memory footprint on one hardware node, but it will however guarantee lower worst-case execution time values, which enables a more flexible scheduling policy. This may mean that one mapping is "better" than the other, depending on what the developer values more in this case, less memory usage or flexible scheduling. When a desired mapping is found, it serves as input to the synthesis process, i.e. generating the implementation code.

Our research is performed in the context of ProCom [5, 17], a component model for real-time distributed embedded systems in the vehicular- and automation domains. ProCom aims at giving a holistic solution, by providing support for design, analysis and synthesis of embedded systems. Here we describe what support ProCom provides regarding the aforementioned. The architectural model in ProCom has two layers. The upper layer, called ProSys, models a system as a collection of active, concurrent subsystems that communicate by asynchronous message passing, and are typically distributed. The lower layer, ProSave, models the detailed structure of individual ProSys subsystems. ProSave components are passive, and represent smaller and simpler units of functionality. ProCom's platform model also has two layers, *the virtual platform* and *the physical platform* [6]. This approach allows more detailed analysis to be performed on the virtual platform without full knowledge of other parts that will share the same physical (hardware) node in the final system. The mapping model in ProCom is defined by allocating ProSys subsystems to virtual nodes, and by allocating virtual nodes to physical nodes. Specifying EFPs in ProCom is done through the support of *the attribute framework* [16].

ProCom is supported by an integrated development environment called Pride [12]. The architectural model and the attribute framework are fully implemented in Pride, while the platform- and mapping models are partially implemented. Pride has a flexible *analysis framework*, which allows plugging external analysis tools. Through the analysis framework various analysis techniques, that represent the backbone of design-time verification with respect to EFPs, will be included. The fact that ProCom is supported by an integrated development environment provides good foundation for validation of our research results, as the environment can be used to perform case-studies.

3.3 Research Plan and Expected Results

We have started the research by studying how EFPs are handled in existing component models. As the next step we will identify exactly which timing and resource-wise EFPs to focus on. Having in mind the intended domain — safety-critical, real-time embedded systems — these will mainly be worst-case values, rather than average ones. They will include worst-case execution time (WCET), which is crucial information in guaranteeing that a real-time system meets all of its deadlines; static-memory consumption and CPU utilization, which are important attributes in resource-constrained systems.

For all the selected EFPs respectively, we will identify (develop new, or find and tailor existing) a suitable analysis method which answers whether a particular system design is feasible, i.e. satisfies the constraints set on that particular EFP. We will extend the application-, platform- and mapping models in order to support the aforementioned analysis methods.

In order exploit the full potential of CBSE, we will develop a special analysis framework (not to be mistaken with the existing analysis framework from Pride, i.e., a tooling concept which enables plugging new analysis tools into Pride). Our assumed CBSE process is not a one way process going sequentially from design, through analysis to synthesis. Rather, it is a fully incremental process where information from a later stage might be propagated back to an earlier stage, in order to tailor a particular aspect of the system. Furthermore, due to reusability being one of the key concepts in CBSE, when developing a system we envision having a mix of fully implemented, analyzed and tested components with early components having only an interface. Having this in mind, one of the key challenges that our approach tackles is supporting early analysis that can leverage information coming from a later stage. For example, let us imagine a simple system with only two components, one reused and one new. The reused component comes bundled with a value for WCET coming from code analysis, while the new one has an expert estimation as its WCET value. We can perform schedulability analysis based on the two WCET values. Later in the development process the new component will get an implementation, which enables getting a more accurate WCET value based on code analysis. Now we can repeat the schedulability analysis with the new WCET value. Therefore we will have two analysis results based on the different WCET values for the new component. In a real-life application both the number of components and the number of different sources for a single EFP value will be significantly higher, raising the question how to value the different results coming from the same type of analysis when using different EFP inputs. What if we get conflicting analysis results, depending on where the inputs come from? Another challenge is determining the validity of an EFP value bundled with a pre-existing component, since some EFPs are platform-dependent. We will develop an analysis framework that will support our incremental CBSE process, i.e., that will (i) handle the validity of different EFP values in a particular context, (ii) handle the validity of (possibly conflicting) analysis results when using different EFP inputs, and (iii) bundle the analysis results and their validity context together with their respective components.

When being able to analyze a particular mapping, and leverage an iterative analysis process, we plan to develop a method for automatically finding a "good enough" mapping, based on a trade-off between different criteria. As a particularly interesting relation in resource-constrained and real-time embedded systems, we will focus on trade-off between resource consumption and timing. A possible option for performing the trade-off analysis is Pareto optimality. Given a mapping of software components to hardware nodes, a Pareto improvement is a change to a different mapping that will improve a certain EFP, without making any other EFP

worse. A mapping is Pareto optimal when no more Pareto improvements are possible.

Having the aforementioned research plan in mind, as the main contributions of this research we expect:

- for each EFP selected, a new or tailored analysis method for verifying if a particular component-based embedded system design satisfies the constraints on that EFP;
- an analysis framework leveraging an incremental CBSE process, i.e., backtracking late analysis results to early analysis; and
- a method for automatically deriving a "good enough" mapping (deployment) of software components to the hardware nodes of a system, according to certain trade-off criteria.

4. RELATED WORK

We present related work by overviewing how particular component-based- and model-driven technologies for embedded systems provide support for EFPs.

BlueArX [11] is a component model developed and used by Bosch for real-time embedded automotive applications, for example in engine control systems or chassis systems. EFPs in BlueArX are handled through *Analytic Interfaces*. An Analytic Interface is used to store a component's EFPs. EFP values are specified in XML. Since EFP values have dependencies on the hardware platform, compiler, software context etc., the context has to be specified. Analytic Interfaces in BlueArX systems are connected to *Reasoning Frameworks*, which are used for various EFP consistency checks.

DeepCompass [4] is an analysis framework for predicting performance related properties of component-based real-time systems. It combines models of individual software components and hardware blocks to produce an executable model of the system. By simulating this model, performance predictions are obtained. DeepCompas also supports trade-off between several architecture alternatives. What differentiates our planned research from DeepCompas is the fact that we focus on leveraging backtracked information from later stages of development in early analysis, and intend to support automatic derivation of "good enough" mappings according to certain trade-off criteria.

MARTE (Modeling and Analysis of Real Time and Embedded systems) [15] is a UML profile defined by OMG, that enables the use of UML for model-based development of embedded systems, covering the specification, design, and verification/validation stages. It provides facilities to annotate models with information required to perform various analyses. It focuses on performance- and schedulability analysis, but enables any kind of quantitative analysis. MARTE provides a common way to model both the hardware and software aspects of systems.

ROBOCOP [13] is a component model for high volume consumer electronics developed at the Eindhoven University of Technology. EFPs of a ROBOCOP component can be given by models that the component consists of. These EFP models can include timeliness, reliability, safety, security and resource consumption. ROBOCOP implements resource management through the *Resource management framework*. The aim of this framework is to prevent resource overloads on embedded devices that support dynamic updates or upgrades. It introduces a notion of *resource-aware consumers*, which are application entities that have information about resources needed for their operation. A special type of such entities are the *quality-aware consumers*, which consume different amounts of resources depending on the level of quality they are provided in a given moment. The consumers can register their resource needs to the framework, which can then guarantee them the requested resources or deny their request. The framework can also optimize the system quality depending on the available resources.

Rubus [10] is a component model developed by Arcticus Systems in cooperation with Mälardalen University. It is intended for development of distributed, resource-constrained, embedded control systems, with a mix of hard-, soft- and non real-time requirements. Timing properties of SWCs (software circuits, i.e Rubus components) and real-time requirements on the execution can be specified. Regarding the former, to enable timing analysis at design time, each SWC is associated with a run-time profile describing its run-time properties on different platforms. The latter are specified within the context of an assembly/composite as bounds on time from the generation of a trigger signal to the generation of another trigger signal.

SaveCCM [2] is a component model intended for embedded control applications in vehicular systems, developed at Mälardalen University. It is a simple component model that limits the flexibility of modeling to enable analyzability with respect to timing. Regarding EFPs, SaveCCM focuses on timing properties. It supports specification and analysis of timing properties. They can be analyzed at design time using the UPPAAL Port model checker [9]. There is also support for generic specification of EFPs. An EFP is represented as a triple $\langle$*Attribute*, *Value*, *Credibility*$\rangle$ where *Attribute* is the property name, *Value* holds the property data, and *Credibility* gives the confidence measure that *Value* represents the actual value.

Palladio [3] is not a component model for embedded systems, but for business information systems. It is described here since it is designed to enable early performance predictions for component-based software architectures of business information systems, and early EFP predictions are in the focus of our research. The development of the model started in 2003 at the University of Oldenburg and is since 2006 continued at the University of Karlsruhe. The key feature of Palladio is the parameterized component quality-of-service (QoS) specification. It is a special QoS specification for software components, parameterized over environmental influences that are unknown to component developers during component design and implementation. This specification is called *resource demanding service effect specifications* (RDSEFF). RDSEFFs abstractly model the externally observable behavior of a component. They specify: how a provided service calls the required services of a component, resource usage, transition probabilities, loop iteration numbers and parameter dependencies, all this to allow accurate performance predictions. RDSEFFs can be considered as a domain-specific modeling language which the component developer uses to specify performance related information for components. They represent the gray-box view of components.

5. CONCLUSION

This article has presented our planned research on handling extra-functional properties of embedded systems, or to be more specific, design-time verification of component-based embedded systems with respect to timing and resource consumption. We have described the background, motivation and the context of the research, and discussed our research plan and expected results.

The research is motivated by the fact that EFPs have significant importance in safety-critical, real-time embedded systems, and have to be explicitly addressed during the development process. In order to save money and time by avoiding redesign after the implementation had already been done, it is desirable to be able to discard designs infeasible with respect to constraints on EFPs prior to the implementation. Early verification with respect to EFPs carries several challenges. For example, composition of some extra-functional properties is not straightforward; EFPs can depend both on software and hardware; how to leverage in early analysis information backtracked from later stages of development; how to perform trade-off analysis between different EFPs, etc.

6. REFERENCES

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990.

[2] M. Åkerholm, J. Carlson, J. Håkansson, H. Hansson, M. Nolin, T. Nolte, and P. Pettersson. The SaveCCM Language Reference Manual. Technical Report, Mälardalen University, 2007.

[3] S. Becker, H. Koziolek, and R. Reussner. Model-Based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th International Workshop on Software and Performance*, 2007.

[4] E. Bondarev. *Design-time performance analysis of component-based real-time systems*. PhD thesis, Eindhoven Universty of Technology, 2009.

[5] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report, Mälardalen University, 2008.

[6] J. Carlson, J. Feljan, J. Mäki-Turja, and M. Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2010.

[7] I. Crnković and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[8] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 2010.

[9] J. Håkansson, J. Carlson, A. Monot, and P. Pettersson. Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT. In *6th International Symposium on Automated Technology for Verification and Analysis*, 2008.

[10] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. In *3rd Int. Symposium on Industrial Embedded Systems*, 2008.

[11] J. E. Kim, R. Kapoor, M. Herrmann, J. Haerdtlein, F. Grzeschniok, and P. Lutz. Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, 2008.

[12] Mälardalen University. PRIDE. `http://www.idt.mdh.se/pride/`.

[13] J. Muskens, M. V. Chaudron, and J. Lukkien. A Component Framework for Consumer Electronics Middleware. In *Lecture Notes in Computer Science*, 2005.

[14] Netrino. Embedded Systems Glossary. `http://www.netrino.com/Embedded-Systems/Glossary`.

[15] OMG. MARTE. `http://www.omgmarte.org/`.

[16] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković. Integration of Extra-Functional Properties in Component Models. In *12th International Symposium on Component Based Software Engineering (CBSE)*, 2009.

[17] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *11th International Symposium on Component Based Software Engineering*, 2008.