Component-Based Context-Dependent Hybrid Property Prediction

Anders Möller[†⋆] Ian Peake[‡] Mikael Nolin[†⋆] Johan Fredriksson[†] Heinz Schmidt[‡]

[†]MRTC, Mälardalen University, Västerås, Sweden
[‡]Monash University, Melbourne, Australia
[⋆]CC Systems, Uppsala, Sweden

E-mail: `Anders.Moller@mdh.se`

Abstract

Many embedded systems for vehicles and consumer electronics critically depend on efficient, reliable control software, and practical methods for their production. Component-based software engineering for embedded systems is currently gaining ground since variability, reusability, and maintainability are efficiently supported. However, existing tools and methods do not guarantee efficient resource usage in these systems.

We present a method that enables resource-efficient component-based control software by extending hybrid property prediction methods (i.e. combining static and dynamic techniques) to be context-dependent, *enabling less pessimistic extra-functional component property predictions and, hence, improved resource utilisation.*

1 Introduction

Increasing reliability and efficiency of software intensive dependable embedded control systems is critical [1]. Hence, industry demands practical and accurate engineering approaches to model, predict, and verify both core software functionality and extra-functional aspects of the software.

Component-Based Development (CBD) is successfully practised to achieve enhanced software reuse and maintainability in office/Internet applications. However, in order to be equally successful in the area of embedded control – component technologies have to be resource-constrained and equipped with methods to model and predict extra-functional aspects of the software (e.g. timing and memory consumption).

The key to achieve efficient resource utilisation, on a system level, is to have access to tight and accurate models of the resource needs of the components in the system. One key resource is the CPU, where the resource requirement of a component is Worst-Case Execution Time (WCET). Recent *hybrid* methods for WCET prediction [2] have been proposed which promise a practical approach to gaining tight WCET estimates for traditional, monolithic, programs. However, existing methods for WCET estimation are overly pessimistic in a CBD setting since they are *context-oblivious*, i.e. not explicitly taking into account the current usage-context of each component.

We focus on the problem of efficient resource usage with preserved analysis accuracy of embedded Product Line Architectures (PLAs) [3], like, e.g., control software in vehicles and consumer electronics. In order to facilitate PLAs – software components must be used (and reused) across different hardware platforms and products.

To maximise reuse in these systems, components need to be flexible. Hence, reusable components often include behaviours that are only used in a few configurations. These behaviours cannot easily be removed (e.g., by dead-code elimination), because they are offered by interfaces and cannot be removed by methods based on analysis of components in isolation. Hence, most existing property prediction approaches are overly pessimistic, and, thus, design-for-reuse tends to work against accurate WCET predictions (and, therefore efficient resource utilisation) in existing models and approaches.

In previous work [4] we showed how a component model, custom-made for embedded control-systems [5, 6], can be combined with novel methods for architecture-based, compositional reasoning, modelling, and prediction [7, 8, 9].

In this paper, we propose the use of *context-dependent hybrid property prediction* methods to make efficient use of system resources. We extend the existing hybrid prediction methods by considering the component *usage-context*. We use Dependable Finite State Machines (DFSMs) to facilitate compositional, architecture-based, reasoning about system properties based on context-dependent component properties and the structure of the component assembly [9].

We illustrate our approach using the SaveComp Component Model [5], and an adaptive cruise controller implementation [6]. In this paper, we limit our context-dependent predictions to component WCET in order to reach efficient processor utilisation. Nevertheless, our approach is generally applicable to other extra-functional properties, such as memory usage, assuming that properties are compositional.

2 Background

The systems considered in this paper are categorised as dependable complex distributed computational-intense embedded real-time systems running in, e.g., vehicles or customer electronics. In these business segments methods to reuse software cross different hardware platforms and product families/versions are inquired [1]. Hence, component-based software engineering is gaining more and more interest.

2.1 The SaveComp Component Model

The SaveComp Component Model (SaveCCM) [5, 6] is a component model for control software development. SaveCCM provides three main architectural elements: *components*, *switches*, and *component assemblies*. A component is not allowed to have any dependencies to other components, or other external software (e.g. the operating system), except the visible dependencies through its input- and output-ports. A switch provides means for conditional transfer of data and/or triggering between components. Component assemblies allow composite objects to be defined, and make it possible to form aggregate components from groups of components, switches, and assemblies enabling different levels of abstraction.

The interface of an architectural element is defined by a set of ports, i.e. points of interaction between the element and its environment. SaveCCM distinguish between input- and output ports, and there are two complementary aspects of ports: the data that can be transferred via the port and the triggering of component executions. The graphical syntax of SaveCCM (see Figure 1), is similar to UML 2.0 component diagrams, but with additions to distinguish between the different types of ports. Principally, the SaveCCM syntax uses ▷ to represent triggering ports (i.e. the control flow) and □ to represent data ports (see Figure 1).

Example: An Adaptive Cruise Controller
In this section we present an Adaptive Cruise Controller (ACC) prototype, implemented in SaveCCM [6] (see Figure 1).

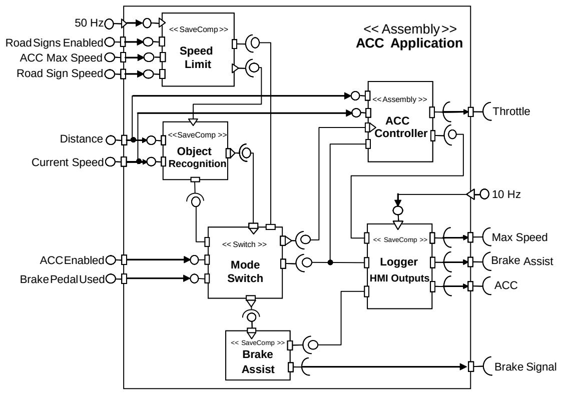

Figure 1. An Adaptive Cruise Controller described in the SaveCCM graphical modelling language

The ACC extends the regular cruise controller (used in most cars) in that it helps the driver keep a safe distance to a preceding vehicle, autonomously changes the speed depending on the speed limit regulations, and helps the driver to slam the brake in extreme situations.

The application is based on four components, one switch, and one component assembly. The assembly (Figure 2 (a)) is in turn implemented using two assemblies (Figure 2 (b)). Furthermore, the application has two different trigger frequencies, 10 Hz and 50 Hz. Logging and HMI output activities execute with the lower rate, and control related functionality at the higher rate.

For a detailed presentation of the ACC application functionality, we refer to [6].

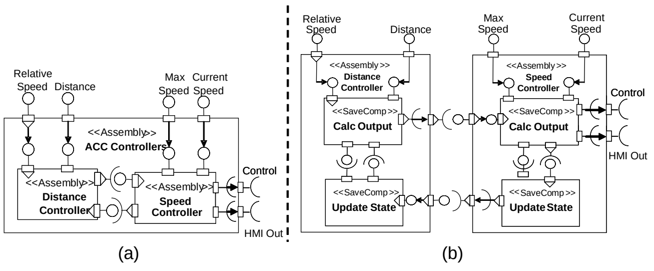

Figure 2. The ACC control assembly (a), and the implementation of the feedback controller (b)

2.2 WCET Prediction Methods

WCET bounds may be obtained via *static* (model-based), *dynamic* (measurement-based) or more recently proposed *hybrid* methods.

Static methods promise safe WCET estimates but critically depend on time-intensive construction and evaluation of models of underlying platforms [10]. Moreover, as hardware becomes increasingly complex (processors with pipelines and caches) the variance between typical and worst-case performance is growing significantly. As a result, static WCET analysis tends to produce increasing pessimism in the calculated WCET bound [11].

Dynamic methods are often cheaper to construct, but with little guarantee that acquired measurements can be used directly to derive WCET upper bounds (see Figure 3). Realising run-time measurements by trying all possible input data combinations (i.e. the complete value space) is typically not feasible.

Hybrid methods overcome some deficiencies by combining static and dynamic methods. Static analysis is used to limit the input value-space, and run-time measurements are used to calculate upper bound WCET predictions.

However, existing methods for WCET predictions are not suitable for component-based development. Existing methods take a whole-of-system approach and thus produce overly pessimistic predictions for components. Effectively these methods are only capable of producing a single portable WCET estimate for a component, whereas, in fact, the true WCET of the component may be dependent on the context in which it is later deployed.

In, e.g., [2], static (model checking) approaches are used to generate test-cases which are in turn used to generate WCET observations for functions (strictly program segments), or even basic blocks within functions. These include the use of heuristics and model checkers to generate the necessary test harnesses which exercise all possible paths leading to instrumented points within the code around "primitive" elements, then incorporating the results into conventional predictions. However, since such techniques do not make use of mechanisms to qualify the usage context of a given function f, it is not possible to reuse a predicted WCET for f in a different context. Indeed it is not clear from the content of several published papers whether it is possible to discriminate between different implicit contexts for f within the original program, or whether the same WCET for f is used regardless of context.

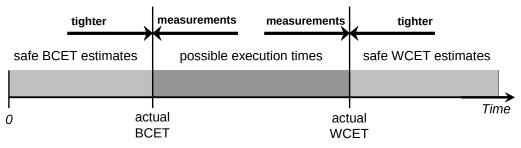

Figure 3. Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET) predictions

2.3 Context-Independent Analysis

Consider the ACC controller assembly Distance Controller from Figure 2 (a), which for the purposes of this illustration we treat as a component (see Figure 4). The Distance Controller asynchronously interoperates with its environment via four ports: three input ports and one output port.

Two input ports are for Relative Speed and Distance to the car in front, and a pair of input/output ports are for communicating with the Speed Controller. The Distance input port contains information about the distance to the vehicle in front together with a enabled/disabled boolean value. The Relative Speed input port has in the same way an integer representing the relative speed compared to the vehicle in front and a boolean enabled/disabled port. The ports related to Speed Controller communication are not considered in this example. Hence, in Figure 4 only the Distance and Relative Speed inputs are visualised.

When used in a product line of vehicles, some vehicles will be sold with the full functionality of the ACC, whereas others will be sold with a simpler, traditional cruise control function. However, this component is able to provide both functions, and will hence be deployed in both vehicle types (i.e. in two different contexts). By *disabling* both the Relative Speed and Distance ports, the Adaptive Cruise Controller (ACC) becomes a more traditional Cruise Controller (CC), by not taking into account the distance to the vehicle in front. Figure 4 visualises these two component modes by separating the internal control flow of the component.

For many components, a WCET bound, even if tight, occurs rarely, and only in certain situations. In many contexts the execution time may be much less. For example, assume that the WCET for the Distance Controller is 4ms. This might only occur rarely, in ACC mode. In the less demanding

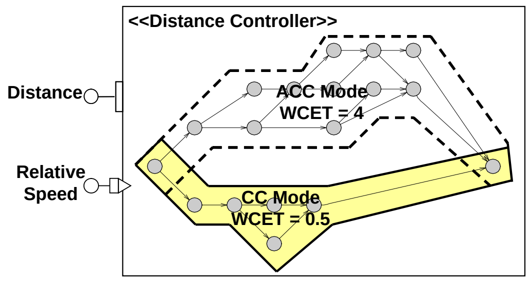

Figure 4. Context-dependent control-flow for the Distance Controller

CC-mode, the execution might never be more than say 0.5ms. Ideally, then, the WCET of a given component should always be qualified by stating the context in which it is valid (see Figure 4).

3 Dependent Finite State Machines

To model context, we make use of the formal notion of a *protocol type* from Dependent Finite State Machines (DFSMs).

DFSMs are parameterised dynamic formal models for components. They extend communicating finite state machines and model components' abstract implementation (abstraction) and deployment context as rigours parameters of a components interface specification. The abstraction parameters cater for dynamic specialisations and variations in product lines, the requires parameters capture properties and variation in different deployment contexts. Network of DFSMs represent parameterised product line architectures. Actualisation of parameters are compositional as well as the incremented assembly of components into such networks. DFSMs are particularly suitable for architecture-based reasoning about extra-functional properties.

An abstract example of what can be modelled with DFSMs is the context dependent behaviour of CompC in Figure 5. The behaviour of CompC is limited to the requsted services from CompA and CompB, i.e. the values on CompC input ports are limited to the valid output from CompA and CompB. Hence, the execution behaviour of CompC can be described as a function of the critical services required by CompA and CompB.

DFSMs have their basis in trace languages, which can be regarded as an extension of regular languages.

Regular languages promises a useful trade-off between precision and computational feasibility suitable for solving the problem identified above. DFSMs describe the allowed interactions between a given component and its environment (i.e. protocol types) as well as how the component itself is implemented. DFSMs also provide ways of talking about the structure of, and relationships between, those protocols by modelling a network of interface-protocol dependencies (see Figure 5).

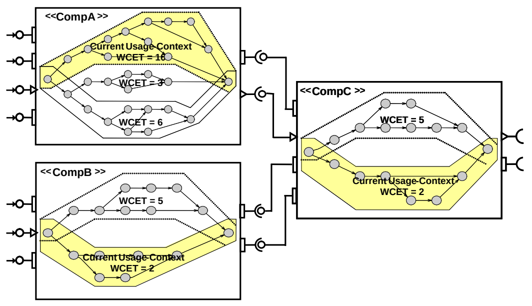

Figure 5. An abstract example of a context-dependent control-flow

3.1 Protocol Types

A *protocol type* is a formalisation of the protocol acceptable to a component, defined as a regular language.

For a concrete example of DFSMs, consider the protocol type of Distance Controller (see Section 2.3), taking into account parameters (and their respective abbreviations) Distance.Value ($Dist$), Distance.Enabled ($DistE$), RelativeSpeed.Value ($Speed$), and Speed.Enabled ($SpeedE$).

When Distance Controller is triggered, it reads values from single element buffers corresponding to its input ports. The allowed values of the relevant ports are determined by their types.

As is standard in behavioural contract specifications, valid calls to Distance Controller can be represented by a regular language. Consider representing a call to a component by a string of symbols consisting of the component's name and its actual parameters – simple binary encodings of the values present in port buffers as actual parameters.

For example, triggering the Distance with port assignments ($Dist$=01100100, $DistE$=1, $Speed$=00001010, $SpeedE$=1), would be represented by the string `invocdistance` 01100100 1 00001010 1.

A *regular language* L giving the set of all valid calls based on the above scheme can be defined using regular expressions as follows:

$L ::= \texttt{invocdistance}\ \text{Dist DistE Speed SpeedE}$

where (assuming a simple unsigned 8-bit representation for integer values):

$\text{Dist} = \text{integer (8-bit)}$
$\text{DistE} = \text{bit}$
$\text{Speed} = \text{integer (8-bit)}$
$\text{SpeedE} = \text{bit}$
$\text{integer} = \text{bitstring}$
$\text{bitstring} = \text{bit}^8$
$\text{bit} = 0|1$

Most importantly for our purposes, a protocol type can be used not only to describe the protocol acceptable to a component, but also the way a component is used in a given context.

Consider a component C whose protocol type is defined as the language L. Then it is possible to consider, for a given deployment context of the component, another protocol L' which describes the way C is used in that context. The subprotocol L' must conform to L, that is, L' must be a *sublanguage* of L (contain only strings of L).

For example, if Distance were deployed into an environment where Distance and Relative Speed inputs were permanently disabled (i.e. CC-mode), the context could be described by the language L_{CC} (a sublanguage of L):

$L_{\text{CC}} ::= \texttt{invocdistance}\ \text{integer}\ 0\ \text{integer}\ 0$

For the remainder of this paper, when we use the "context", it should be understood to include the notion of protocol subtype.

Finally, the notion of subprotocols leads to the following critical observation. For an upper-bound property such as WCET, the WCET cannot go up when context is restricted. Formally, consider a component C whose protocol type is L and two usages of C where one is strictly narrower than the other: formally L and L' where $L' \subseteq L$, with corresponding WCETs W_L and W'_L. Then the inequality $W'_L \leq W_L$ must be satisfied. This property forms the basis of context-dependent property predictions.

4 Context-Dependent Property Prediction

Accurate architectural-based reasoning about system performance is enabled by exploiting *context-dependent component property models*. Such models make it possible to formally and accurately capture the variation in properties that may occur depending on the way the components are used.

The above properties of protocol types can be exploited to derive context-dependent property predictions. Our approach conceptually separates engineer-defined static design-time configurations of the components (e.g. components differently configured to suit different product lines) from the deployment-context (i.e. component relations in the current assembly).

The usage condition, or context, of a specific component usage, instantiation or deployment, is formalised in terms of a protocol type.

Context-dependent property models are collections of *guarded* component properties: a value representing a property of some component is always qualified by pairing it with the context (i.e. a protocol type) in which it is valid.

4.1 Static Design-Time Contexts

So, for example, the fact that the WCET for Distance Controller is 4ms is expressed by the pair $(4ms, L)$, where L is as given above.

For accuracy, additional pairs may be added, refining the property model in other useful contexts. For maximum accuracy, search the pairs for the narrowest matching context with the lowest WCET. For maximum efficiency, it is possible to impose a lattice ordering over pairs based on the subprotocol relationship between guards so that not all pairs need to be considered for a given context.

For example, to express that, when neither the distance nor relative speed inputs are enabled, the WCET for the Distance Controller is much lower, e.g., 0.5ms, we simply add the pair (0.5ms, L_{CC}), where L_{CC} is as given above.

Practically speaking, implementing a subprotocol test amounts to a test for regular language inclusion, which can be implemented relatively cheaply using finite automata.

Each component in an assembly can in the same way be equipped with context-dependent information about WCET (see Figure 5) that can be reused for accurate design-time property predictions. For any component with a suitable set of guarded property pairs, in a given deployment context, a property value can be predicted at design time. A deployment context includes information about how a given component is used, particularly the deployment environment model (i.e. static design-time configurations of the components).

In this way system-level properties of the application can be derived from the context-dependent component properties. In the same way, component assemblies (i.e. hierarchical groupings of components) can be accurately predicted by propagating the usage conditions down through the assemblies.

Analyses are performed component-wise, deriving for each component a set of guarded WCETs, as needed to provide the desired level of accuracy. Where components contain no branches, a single, general, usage context may suffice. In other cases, more detailed characterisations of usage context may be required, but only up to the required level of accuracy.

4.2 Deployment Contexts

To further tighten the property predictions, one is not limited to engineer-defined static design-time configurations of the components. Additionally, to facilitate more fine-grained architectural-based reasoning, properties may be further constrained by considering the effect of connected components on the deployment context (see Figure 5).

Usage conditions can be fed into the network as constraints that propagate through the network and eliminate execution alternatives. The process can be likened to dead-code elimination, except that it is performed at the level of the property model – component code itself is not affected (see Figure 4).

For adequate accuracy and performance, this approach relies on two assumptions: (i), that correct (but not perfectly tight) upper bounds are acceptable; and, (ii), that component types with widely varying WCETs (see Figure 6) can be considered to be the union of a small set of sub-behaviours induced by non-overlapping contexts (see Figure 4), where each sub-behaviour can be accurately characterised by a single correct (and accurate) property bound.

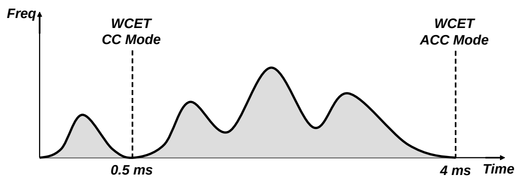

Figure 6. A conceptual context-dependent WCET graph for the Distance Controller

4.3 Context-Dependency in SaveCCM

To fully achieve the above in SaveCCM, the component protocol semantics, presented in [4] must to be extended in order to sufficiently detailed characterise the conditions leading to the WCET variation. While in general such problems are equivalent to the halting problem, existing WCET techniques already take into account context by identifying e.g., simplified domain models for variables including sub-ranges.

Modelling SaveCCM sufficiently to propagate interesting contexts from higher architectural levels is additionally complex because it includes component scheduling and asynchronous [12], buffered data flow between tasks of different frequency. Nevertheless, it should be feasible to derive such a semantics, since there are well understood extensions to automata to model relevant primitives, such as concurrency and variables.

5 Context-Dependent Hybrid Prediction

As stated in section 2, existing hybrid analysis techniques are based on a whole-of-system approach which is unsuitable for component-based analysis, making it difficult to reuse parts of an WCET analysis for individual components with sufficient accuracy.

In this section we present a hybrid property prediction method that uses DFSMs to generate context-dependent instrumentation data. We then employ the framework-based software component monitoring apporach described in [13] to measure the execution behaviour using the obtained instrumentation data.

This approach is especially beneficial in embedded product line architectures, since it facilitates early predictions of system-level properties (that can be used to guide the developer choosing inexpensive hardware) and, also, a pragmatic way to obtain extra-functional performance properties.

The resulting monitoring information, as well as the instrumentation data, is stored together with the component in the repository. Since, in a product line architecture the same application might execute on different target hardware. In this case, the engineers will simply reuse the instrumentation data and just measure the component performance on the new target hardware.

5.1 Development Process

Using the context-dependent hybrid property prediction approach together with hybrid schedulability analysis [14, 15], an attractive development process can be obtained (see Figure 7). The architectural model of the component assembly is used for architectural-based analysis, but also for test and instrumentation:

Test and Instrumentation: Design-time context-dependent analysis (as described in Section 4) is used in order to achieve early assessments about extra-functional component properties [9]. This approach yields formal context models which may be used to derive run-time measurements of the component properties as for existing hybrid approaches. Critically, however, both observed and predicted properties are reusable with confidence since they are combined with context models which state in which contexts the properties are valid. Hence, we can obtain sufficiently accurate estimations of the components extra-functional properties.

System Deployment: These properties are then used to perform tight schedulability analysis using hybrid techniques [4, 15]. Also, the system generator can be augmented to automatically insert instrumentation code to perform run-time monitoring [13] of the deployed system. And, during runtime, the unused processing time can be reclaimed using variable component performance modes as described in [14].

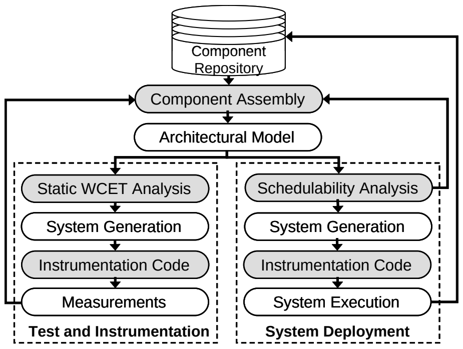

Figure 7. Component-based development using context-dependent predictions

6 Conclusions and Future Work

We present a compositional method to increase resource-efficiency in component-based control systems for product line architectures by extending hybrid property prediction methods to be context-dependent.

We introduce component usage-profiles, where extra-functional properties depend on the current usage-context, in order to obtain tighter property predictions and, hence, more efficient resource-utilisation.

As for future work, we plan to extend our theories to work with other performance properties, such as, e.g., memory or reliability [7]. We also plan to extend hybrid resource reclamation methods (such as [14, 15]) using probability distributions of the components execution time in order to predict the quality-of-service level that can be performed in the background alongside the hard real-time schedule.

Acknowledgements

We would like to thank Prof. Hans Hansson and Dr. Jukka Mäki-Turja for valuable feedback on early versions of this article.

The Monash authors acknowledge the assistance of Mr Yauheni Veryha and Dr Peter Bort for their collaboration on the Extra-Functional Consistency and Prediction (eCAP) project, seeking to apply DFSM-based techniques to a commercial WCET tool.

References

[1] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, May 2004. Edinburgh, Scotland.

[2] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Proc. Conference on Design, Automation, and Test in Europe*, March 2005.

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN 0-201-70332-7.

[4] I. Peake, A. Möller, and H. W. Schmidt. Modelling and Verification of Dependable Component-Based Vehicular Control-System Architectures. Submitted for publication, availabe as MRTC report ISSN 1404-3041 ISRN MDH-MRTC-180/2005-1-SE, Mälardalen University, May 2005.

[5] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - a Component Model for Safety-Critical Real-Time Systems. In *Proceedings of 30th Euromicro Conference, Special Session Component Models for Dependable Systems*, September 2004.

[6] M. Åkerholm, A. Möller, H. Hansson, and M. Nolin. Towards a Dependable Component Technology for Embedded System Applications. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-Time Dependable Systems (WORDS05)*, February 2005. Sedona, Arizona, USA.

[7] R. Reussner, H. Schmidt, and I. Poernomo. Reliability Prediction for Component-Based Software Architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.

[8] H. W. Schmidt, I. Peake, J. Xie, I. Thomas, B. Krämer, A. Fay, and P. Bort. Modelling Predictable Component-Based Distributed Control Architectures. In *Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, January 2004. Anacapri, Italy.

[9] H W Schmidt, B J Krämer, I Poernomo, and R Reussner. Predictable Component Architectures Using Dependent Finite State Machines. *Lecture Notes in Computer Science*, 2941:310–324, 2004.

[10] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *Software Tools for Technology Transfer*, 14, 2001.

[11] R. Kirner and P. Puschner. Discussion of Misconceptions about Worst-Case Execution-Time Analysis. In *Proceedings 3rd Euromicro International Workshop on WCET Analysis*, July 2003. Porto, Portugal.

[12] A. Wall, K. Sandström, J. Mäki-Turja, and C. Norström. Verifying Temporal Constraints on Data in Multi-Rate Transactions. In *Proceedings of the 7th International Workshop on Real-Time Computing and Applications Symposium*, December 2000. Cheju Island, South Korea.

[13] D. Sundmark, A. Möller, and M. Nolin. Monitored Software Components – A Novel Software Engineering Approach –. In *Proceedings of the 11th Asia-Pasific Software Engineering Conference, Workshop on Software Architectures and Component Technologies*, November 2004. Pusan, Korea.

[14] J. Fredriksson, M. Åkerholm, K. Sandström, and R. Dobrin. Attaining flexible real-time systems by bringing together component technologies and real-time systems theory. In *Proceedings of the 29th Euromicro Conference, CBSE Track*, September 2003. Belek, Turkey.

[15] J. Mäki-Turja, K Hänninen, and M Nolin. Efficient Development of Real-Time Systems Using Hybrid Scheduling. In *Proceedings of the International conference on Embedded Systems and Applications*, June 2005. Las Vegas, Nevada, USA.