Modelling and Verification of Dependable Component-Based Vehicular Control-System Architectures

Ian Peake[1] Anders Möller[2,3] Heinz Schmidt[1]

[1]Monash University, Melbourne, Australia
[2]MRTC, Mälardalen University, Västerås, Sweden
[3]CC Systems, Uppsala, Sweden

E-mail: `Ian.Peake@infotech.monash.edu.au`

May 30, 2005

(Research Paper)

Abstract

Domains such as vehicular control system design illustrate the need for component-based development methods based on architectural component models, and formal, design-time verification of extra-functional requirements such as schedulability. However, methods for verification and prediction of component-based systems are still impractical, requiring improved performance and accuracy.

We show how using *dependent finite state machines* (DFSMs) enables practical property prediction, through a case study illustration of an automotive cruise control system, with varying behaviour depending on configuration choices, represented in the SaveComp component model (SaveCCM). Variability is a hallmark of component-based design, in particular of product line architectures in automotive control. The parameterised component protocol types in DFSMs permit analysis of behaviour dependencies and allow refined predictions for improved accuracy in worst-case execution time (WCET) bounds for particular configurations. Since task schedulability critically depends on WCET, schedulability can be predicted more accurately. Many other approaches take a whole-of-system analysis approach, requiring computation of a detailed behavioural model of the entire system. In contrast, hierarchical DFSMs permit propagation of behaviour constraints through networks of mutually dependent state machine behaviour models. The propagation operates on hybrids of protocol state machines and simplified ”property” models – such as formulae or table representations for WCET properties of state machines.

Combining SaveCCM and schedulability analysis with DFSM semantics and dependency analysis allows scalable and accurate analysis of SaveCCM systems and extends the range of compositional extra-functional properties studied and analysed in the context of DFSMs.

Keywords: *Software quality, Evaluation of software products and components, Applications (component-based systems), Modelling*

1 Introduction

Developers of embedded vehicular control systems face challenges of (*i*) high demands on reliability and performance (*ii*) requirements on lowered product cost, and (*iii*) supporting many configurations, variants and suppliers. To meet these requirements, more and more electronics and software are introduced. In, e.g., BMW’s new 7-series luxury cars there are more then 65 electronic control units (ECUs), each of which includes its own CPU, RAM and communication devices. In the Volvo XC90, the maximum configuration contains about 40 ECUs connected via two Controller Area Networks [1], one MOST ring [2] and a set of Local Interconnect Networks [3].

However, whilst computer systems offer the performance needed for the functions requested, they also add new sources of failures that might jeopardise product reliability and safety. Also, in order to keep the software development costs within budget, more and more Original Equipment Manufacturers (OEMs) use sub-contractors (and/or

Commercial-Off-The-Shelf (COTS) components) to develop various parts of their computer system. This further increases complexity of system analysis and jeopardises software system trust, and due to the potentially high (economic and/or safety) impact of software failures (e.g., passenger safety in a car) – predictable software becomes increasingly important.

This calls for new systematic engineering approaches to design, develop, and maintain vehicular control-system software. Component-Based Software Engineering (CBSE) is such a technique, successfully used in many Internet/office applications. However, in order to be as successful in the area of dependable and safety-critical embedded vehicular control-systems, CBSE must be equipped with tools and methods to *model*, *predict*, and *verify* both core software functionality and extra-functional properties such as real-timeliness, reliability, and safety.

In this paper we combine a component model custom-made for safety-critical embedded control-systems [4] with novel methods for architecture-based reasoning, modelling, and prediction [5, 6, 7]. We show how our approach, based on dependent finite state machines (DFSMs), may be used for practical verification of extra-functional properties expressed in the SaveComp component model.

By doing this we can, e.g., guide software system developers to put focus on the part of the application that is most crucial in terms of extra-functional system properties such as real-timeliness or reliability.

We employ dependent finite state machines (DFSMs) [7] to facilitate in turn the modelling of protocols at component interfaces, practical composable models of extra-functional properties, and, e.g., verification of system schedulability under real-time constraints in distributed embedded real-time control-systems.

The distributed vehicular control-systems of interest to us require dealing with parallel behaviour (or at least communication of a sequential component with parallel components in its deployment environment). However, our work focuses initially on the sequential components and the behaviour of multiple components executing on the same embedded controller. Therefore asynchrony and scheduling issues are inside the boundaries of our project, whilst true (distributed) parallelism is outside.

The paper is outlined as follows. Section 2 introduces the novel SaveCCM, and in Section 2.1 we present an industrial prototype implementation of an adaptive cruise controller (implemented in SaveCCM). Section 3 presents the scheduling analysis performed using a novel technique combining elements of SaveCCM and extended DFSMs. Section 4 discusses related work.

2 The SaveComp Component Model

The SaveComp Component Model (SaveCCM) is for development of software for vehicular systems. The model is restrictive compared to commercial component models for enterprise systems, like, e.g., Microsoft's .NET or SUN's EJB. SaveCCM provides three main mechanisms for designing applications: (i) components which are encapsulated units of behaviour, (ii) component interconnections which may contain data, triggering for invocation of components, or a combination of both data and triggering, and (iii) switches which allow static and dynamic reconfiguration of component interconnections.

The main architectural elements in SaveCCM are components, switches, and component assemblies. The interface of an architectural element is defined by a set of ports, which are points of interaction between the element and its external 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 (for more details, see [8]).

Components are the basic units of encapsulated behaviour, and are defined by an entry function, input and output ports, and extra-functional properties. 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. A switch specifies a set of connection patterns, each defining a specific way of connecting the input and output ports of the switch. Component assemblies allow composite objects to be defined, and make it possible to form aggregate components from groups of components, switches, and assemblies.

The graphical syntax of SaveCCM, derived from UML 2.0 symbols but with additions to distinguish between the different types of ports, is available in [4]. The textual syntax is XML-based, and available in [8]. The XML-description, which contain no dependencies to the underlying system software or hardware, is used as input to the compiler. The compilation is performed in four stages, explained below.

Task-allocation. During the task-allocation step, components are assigned to operating-system tasks (independently of the execution platform). The algorithm strives to reduce the number of tasks by allocating components to the same task whenever possible, i.e. (i) when the components execute with the same period-time, or are triggered by the same event, and, (ii) when all precedence relations between interacting components are preserved. A description of the algorithm is available in [8].

Attribute Assignment. Assigning attributes is dependent on the underlying platform and on the analysis goals. In the current implementation, the task attributes are: *period-time*, *priority*, *worst-case execution-time* (WCET), and (iv) *deadline*. The period time, deadline, and WCET are derived from the components included in each task. Priority is assigned in deadline monotonic order, i.e., shorter deadline gives higher priority.

Analysis. The analysis step is dependent on the underlying platform, e.g., schedulability analysis is limited to the algorithms available in the OS used. However, in the current prototype implementation, schedulability analysis according to FPS theory is performed.

Code Generation. The code generation module of the compile-time activities generates all source code that is dependent on the underlying operating system.

2.1 An Adaptive Cruise Controller Application

To evaluate our ideas, we use a SaveCCM prototype implementation of an Adaptive Cruise Controller (ACC), first presented in [8]. The ACC extends the regular cruise controller (typically used to keep a desired speed) in that it (i) helps the driver keep a safe distance to a preceding vehicle, (ii) autonomously changes the speed depending on the speed limit regulations, and (iii) helps the driver to slam the brake in extreme situations.

When implementing the ACC application using SaveCCM (see Figure 1), we distinguish between three different sources of input to the ACC application: (i) the Human Machine Interface (HMI), (ii) the vehicular internal sensors, and, (iii) the vehicular external sensors. The outputs can be divided in two categories, the HMI outputs, and the vehicular actuators for controlling the speed of the vehicle. 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.

The application has three different operational modes: *Off*, *ACC Enabled*, and *Brake Assist*. In the *Off* mode, none of the control related functionality is activated. During the *ACC enabled* mode the control related functionality is active. In the *Brake Assist* mode, braking support for extreme situations is enabled. The ACC system (Figure 1 a) is built-up from four basic components, one switch, and one sub-assembly. The sub-assembly (ACC Controller) is in turn implemented as shown in Figure 1 b.

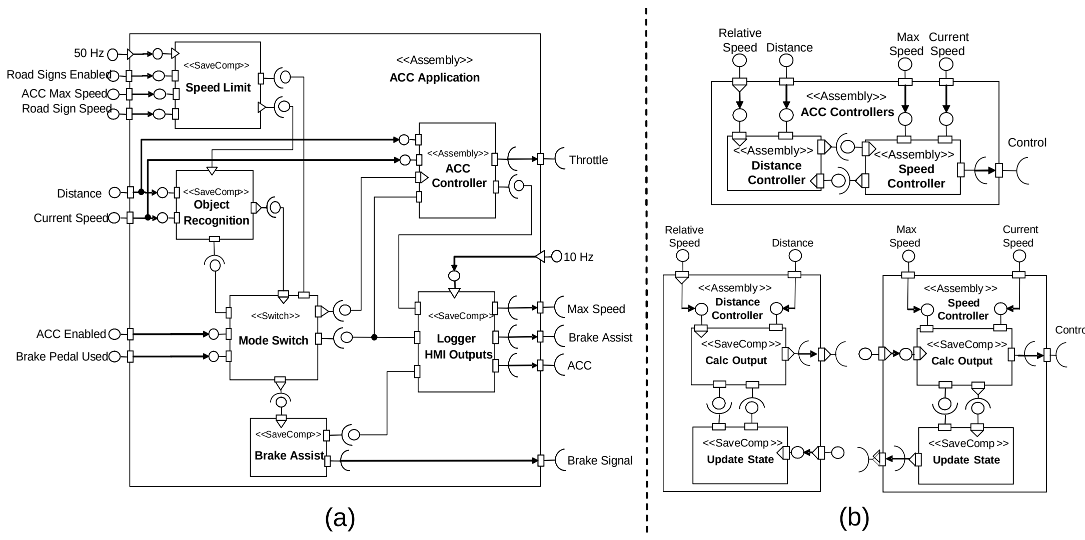

Figure 1: Adaptive Cruise Controller (ACC) application

The *Speed Limit* component calculates the maximum speed, based on input from the vehicle sensors and the maximum speed of the vehicle (speed-limit regulation dependent). The component runs with 50 Hz and triggers the

Object Recognition component. The *Object Recognition* component, in turn, is used to decide whether or not there is a car in front of the vehicle. In case there is, it calculates the relative speed to this car. The component is also used to trigger *Mode Switch* and to provide *Mode Switch* with information indicating if there is a need to use the brake assist functionality or not. *Mode Switch* is used to trigger the execution of the *ACC Controller* assembly and the *Brake Assist* component, based on the current system mode and information from *Object Recognition*. The *Brake Assist* component is used to assist the driver, by slamming the brakes if there is an dangerous obstacle in front of the vehicle. The *Logger HMI Outputs* component is used to communicate the ACC status to the driver via the HMI, and to log the internal settings of the ACC. The *ACC Controller* assembly is built up of two cascaded controllers (see Figure 1 b), managing the throttle lever of the vehicle. This assembly has two sub-level assemblies, the Distance Controller assembly and the Speed Controller assembly. A control feedback solution is used between these two controllers, see [8] for details.

Figure 2 illustrates how the components in the ACC application are allocated to operating system (OS) tasks. *Task A* is triggered at 50 Hz and, hence, executes every 20 msec. *Task A* triggers task *Task B*, *Task C*, a combination of both, or neither of these depending on the system mode. *Task D* is triggered at a frequency of 10 Hz, hence executes every 100 msec.

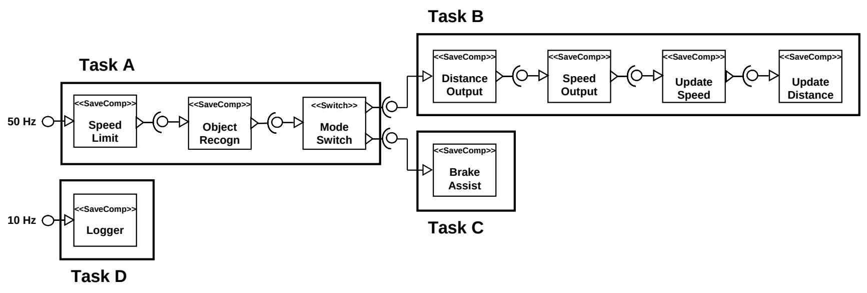

Figure 2: A control flow graph, including an illustration of the operating system tasks, of the ACC application described in SaveCCM graphical syntax

3 Modelling and Analysing Schedulability

To illustrate how the ACC application can be used in analysis of timing properties (schedulability), for the sake of brevity in this paper, we assume that the application executes on a single ECU and that the components are allocated to operating system tasks as shown in Figure 2. We further assume that the tasks are executing under a fixed priority real-time kernel, with zero execution time overhead. In Figure 3 we show how the extra-functional properties (only WCET in this specific example) of the components are summed up in order to get the schedulability attributes of *Task A* - *Task D*. The WCET's are obtained using the framework-based run-time monitoring technique described in [9].

3.1 Context-free Schedulability Analysis

If we model or analyse extra-functional properties of a component-based system without regard to the deployment context, we call this model (or the analysis, respectively) *context-free*. Context-free schedulability analysis typically follows the hierarchical composition structure or the acyclic data-flow structure between components to assign WCET and solve the schedulability problem utilisation this tree or dag structure.

For a set of independent periodic tasks, with deadlines within the period – the Deadline Monotonic priority assignment model is optimal. However, for simplicity, we assume that deadlines equal the period. Hence, we use the Rate Monotonic (RM) priority ordering, where tasks get priorities according to their periods. The task with the shortest period gets the highest priority, and the task with longest period gets the lowest. Tasks with higher priority can preempt lower priority tasks. Given this, together with the execution time attributes of the components, we can derive the following tasks for the ACC application:

Component	WCET (ms)
Speed Limit	2
Object Recogn	3
Mode Switch	1
Distance Output	3
Speed Output	3
Update Speed	1
Update Distance	1
Brake Assist	2
Logger	4

Tasks	Period (ms)	WCET (ms)	Deadline (ms)	Priority
Task A	20	6	20	4
Task B	20	8	20	3
Task C	20	2	20	2
Task D	100	4	100	1

Figure 3: Component attributes and task set information for the ACC application

By applying, e.g., response-time analysis [10], it is easy to show that this task-set is schedulable.

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j \tag{1}$$

Solving the recurrences in Equation 1 on the task-set described in Figure 3 gives us, $R_{T_A} = 6$, $R_{T_B} = 14$, $R_{T_C} = 16$, and $R_{T_D} = 20$. Since the condition $R_{T_i} < D_i$ is fulfilled for all tasks in the task-set, the ACC application is schedulable. In the upcomming versions of the SaveComp Component Technology, we will deploy more advanced schedulability tests, such as response-time analysis for tasks with offsets [11], which will allow high utilisation and can cater for real-world phenomena such as jitter and OS-overheads.

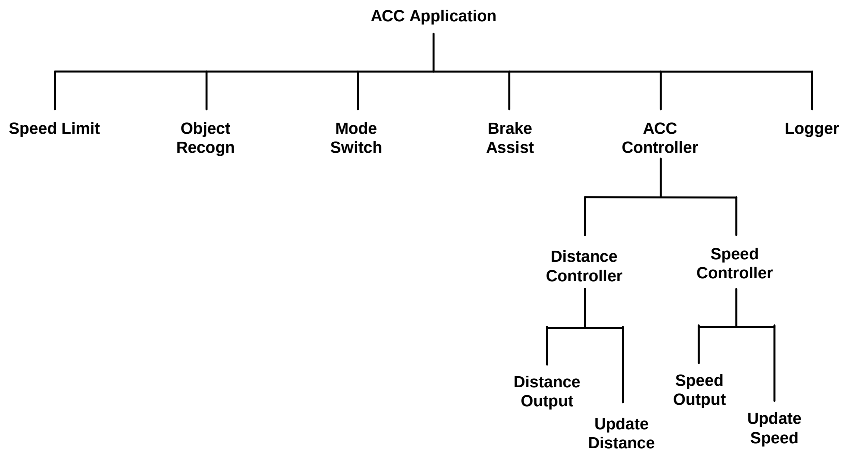

Figure 4: A component-wise hierarchical decomposition of the adaptive cruise controller

3.2 Context-dependent Schedulability

Component-based control systems architectures –and especially product-line architectures– aim at maximising reuse. The units of reuse in this approach are software components, i.e., binary deployable, independently developed, black-box entities (see e.g., [12]) and variable configuration units where variation point specifications describe dependent behaviour or component choices. Design-for-reuse maximises the flexibility and reuse of such units, however, *at the price of including behaviours in most configurations that are used only in a few of them*. Unlike dead-code elimination, these behaviours cannot be removed because they are offered by interfaces to the given deployment context of those components - an intrinsic by-product of the openness of components.

Even if these behaviours remain unexecuted in all execution contexts, or unexecuted with a very high probability, they contribute to the WCET prediction, all adding up to WCET bounds that are deviating significantly from any WCET observed in practice. Thus design-for-reuse tends to work against accurate WCET prediction in existing WCET models and approaches, because they are *context-free*, i.e., defacto do not or cannot take into account the deployment context of a set of components.

For example the ACC Controller in the Cruise Control case study includes four components: Distance Output, Speed Output, Update Speed and Update Distance, each exhibiting several different behaviours. A configuration which selects the regular ACC Control (non-adaptive behaviour) corresponds to a variation point choice that selects mutually dependent component behaviours for all four components. Let us abbreviate the relevant WCETs of the corresponding component behaviours by the component initials prefixed by 'r' for 'regular' and 'a' for 'adaptive'. Let us also assume the regular WCET of these components are $0.5ms, 0.5ms, 1ms$ and $1ms$ respectively and the adaptive WCETs are $3ms, 3ms, 0.5ms$ and $0.5ms$, respectively. Then implicitly the context-free schedulability analysis above (cf. Sec. 3.1) uses the maximum of regular and adaptive times for *all* four ACC subcomponents and sums them to arrive at the ACC component WCET.

$$WCET_{\mathrm{ACC}} = max(aDO, rDO) + max(aSO, rSO) + max(aUS, rUS) + max(aUD, rUD) = 8ms.$$

However, clearly

$$8ms = WCET_{\mathrm{ACC}} > max(aDO + aSO + aUS + aUD, rDO + rSO + rUS + rUD) = 7ms.$$

I.e. the maximum of the modelled regular and adaptive behaviors is actually smaller. Moreover in a deployment context selecting the regular behaviour only, the relevant time is considerably smaller:

$$WCET_r = rDO + rSO + rUS + rUD = 3ms.$$

In large-scale control systems such differences are considerably more pronounced. Thus, a refined analysis is necessary to arrive at accurate WCET predictions at the level of tasks to which several hierarchical components are allocated. Furthermore, such a refined analysis requires context-dependent reasoning.

Some approaches to WCET prediction propose such context-dependent analysis using data flow techniques known from compiler construction. By means of so-called scope trees, they incorporate the subcomponent models, flatten the hierarchy and then analyse the compound model of the entire product bottom-up (see for example, [13]). This white-box approach to components, obviously does not scale and also requires access to the complete source code, which is not always available. Small incremental extensions of a product by a single component may require repeated full flattening, expansion and recalculation of the WCET, which can be prohibitive in large-scale component-based software development.

3.3 Dependency networks of component protocol types

Elsewhere we have described an alternative architecture description language RADL [refs], and a semantics for RADL expressed in terms of Dependent Finite State Machines (DFSMs) allowing a compositional approach to extra-functional properties such as WCET, reliability and others - provided the properties themselves or approximate and still accurate models are reasonably compositional.

DFSMs provide a formal mechanism for describing the allowed interactions between a given component and its environment, i.e., protocols, and provide ways of talking about the structure of and relationships between protocols, by modelling a network of interface protocol dependencies. Known usage profiles and deployment environment models can be modeled as protocol types themselves, then fed into the given network as constraints which propagate through the network and (conceptually) eliminate non-executing behavioural alternatives. The process can be likened to dead-code elimination, except that it is performed at the level of the property model, and the component code itself is not affected at all.

DFSMs are defined in terms of finite state machines extended to generate trace languages, which capture notions of true concurrency. A (regular) trace language, analogously to a regular language, gives a set of traces. A trace is a set of strings, which are equivalent up to arbitrary permutation of pairs of symbols which are not ordered with respect to each other. A core notion provided by DFSMs is that of a component's *abstract machine*, that is, a model of the component that expresses how it implements its interactions with its environment. (The abstract machine is a white-box notion: it may reveal internal structure of the component, i.e., dependence on subcomponents through the use of *hidden symbols* in the language it generates.)

3.3.1 Towards DFSM semantics for SAVECOMP

Below we give a simple semantics for SaveCCM in the form of a regular trace language. We restrict ourselves, without loss of generality, to modelling features which impact on task assignment and schedulability, by modelling control, but not data, signals. The goal is to give a compositional semantics, where models of component assemblies are derivable from models of subcomponent assemblies via "simple" (i.e. local) composition.

For the sake of simplicity, we consider only tasks to be "true" components, and we assume that the assignment of components to tasks (based on inter-component triggers and data flow) has been performed already by the SaveCCM compiler. Ideally, an extension to the semantics would demonstrate how the task assignment algorithm could itself be specified and implemented compositionally.

The semantics is defined as follows:

- An entire system has a protocol which accepts a "ticking" language. The system accepts repeated "system tick" symbols in the trace language corresponding to trigger events occuring at the greatest common multiple of all trigger frequencies;
- Individual trigger frequencies have their own protocols which accept (hidden) repeated "trigger tick" symbols corresponding to their own trigger frequency;
- A system synchronisation protocol accepts both repeated system ticks and various trigger ticks, and restricts both the rate and ordering of the trigger ticks with respect to system ticks, so that trigger ticks occur at the appropriate integral fractions of the system tick rate. This includes making trigger ticks and system ticks *dependent*;
- Individual tasks (considered atomic for the purpose of this semantics) are represented by protocols which accept repeated instances of a unique corresponding task symbol, which are mutually dependent on the trigger tick corresponding to their invocation frequency. (This correctly leaves certain behaviour nondeterministic, for example the ordering of multiple tasks coincidentally scheduled for the same real time instant in the absence of other constraints);
- Where tasks invoke each other, this is represented by mutual dependence of their respective symbols, and appropriate ordering within the trace language;
- Where tasks may only optionally invoke another task, this is further represented by optionality of the invoked task in the trace language;
- Where one task has priority over another, an additional special synchronisation protocol which respects.

3.3.2 The ACC model in DFMSs

To illustrate the semantics, we give an abstract machine for the Adaptive Cruise Controller (ACC) without distinguishing adaptive versus regular behaviour. Even using a relatively simple, coarse-grained characterisation of this component, the abstract machine is surprisingly complex—indeed, it is not practical to show a complete finite state machine because of the state explosion which occurs when considering all the possible valid ordering of events occuring within its implementation.

The trace language generated by the abstract machine is as a regular trace expression as follows:

$$L = (a(b|c|bc|t_1)^*||(dt_2)^*||(t_1t)^*||(ttttt_2t)^* \tag{2}$$

$$\Sigma = \{a, b, c, t_1, t_2, t\} \tag{3}$$

$$D = \{a, b\}^2 \cup \{a, c\}^2 \cup \{b, t_1\}^2 \cup \{c, t_1\}^2 \cup \{d, t_2\}^2 \cup \{t_1, t\}^2 \cup \{t_2, t\}^2 \tag{4}$$

While the trace language semantics uses restricted shuffle products, the DFSM semantics cited avoids the associated state space explosion by using Petri nets in which independent behaviours (actions or entire components) remain unordered.

In addition, our approach keeps the state space explosion in check, which could result from flattening a large hierarchy of component models, by treating components as black boxes and associating approximate and simplified property models with them. At higher levels in the hierarchy the simpler models are propagates without recursing to the details from which they were derived. We call this compositional modeling ’property-enriched component models’.

For WCET we use formulae and tables in our implementation that describe worst case times or reliability conditional upon classes of deployment contexts. These compositional DFSM semantics are currently being evaluated in large-scale real-world case studies in collaboration with industry partners.

4 Related Work

During the last decade, tremendous advancements have been made in component-based development (CBD) for desktop- and Internet-applications, e.g., Microsoft’s COM and .NET, SUN’s EJB. However, for embedded systems no readily available technique exist [14, 15]. To this end, many projects have come up with component models that should support analysable systems, e.g. [8, 16, 17]. All these techniques are based upon static, worst-case analysis of the system. Crnkovic and Larsson provides a good overview of the problems needed to be tackled when employing component-based development for embedded systems [18], and Möller *et al.* [19] describes industrial requirements to be met by a component technology for embedded systems.

While traditional WCET approaches such as that of [**?**] are reaching maturity in the research community, our work is novel in that it seeks to address the need for a true component-based WCET analyser which, while still accurate, is also scalable to large systems for which complete source analysis may not be feasible due to sheer size or even the unavailable of source code for third party components.

In collaboration with ABB Corporate Research Center in Germany, the Monash authors are applying DFSM-based techniques to the “Extra-functional Consistency And Prediction of Component-Based Control Systems” (eCAP) project, developing a commercial prototype for the analysis of controller designs deployed using ABB’s Industrial^{IT} suite, which supports notations compliant with IEC 6 1131-3 including procedural code (“structured text”), dataflow-style “function block diagrams”, and some proprietary extensions.

5 Conclusions

In this paper we described an extension of the SaveCCM schedulability analysis to context-dependent WCET modelling. To this end we based the models on dependent finite state machines (described elsewhere). The paper contributes refined and more accurate schedulability analysis—again, for SaveCCM. It also extends DFSMs to multi-task analysis; they have so far only been used for single-task WCET analysis.

The combination of SaveCCM and schedulability analysis with the DFSM semantics and dependency analysis allows scalable and improved schedulability analysis of SaveCCM systems and extends the range of compositional extra-functional properties studied and analysed in the context of DFSMs to date.

References

[1] International Standards Organisation (ISO). Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication, November 1993. vol. ISO Standard 11898.

[2] MOST. Specification framework rev 1.1. MOST Coopertion, http://www.mostnet.de, November 1999.

[3] LIN. – Protocol, Development Tools, and Software for Local Interconnect Networks. In 9th International Conference on Electronic Systems for Vehicles, October 2000. Baden-Baden, Germany.

[4] 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.

[5] H. W. Schmidt, I. Peake, J. Xie, et al. 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.

[6] H. W. Schmidt. Trustworthy components: compositionality and prediction. *Journal of Systems and Software, Elsevier Science Inc*, 65(3):215–225, 2003.

[7] 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. (Proceedings of the 9th International Workshop in Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy; revised version of the paper of the same title in the Sep/2002 proceedings published as a TR by Universita Ca Foscari di Venezia).

[8] 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.

[9] 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.

[10] L. Sha, T. Abdelzaher, K-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[11] J.C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS)*, December 1998.

[12] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, ISBN: 0201745720, 1998.

[13] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A worst-case execution-time analysis tool prototype for embedded real-time systems. In *Workshop on Real-Time Tools (RT-TOOLS 2001) held in conjunction with CONCUR 2001*, Aalborg, Denmark, Aug 2001.

[14] I. Crnkovic. Componet-Based Approach for Embedded Systems. In *Proceedings of 9th International Workshop on Component-Oriented Programming*, June 2004. Oslo, Norway.

[15] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin. Evaluation of Component Technologies with Respect to Industrial Requirements. In *Euromicro Conference, Component-Based Software Engineering Track*, August 2004.

[16] K. C. Wallnau. Volume III: A Component Technology for Predictable Assembly from Certifiable Components. Technical report, Software Engineering Institute, Carnegie Mellon University, April 2003. Pittsburg, USA.

[17] M. Winter, T. Genssler, et al. Components for Embedded Software – The PECOS Apporach. In *The 2nd International Workshop on Composition Languages, in conjunction with the 16th ECOOP*, June 2002. Malaga, Spain.

[18] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[19] 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*. 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.