Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 80 (2007) 655–667

The Journal of Systems and Software

www.elsevier.com/locate/jss

The SAVE approach to component-based development of vehicular systems

Mikael Åkerholm [a], Jan Carlson [a,*], Johan Fredriksson [a], Hans Hansson [a], John Håkansson [b], Anders Möller [a], Paul Pettersson [b], Massimo Tivoli [c]

[a] *Mälardalen University, Department of Computer Science and Electronics, P.O. Box 883, SE-721 23, Västerås, Sweden*
[b] *Uppsala University, Department of Information Technology, P.O. Box 337, SE-751 05, Uppsala, Sweden*
[c] *University of L'Aquila, Department of Computer Science, Via Vetoio No.1, 67100 L'Aquila (AQ), Italy*

Available online 26 September 2006

Abstract

The component-based strategy aims at managing complexity, shortening time-to-market, and reducing maintenance requirements by building systems with existing components. The full potential of this strategy has not yet been demonstrated for embedded software, mainly because of specific requirements in the domain, e.g., those related to timing, dependability, and resource consumption.

We present SaveCCT – a component technology intended for vehicular systems, show the applicability of SaveCCT in the engineering process, and demonstrate its suitability for vehicular systems in an industrial case-study. Our experiments indicate that SaveCCT provides appropriate expressiveness, resource efficiency, analysis and verification support for component-based development of vehicular software.

Keywords: Component based software engineering; Component technology; Embedded systems; Vehicular systems

1. Introduction

Component-Based Software Engineering (CBSE) is a relatively new software engineering approach which has already been successful in many software development projects. It has, however, been used mainly in the domains of desktop and e-business applications, less frequently for embedded applications.

In this article we address the problem of defining a component technology suitable for the development of embedded vehicular control-system software. The underlying assumption is that an important reason for the limited success of CBSE in the embedded systems domain is the inability of commercially available component technologies to provide solutions that meet typical embedded application requirements, such as resource-efficiency, predictability, and safety. We believe that these requirements should be considered early in the software development process and then at all stages in the process, since they are demands that cannot be satisfied by consideration at only one phase in the software life-cycle. To satisfy domain requirements, the proposed component technology enables the easy usage of analysis and verification methods during the entire software development process, through automated connectivity to test and analysis tools. Constructing formal models for analysis tools manually can be a time-consuming and demanding task, which often cannot be afforded as a repeated activity. Our work has been guided by the continuous evaluation of its suitability in an industrial case-study.

The research presented in this article has been carried out within the SAVE project,[1] which has as its long-term goal the establishment of an engineering discipline for the

[*] Corresponding author. Tel.: +46 21 151722; fax: +46 21 101460.
E-mail address: jan.carlson@mdh.se (J. Carlson).

[1] http://www.mrtc.mdh.se/save/.

systematic development of component-based software for safety-critical embedded systems. The focus of SAVE is on a single application area (vehicular systems), but its results are expected to be applicable in a wider area. The component technology presented here is one of the core parts of the project.

The reuse of components has not been as generally successful in the software engineering field, as it has been in others, e.g., mechanical engineers have reused well defined components, such as nuts and bolts, for many decades. Historically, attempts to reuse software have resulted in problems due to architectural mismatches between components (Garlan et al., 1995). CBSE tries to overcome these and other obstacles hindering reuse, through processes, technologies, and tools supporting and enhancing a component-based design strategy for software (Crnkovic and Larsson, 2002). One of the central concepts in CBSE theory is the *component technology*. A component technology provides support for the composition of component-based software. It often contains various development tools for simplifying the engineering process, and provides necessary run-time support for the components. A component technology can be seen as a realisation of a *component model*. The component model specifies the common rules that all developers must follow, e.g., basic requirements for the classification of elements as components, and certain patterns for assembling components. Component technologies for embedded systems should support general embedded domain characteristics, e.g., as described by Wolf (2002): applications should use resources efficiently; it should be possible to model different aspects of the applications; the technology should support analysis early in the design process; and provide possibilities for the verification of functional and extra-functional specifications.

This article is organised as follows. The remainder of this section gives an introduction to vehicular systems, and surveys related work. Section 2 describes the different parts of our component technology SaveCCT. In Section 3 we describe the underlying component model SaveCCM. Section 4 describes the analysis techniques currently integrated with SaveCCT. Section 5 presents a case-study in which SaveCCT has been used. Finally, Section 6 concludes the article.

1.1. Vehicular systems

Our work is focused on embedded control software for vehicle systems, e.g., passenger cars, trucks, and heavy vehicles. We focus on power train and chassis systems, which we refer to as vehicular systems. These systems are highly critical for the vehicles functionality, controlling, for example, engine, brakes, and steering. Other classes of electronic systems in modern vehicles include cabin systems, and infotainment systems (Sangiovanni-Vincentelli, 2000).

The physical architecture of the electronic system in vehicles is a complex distributed computer system. The computer nodes are designated Electronic Control Units (ECUs), and are often developed by different vendors and use different hardware. As an example, Fig. 1 (from Fröberg, 2004) shows the approximate location of the 40 ECUs in a Volvo XC90. The location is primarily determined by the location of the controlled object in order to minimize the length of wiring to sensors and actuators.

Vehicular system manufacturers are interested in the CBSE approach because it facilitates the reuse of software components. In addition to the obvious advantage of the reuse it enables, CBSE also increases the maintainability of the software. Component based software is by definition modularised and any changes required can be isolated to a limited set of components. The maintenance requirements of CBS systems are thereby less than those for systems based on monolithic software. Another important benefit of using CBS in preference to other approaches is its suitability for use in product-line architectures, common to essentially all high volume products (e.g., vehicles). Product-line architectures are used to rapidly and cost-effectively provide new products in a product family. The software is organised in a base-line variant, and new products are obtained by additions to and/or replacements of the components in the base-line. However, vehicular software has certain demands that must be considered when choosing development techniques and technologies, including:

- *Analysis* – Developers of vehicular software must, during early stages in the development, perform analyses of extra-functional properties, e.g., memory consumption and processor utilisation. Furthermore, the software is critical for the vehicle behaviour, which means that predictability is very important to allow analysis of, e.g., safety invariants, real-time attributes, and reliability attributes.
- *Verification* – Developers must verify that applications meet their functional and extra-functional specification. To date, the main method is by extensive testing,

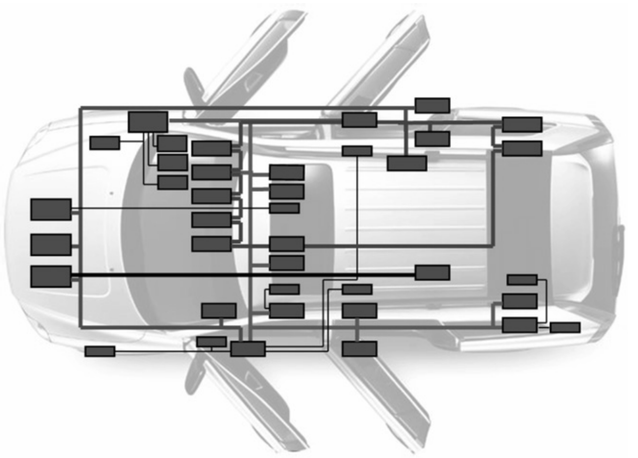

Fig. 1. Overview of the electronic system architecture in Volvo XC90.

complemented with formal methods. A component technology can improve testability by, e.g., well-defined and understandable run-time mechanisms, support for simulation to increase observability, and possibilities for mixed hardware-software tests.

- *Resource efficiency* – Because vehicles are produced in large volumes, the manufacturing cost is an important aspect. Consequently, the software platform, development technologies, and system architecture must be chosen to serve the particular needs of resources efficient systems.

1.2. Related work

In this section we relate our work to recent CBSE research from academia and industry.

Our strongest influence is the Rubus Component Technology (Lundbäck et al., 2004), which originated in our previous work with Basement (Hansson et al., 1997). The Rubus Component Technology is commercially available and is successfully used in the vehicle industry. The applications are statically scheduled, and components can be associated with timing properties such as release time and worst-case execution time. The limitations to Rubus are that the static scheduling approach only supports periodic activation and that timing aspects are the only extra-functional properties considered.

From Koala (van Ommering et al., 2000), SaveCCT has adopted the idea of using switches as the main method to achieve run-time flexibility, run-time mode changes, and design-time configuration. Koala is a component technology for consumer electronics originally designed by Philips, and then further developed in the projects Robocop[2] and Space4U[3] with Philips and Eindhoven Technical University as the main actors. These projects focus on areas such as analysis, fault prevention, power management, and terminal management; but compared with SaveCCT they are primarily intended for less safety-critical applications, such as consumer electronics.

An ongoing project with similar goals is in progress at the Software Engineering Institute. This project, designated Predictable Assembly from Certifiable Components (PACC),[4] has taken a different approach from that used in the SaveCCT project. The project focuses on how a component technology can be used and adapted to achieve predictable assemblies. Their concept of Prediction Enabled Component Technologies (PECT) (Wallnau, 2003) involves the integration of component technologies and analysis techniques. Rather than being a concrete technology (as SaveCCT), PECT is the means of restricting the usage of a given component technology in such a way that it is possible to reason about desired user-specified run-time properties, with respect to available analysis techniques.

PECOS (Nierstrass et al., 2002) is one of the component technologies targeting the automation industry. It emerged from a joint ABB and academic project focusing on developing a component technology specifically for field-devices, i.e., small reactive embedded systems. PECOS is similar to SaveCCT in the sense that it considers extra-functional properties very thoroughly in order to enable analysis, although focusing on other properties and using different techniques.

The IEC61131-3 standard (IEC, 1992) defines a graphical language that can be used for the composition of components. The language uses the same pipes-and-filters interaction model between components as SaveCCT, but the analysis of extra-functional properties is not given priority in the standard, e.g., the semantics of the different elements is not formally defined. However, the extra-functional consistency and prediction for component-based control systems project (Schmidt, 2003), develops and implements a model for prediction and consistency-checking of extra-functional properties relevant to distributed real-time control-systems. The main focus of the project is on enabling prediction in conjunction with the IEC61131-3 standard, which seems to be a promising CBSE approach for embedded systems since the standard is mature and well known. The project is being executed in parallel with our project, but no results in a real context have been made public to date.

There are also component technologies targeting distributed embedded systems using an implementation of the Real-Time Common Object Request Broker Architecture (RT CORBA) as execution platform, e.g., The Ace ORB (TAO) (Schmidt et al., 1998). RT CORBA defines a middleware architecture dealing with transparency of application-location in distributed real-time systems. The CORBA Component Model (CCM) (OMG, 2002) defines features and services related to components. As a representative for these types of technologies we choose to relate to the Boeing Bold Stroke technology implementing a component model with avionics domain-specific deviations from the CCM standard called PRISM (Roll, 2003). PRISM components interact through a client-server pattern, which is different from the pipes-and-filters model chosen to support control related functionality in SaveCCT. Despite this major difference, which might be derived from the intended usage of the components, the same type of extra-functional properties seems to be in focus, e.g., static configuration when possible, model checking, and timing analysis.

2. The SaveComp Component Technology

The SaveComp Component Technology (SaveCCT) is described here by discussing separately in the following sub-sections: *manual design*, *automated activities*, and *execution*. In Fig. 2, which provides an overview of SaveCCT, the entry point for a developer is the design tool,

[2] http://www.extra.research.philips.com/euprojects/robocop/.
[3] http://www.extra.research.philips.com/euprojects/space4u/.
[4] http://www.sei.cmu.edu/pacc/.

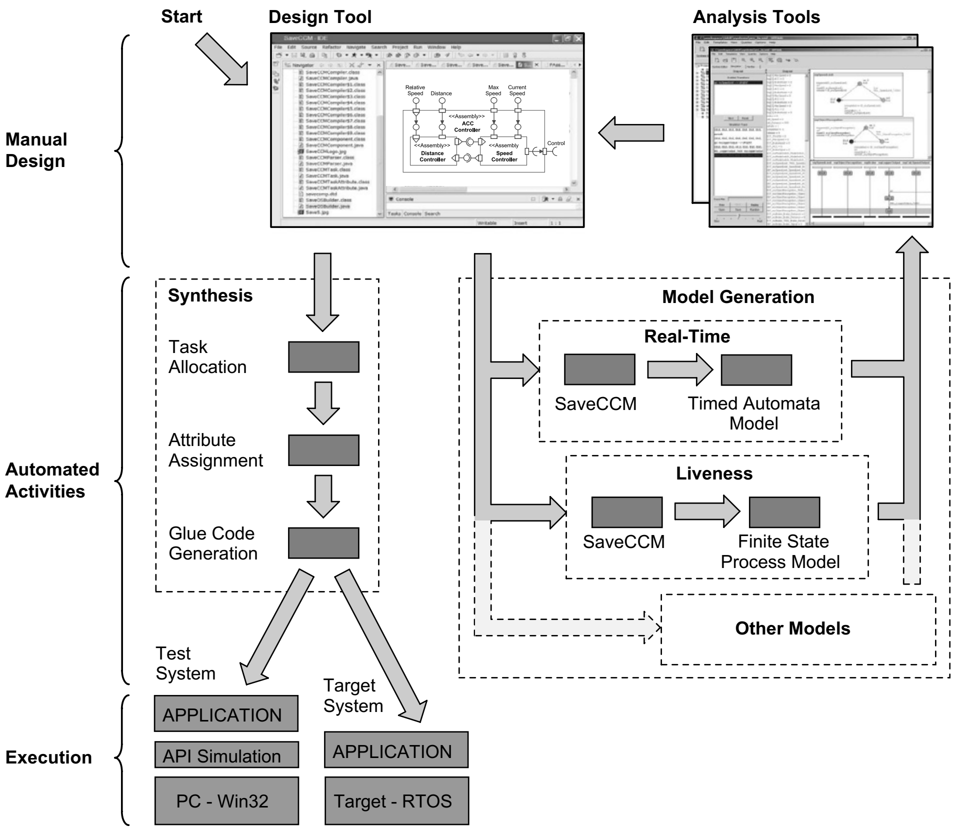

Fig. 2. Overview of the SaveComp Component Technology.

with which the application is created. During a development, a developer can utilise a number of available analysis tools with automated connections to the design tool. Analysis should be complemented by testing, which is possible at early stages in the project by replacing hardware, run-time platforms, and missing parts of the system with simulated equivalents. To simulate a system, the developer performs the same automated synthesis steps as when generating code for the real target system, only the last compilation steps being different.

2.1. Manual design

When designing, developers use a component-based strategy, supported by a set of tools for design and analysis. When CBSE is used, developers distinguish *component development* from *system development*. Component development is the process of creating components that can be used and reused in many applications. System development with components is the assembly of components to form an aggregate for a particular application. Component development and system development are independent activities that can, e.g., be parallel or performed by different companies.

The development begins with the identification of component requirements. These can be components immediately available; the remainder must be developed in parallel or purchased from third parties. Then, the SaveCCT design tool provides support for the graphical assembly of applications from existing components (i.e., system development). The tool allows designers to specify the component interconnection logic, and express high level constraints on the resulting application. Components are assembled in accordance with the rules of the SaveComp Component Model (SaveCCM) (see Section 3), which are enforced by the design tool. The component model defines different component types that are supported by SaveCCT, possible interaction schemes between components, and clarifies how different resources are bound to components. The component model has been designed so that common functionality in vehicular systems can be expressed. Specific examples of key functionality include feedback control, system mode changes, and static configuration for variability within product-line architectures.

As shown in Fig. 2, SaveCCT incorporates a number of analysis tools, which can be used for verifying specific attributes of the application, e.g., those related to timeliness and safety. To incorporate an analysis tool efficiently, as much as possible of the translation from the model created with the design tool to the model required by the analysis tool should be automated. To date we have incorporated LTSA (Magee and Kramer, 1999), and Times (Amnell et al., 2003), described in Section 4.

2.2. Automated activities

Automated activities produce necessary code for the run-time system (i.e., glue-code), and different specialised models of the application for analysis tools, e.g., finite state processes, and timed-automata models.

The synthesis activity generates all low level code (i.e., hardware and operating system interaction), so that components are free from dependency on the underlying platform. The code generation step statically resolves resource usage and timing, thereby resolving as much as possible during compile-time instead of depending on costly run-time algorithms. Synthesis consists of three steps (task allocation, attribute assignment, and code generation), described in more detail by Åkerholm et al. (2005).

The model generation activity is an automated activity which can be run separately from synthesis. Model generation is a translation from the model created by the design tool to the models (or other form of input) required by the desired analysis tools. The model created by the design tool can be adjusted to include attributes that are required to accomplish the transition, i.e., the component model is extensible in the sense that optional quality attributes of design elements can be specified. However, it might be the case that the input required by a desired analysis tool cannot be created only from information in the model created by the design tool, e.g., safety analysis often requires a model of the environment which is not addressed by the design tool.

2.3. Execution

To achieve efficient and predictable run-time behaviour, and reliable support for pre-runtime analysis, SaveCCT assumes a real-time operating system (RTOS) as the underlying platform. The current implementation uses RTXC from Quadros,[5] which is a standard fixed-priority pre-emptive multitasking RTOS. The supported target hardware in the current version is CrossFire MX from CC Systems[6], which is an electronic control unit intended for control systems running in demanding environments. Tasking[7] is integrated as the target compiler for the CrossFire MX.

To facilitate testing and debugging we incorporate CCSimTech (Möller et al., 2005a), a simulation framework which offers simulated software equivalents as replacements for much common hardware in embedded systems, e.g., IO (digital and analog), network technologies and memories. This enables testing and debugging of distributed embedded control systems in a PC environment without access to target hardware. It enables easy unit-testing by developers in their standard PCs and provides test automation possibilities. Testing can begin even before the intended target hardware is available. CCSimTech also provides support for mixed hardware-software tests in which some of the nodes in the distributed system are simulated and others are real target nodes. Most parts of an embedded application can be more efficiently tested in a PC environment, since the observability is higher than in the target system and efficient development tools for PC platforms can be utilised. However, certain verification, e.g., timing related and acceptance tests, must be performed on the target hardware, in the intended environment.

3. The SaveComp Component Model

The SaveComp Component Model (SaveCCM) formalises the SaveCCT component concept, and defines how components can be combined to create systems (Hansson et al., 2004). For use in the vehicular systems domain, the component model should support the development of resource-efficient systems and thus the run-time framework governing, e.g., component communication, must be lightweight. Another requirement is that system behaviour should be predictable, both functionally and with respect to timeliness and resource usage.

SaveCCM is based on a textual XML syntax, and a somewhat modified subset of UML2 component diagrams is used as a graphical notation. The semantics is formally defined by a two-step transformation, first from the full language to a similar but simpler language called SaveCCM Core, and then into timed automata with tasks. In this article, we use the graphical notation only, and present the semantics informally. The reader is referred to Carlson et al. (2005) for details of the formal semantics. The graphical notation is presented in Fig. 3.

In SaveCCM, systems are built from interconnected elements with well-defined interfaces consisting of input- and output ports. The three element categories; components, switches and assemblies, are described in more detail below. The model is based on the control flow (pipes-and-filters) paradigm, and an important feature is the distinction between data transfer and control flow. The former is captured by connections between *data ports* where data of a given type can be written and read, and the latter by *trigger ports* that control the activation of components. A port can also have both triggering and data functionality.

This separation of data and control flow results in a flexible model that supports both periodic and event-driven activities, since on a system level, execution can be initiated by either clocks or external events. It also allows components to exchange data without handing over the control, which simplifies the construction of, e.g., feedback loops and communication between sub-systems running at different frequencies.

Another aspect of explicit control flow is that the resulting design is sufficiently analysable with respect to temporal behaviour to allow analysis of schedulability, response time, etc., factors which are crucial to the correctness of real-time systems.

[5] http://www.quadros.com.
[6] http://www.cc-systems.com.
[7] http://www.tasking.com.

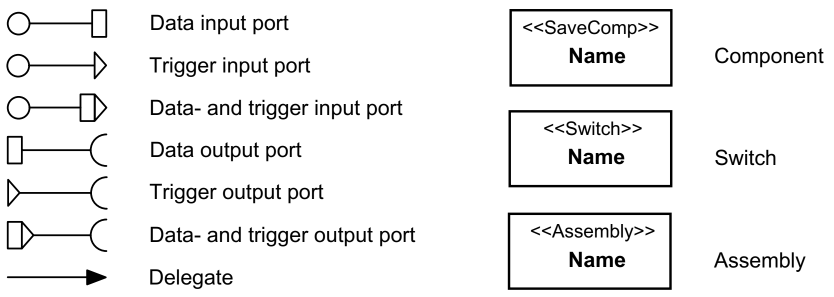

Fig. 3. The graphical notation of SaveCCM.

3.1. Components

Components are the main architectural element in SaveCCM. In addition to input and output ports, the interface of a component contains a series of quality attributes, each associated with a value and possibly a confidence measure. These attributes could include, for example, (worst case) execution time information for a number of target hardware configurations, reliability estimates, safety models, etc. The quality attributes are used for analysis, model extraction and for synthesis.

The concrete functionality of a component is typically provided by a single entry function implemented in C, but the model also allows the use of more complex components that consist of a number of possibly communicating tasks. In both cases, no intercomponent dependencies are permitted, except those explicitly captured by the ports.

A component is initially inactive. It remains in this state until all input trigger ports have been activated, at which point it switches to the executing state. In a first phase of its execution, a component reads all its input data ports. It then performs the associated computations on the basis of this input only and possibly an internal state. When the computation phase is over, i.e., when the function has been executed or, in the case of a more complex component, when all tasks have finished, the output is written to the output data ports. Finally, the input trigger ports are reset and all outgoing trigger ports are activated, after which the component returns to the idle state.

This strict "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity. In particular, a component produces the same output with preemptive and non-preemptive scheduling, i.e., whether or not a task may be interrupted by another task during its execution. The "read-execute-write" semantics also facilitates analysis, since component execution can be abstracted by a single transfer function from input values and internal state to output values.

3.2. Switches

The switch construct in SaveCCM is similar to that in Koala (van Ommering et al., 2000). Switches provide the means to change the component interconnection structure, either statically for pre-runtime static configuration, or dynamically, e.g., to implement modes and mode switches. The switch specifies a number of connection patterns, i.e., partial mappings from input to output ports. Each connection pattern is guarded by a logical expression over the data available at the input ports of the switch, defining the condition under which that pattern is active.

If fixed values are supplied to ports used in connection pattern guards, partial evaluation can determine that parts of a switch will remain unchanged during runtime. Such static parts are optimised into ordinary connections, and components that are rendered unreachable as a consequence are omitted in the final system.

It should be noted that switches are not triggered, as are components. Instead, they respond directly to the arrival of data or a trigger signal at an input port and immediately relay it according to the currently active connection patterns. Switches perform no computation other than the evaluation of connection pattern guards.

3.3. Assemblies

Assemblies are encapsulated sub-systems. The internal components and interconnections are hidden from the rest of the system, and can be accessed only indirectly through the ports of the assembly. Like switches, assemblies are not triggered. Data and trigger signals arriving at a port are immediately relayed to the outgoing connections.

Due to the restricted execution semantics of SaveCCM, an assembly generally does not satisfy the requirements of a component. Hence, an assembly should be viewed as a means for naming a collection of components and hiding its internal structure, rather than as a component composition mechanism. The SaveCCM semantics (Carlson et al., 2005) also defines an encapsulation construct that does exhibit component behaviour, enforced by additional data buffers and a mechanism to monitor the internal components to determine when to make output available at the output ports and forward the triggering. This construct does not occur in the examples in this article.

3.4. Ports and connections

As mentioned above, we distinguish between input and output ports, and between trigger ports and typed data

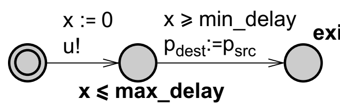

Fig. 4. The behaviour of a complex connection with a non-deterministic delay in the interval [*min_delay*, *max_delay*].

ports. Component input ports, the output ports of the whole system, and switch input ports that occur in some connection pattern guard, are one-place buffers with overwrite semantics. The remaining ports, i.e. component output ports, assembly ports and switch ports that do not occur in any guard, are just conceptual interaction points through which data passes immediately.

There are two types of connections: immediate and complex. *Immediate connections* represent loss-less, atomic migration of data or trigger signals from one port to another, as would typically be the case between components located on the same physical node. For distributed systems, and in particular during early design stages before the deployment of components to nodes has been determined, a more flexible connection concept is convenient. This is provided by *complex connections* that represent data and control transfer over channels with possible delay or information loss. The detailed characteristics of a particular complex connection are explicitly modelled by a timed automaton to capture, e.g., delay constraints, buffer sizes, or the possibility of faults.

As an example, the automaton in Fig. 4 defines the behaviour of a complex connection with a delay of at least *min_delay* and at most *max_delay* time units. When data or a trigger signal enters the connection, the automaton starts in the initial (leftmost) state. The urgent marker *u*! ensures that the first transition is made immediately, to reset the clock *x*. The invariant on the second state and the guard on the outgoing transition ensure that the desired delay is achieved before the data or trigger signal is forwarded to the destination by the assignment statement.

As in UML2, a connection from an assembly input port to an input port of an internal element, or from an internal output port to an assembly output port, is denoted by a delegation arrow, but semantically they are the same as ordinary connections from output to input ports.

4. Analysis

Much of the functionality in vehicular systems is safety-critical, since erroneous or untimely results could potentially result in death or serious injury. This stresses the need for good techniques to verify critical runtime properties of a designed system against functional and extra-functional specifications. Examples of such important properties include the absence of deadlock situations, temporal requirements imposed by the system environment (e.g., response time and jitter constraints), and dependability attributes regarding reliability, availability, and safety (e.g., vulnerability to transient network failures). Ideally, analysis should be highly automated and integrated with the design tool, since manual translation of a design into formats suiting external analysis tools is error-prone, time-consuming, and must typically be revised every time the design changes.

During component development, analysis can be used to derive component quality attributes such as execution time, resource usage, reliability measures, fault tolerance, etc. When components are combined into applications, some of these component attributes are needed as input to the analysis on system level. For example, schedulability and response time analysis require knowledge of component execution times and resource usage as well as information about how the components interact in a particular system.

The current SaveCCT environment incorporates two analysis techniques, each presented in more detail below. Since the formal semantics of SaveCCM is defined in terms of timed automata, general tools for model-checking timed automata (in our case, Times) are relatively straightforward to integrate. The other incorporated analysis technique (LTSA) is more specialised, focusing on control-loop properties.

In addition to these, a number of analysis techniques have been investigated within the SAVE project. Möller et al. (2005b) suggest the use of context-dependent property prediction to establish worst case execution time (WCET) estimates for individual components. This technique can give several execution time bounds for a component, each associated with a certain usage context, which, in some cases, would permit tighter analysis, e.g., for components that behave differently in different operational modes. Elmqvist et al. (2005) define a safety analysis framework in which components are associated with *safety interfaces*, which formally describes how faulty input (such as omission of data) can propagate to the output. Reliability, i.e., the probability of successfully performing a function for a specified period of time, has been investigated in the SaveCCT context by Dimov et al. (2005).

4.1. LTSA

LTSA (Labelled Transition System Analyser) is a verification tool for concurrent systems (Magee and Kramer, 1999). The tool is based on a process algebra notation (FSP) in which the system model and its intended behaviour is specified. The analyses supported by LTSA are *reachability analysis*, which performs an exhaustive search of the state space to verify invariants that a system must satisfy at all times, and *progress analysis* which ensures that a specified action will always be performed, as required, at some point in the future, regardless of the system state. In addition, LTSA supports simulation to facilitate interactive exploration of the system behaviour.

In connection with SaveCCT, LTSA has been used to verify certain aspects of component interaction within a system. The tool was originally incorporated in SaveCCT for analysis of control loops (see Tivoli et al., 2005), but can also be used to analyse general systems. The analysis is based on an FSP model of the system, which defines the possible orders in which actions can be performed on the different ports. Because of the architectural constraints imposed by SaveCCM, the FSP model can be easily derived automatically. For the same reason, it is possible to analyse properties incrementally, thereby avoiding state-space explosions that could otherwise occur in large compositions.

4.2. The Times tool

The modelling language *timed automata* (Alur and Dill, 1994) is useful for modelling and analysis of real-time systems. A timed automaton is essentially a finite state automaton to which real-valued clocks that can be tested and reset are added. The formalism has shown to be suitable for a wide range of real-time systems. Model-checking tools such as Uppaal (Larsen et al., 1997) and Kronos (Yovine, 1997) have been used to analyse many industrial scale systems (Bengtsson et al., 1996; David and Yi, 2000; Havelund et al., 1997; Lindahl et al., 2001).

More recently, the timed automata model has been extended with an explicit notion of tasks with parameters such as priorities, computation times, deadlines, etc. The model, designated *timed automata with tasks* (Fersman et al., 2002), associates asynchronous tasks with the locations of a timed automaton, and assumes that the tasks are executed using static or dynamic priorities by a preemptive or non-preemptive scheduling policy. The model is supported by the Times tool (Amnell et al., 2003), a tool supporting real-time analysis. In particular, the tool can check if a model is schedulable in the sense that all tasks triggered by the timed automaton are guaranteed to meet their deadlines using a given scheduling policy.

In earlier work (Carlson et al., 2005), we have described the semantics of SaveCCM formally using timed automata with tasks. A set of *core components* is identified and their formal semantics is given. It is shown how components, switches, assemblies, ports, and connections of SaveCCM can be modelled using core components.

The Save2Times tool implements the formal semantics of SaveCCM as a transformation to the model of timed automata with tasks. The tool takes as input a SaveCCM model described by an XML-file and outputs a system of timed automata with tasks that can be analysed by the Times tool. A set of properties that should normally be satisfied by any SaveCCM model is also generated in the input format of Times. We will discuss this further in Section 5.3 in which we show how the transformation tool is applied to a concrete example system.

5. Case-study: an adaptive cruise controller

The Adaptive Cruise Controller has been a recurring example throughout the development of SaveCCT. The purpose of this running case-study has been to continuously evaluate and improve the component model. Earlier experiments in collaboration with industry (Åkerholm et al., 2005) identified analysis and tool support as primary targets for improvements, which in turn resulted in a formulation of the SaveCCM semantics by means of timed automata, to simplify the integration of efficient analysis tools.

An Adaptive Cruise Controller (ACC) is a further development of a standard Cruise Controller. In addition to the conventional task of maintaining a constant speed, an ACC provides extra functionality to help the driver keep his distance forward to a preceding vehicle, by autonomously adapting the speed of his vehicle to the speed of the vehicle in front.

To exercise the component model further, its complexity has been increased with two non-standard functional extensions. One extension is the possibility of adjusting the maximum permissible speed to accord with speed limit regulations. This feature would require that the ACC system has access to the relevant speed-limit regulations, provided, for example, by transmitters on the road signs or road map information supplied by a Global Positioning System (GPS). The second extension is emergency brake assistance, helping the driver to brake in extreme situations, e.g., when the vehicle in front suddenly brakes or if an obstacle appears on the road.

In the remainder of this section, we describe the development of an ACC application using SaveCCT. The presentation of the design is followed by two examples of how the integrated analysis techniques can be used to evaluate the appropriateness of the design. We also describe the synthesis of an executable system from the design.

5.1. System design

The sources of input to the ACC application can be divided into three categories: the Human Machine Interface (HMI) (e.g., desired speed and on/off status of the ACC system), the internal vehicular sensors (e.g., current speed), and the external vehicular sensors (e.g., distance to the vehicle in front). For the output, we distinguish between two categories: the HMI outputs (providing the driver with information about the system state), and the vehicular actuators for controlling the speed of the vehicle.

The ACC system is designed as a SaveCCM assembly (*ACC Application* in Fig. 5) built from four basic components, one switch, and one sub-assembly. The design of the sub-assembly (*ACC Controller*) is in turn shown in Fig. 6. The roles of the individual elements in the design are

- The *Speed Limit* component calculates the desired vehicle speed based on input from the driver and the speed-limit regulations.
- The role of *Object Recognition* is to decide if there is a car or another obstacle in front of the vehicle, and if this is so, to calculate its speed relative to the vehicle. Based

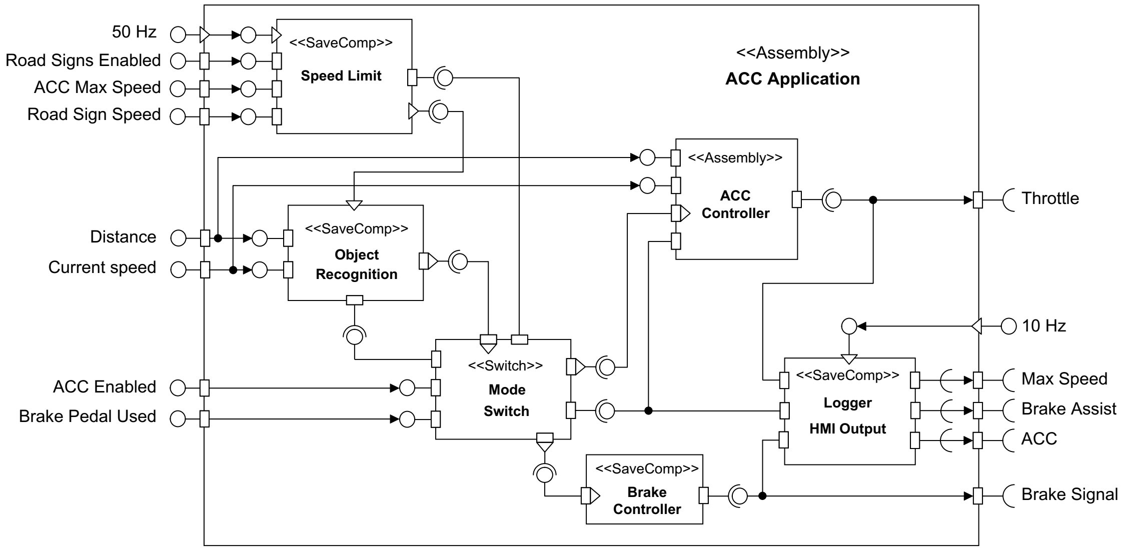

Fig. 5. ACC application design.

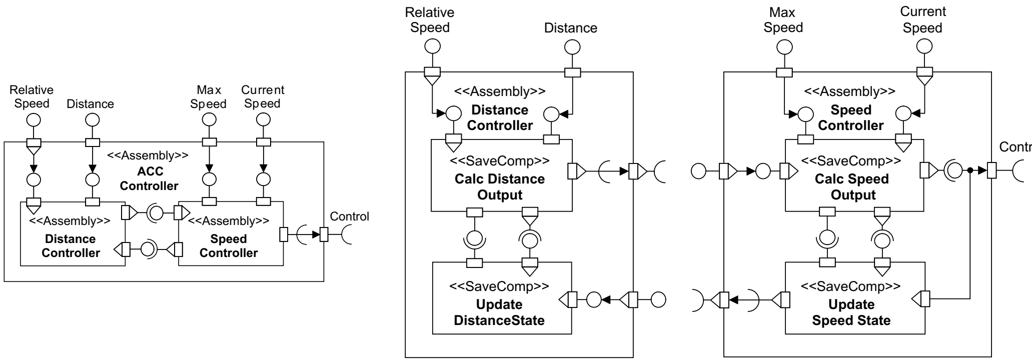

Fig. 6. Internal design of *ACC Controller*.

on these values, the component is also responsible for deciding if the emergency brake assistance functionality is needed or not.

- *Mode Switch* forwards the trigger signal to either the *ACC Controller* assembly, the *Brake Controller* component, or neither of them, depending on the current system mode determined by *ACC Enabled*, *Brake Pedal Used* and information from *Object Recognition*.
- The *Brake Controller* component controls the brake output signal.
- The *Logger HMI Outputs* component is used to communicate the ACC status to the driver via the HMI, and to log the internal ACC settings.
- The *ACC Controller* assembly manages the throttle control of the vehicle, on the basis of the current speed, the desired speed, and the distance to the vehicle in front.

It is worth pointing out that the non-standard functionality of the ACC application (speed-limit awareness and emergency brake assistance) is primarily located in two separate components. The other components can be used throughout the product-line, in product variants with only standard ACC functionality.

The application has two different trigger frequencies: 10 Hz and 50 Hz. Logging and HMI output activities execute at the lower rate, and control related functionality at the higher rate.

The throttle control functionality of the ACC, located within the *ACC Controller* assembly, is particularly important to the overall system quality. Since these calculations are very time critical, delivering the response (throttle status) as quickly as possible is crucial. The assembly is built from two cascaded controllers (see Fig. 6), represented by the sub-level assemblies *Distance Controller* and *Speed Controller*.

This design corresponds to the *control module* concept introduced by Pernebo and Hansson (2002). A control module consists of two sub-structures responsible for *forward* and *backward* activities, respectively. The former is responsible for calculating the output value, and the backward structure updates the state of the module in accordance with the feedback signals. The result is a high-level, flexible building block for control loops. When control modules are combined, for example in a cascade control loop as in the *ACC Controller*, the result is a chain of forward activities that produces the output, and a second chain of state updates that is not performed until the output have been sent to the actuators.

5.2. LTSA analysis

The LTSA tool described in Section 4.1 can be used to check a number of implicit properties, such as the absence of deadlocks and livelocks. These are general properties that can be checked automatically without being explicitly specified by the user. For the ACC design, LTSA can automatically check that deadlocks do not occur and that every action can be performed eventually.

In addition to such implicit properties, we give as examples two explicit properties that can be verified with LTSA. The first property states that the ACC application is safe when disabled, and the second property expresses that the state update activity does not occur before the proper inputs are available, which is required for a correct control loop behaviour.

- *Safe when Disabled*: If the system input ACC Enabled is false, or if the brake pedal is used, then ACC Controller and Brake Assist must be disabled.
- *Control Loop Update*: The triggering of an Update State component is always preceded by a full execution of the corresponding Calculate Output component.

The *Safe when Disabled* property is specified in terms of the actions that can be performed at the ports ACC Enabled, break pedal used and the input trigger ports of *ACC Controller* and *Brake Assist*. It is checked from the derived FSP model of the system by extracting the subsystem formed by *Mode Switch*, *ACC Controller*, *Brake Assist* and the connections between them. The developer can automatically derive an environment for the subsystem concerned. In this case, the environment simply provides the data expected at the input ports of *Mode Switch* the remaining three input ports of *ACC Controller*. It also consumes throttle and brake output data.

To verify *Control Loop Update* it is sufficient to extract the FSP specification of *ACC Controller*, as this is a local property, independent of the interaction with the other components. We specify as valid behaviours of the system all those in which the *Update State* component always reads data from an input port only after that *Calculate Output* has written to its output port.

The two properties were checked on a 1.83 GHz MacBook Pro in 16 s using 121 Mb of memory.

5.3. Analysis using the Times tool

As described in Section 4.2, we use the Save2Times tool to convert the ACC design into a model of timed automata with tasks. The automata model is simulated and verified in the Times tool. In addition to the model of the ACC generated, we have produced an abstract model of the environment that non-deterministically supplies the ACC model with input. The environment model is composed in parallel with the ACC model. In the resulting model, the *Object Recognition* component will be able to switch mode at any time.

The Save2Times tool produces two versions of the ACC model – a version for simulation, and another for model-checking. The simulation model incorporates the program code of the components written in C. This results in a very detailed model that is particularly useful for simulation, since the values of all the variables can be determined during simulations. We use an in-house version of Times that supports tasks programmed in a subset of C (the same subset is supported by version 3.6 of the Uppaal model-checker[8]).

For model-checking, the Save2Times tool produces a more abstract model that preserves inter-component behaviours such as timing of components, data-values of ports, and triggers. This model is useful for model-checking global properties of a SaveCCM model. In the ACC model, the output port *Brake* of the *Object Recognition* component must be retained since it controls the mode switch.

In addition to the two models, the Save2Times tool produces a list of properties that can be checked using Times. All SaveCCM models should normally have these properties. We have checked three kinds of properties of the ACC model and its environment

- *Preservation of triggering*: No trigger input port is activated while the corresponding component is executing. Since input trigger ports are reset when a components execution is completed, the system must have this property to ensure that no triggering is lost. The order of triggering within the ACC model is shown in Fig. 7. Assuming that no triggers are lost, the order can be interpreted as a precedence relation.
- *End-to-End Constraint*: The Throttle port will be updated within 15 ms after the 50 Hz trigger port is activated. When in the *ACC* mode, the ACC controllers are triggered and the output port *Throttle* is updated by the *Calculate Speed Output* component. The *Throttle* port is connected to an actuator controlling the throttle lever. The constraint is of interest because a quick response to input is required for control stability. Such a constraint is checked by annotating the model and introducing an extra clock, as described by Lindahl et al. (2001).
- *Schedulability*: The tasks are guaranteed to meet their deadline. When checking schedulability for the ACC we assume a fixed priority scheduling policy (which is the case in RTXC, the real-time operating system currently supported by SaveCCT), and that the logger com-

[8] For more information about the Uppaal tool, see the web site http://www.uppaal.com.

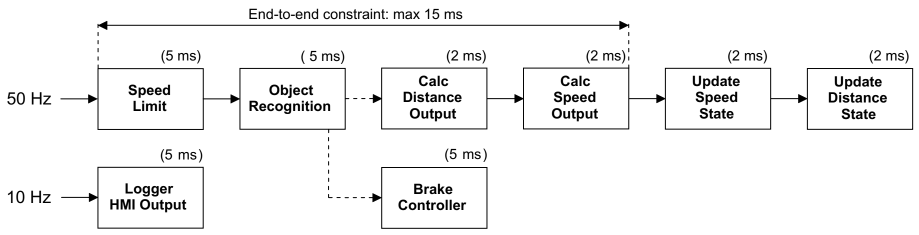
End-to-end constraint: max 15 ms
50 Hz
10 Hz
(5 ms) Speed Limit
(5 ms) Object Recognition
(2 ms) Calc Distance Output
(2 ms) Calc Speed Output
(2 ms) Update Speed State
(2 ms) Update Distance State
(5 ms) Logger HMI Output
(5 ms) Brake Controller

Fig. 7. Component precedence relation, induced by triggering (computation time in parentheses). Solid arrows represent direct precedence, dashed arrows represent conditional precedence.

ponent has the lowest priority. The computation times are shown in Fig. 7. Component code is modelled as tasks, and response time is measured from the triggering of the component. The system is schedulable if the worst-case response time (WCRT) for each task is lower than its deadline. When performing a schedulability analysis, we can extract the actual WCRT of each task. For example, the logger component has the WCRT of 59 ms.

In addition to these properties we can model-check user-specified reachability, liveness, and leads-to properties. For example we can check that the code for the *Calculate Speed Output* component is reachable. A liveness property could state that in all paths, *Object Recognition* will be executed eventually. An example of a leads-to property is that the execution of the *Calculate Distance Output* component will eventually lead to the execution of *Update Distance State*.

The properties were checked using Times installed on a 1.7 GHz PC. Each property was successfully checked in less than 25 s using less than 11 Mb of memory.

5.4. Synthesis

As described in Section 2.2, the automated activity synthesis takes the textual representation of the ACC from the design tool as input, and generates all low-level platform dependent code. The synthesis produced four tasks: one task including *Speed Limit*, *Object Recognition*, and *Mode Switch*; one task including *Logger HMI Outputs*; one task including *Brake Controller*; and one task including the four components in the *ACC Controller*.

To verify the functionality and implementation of the ACC application, we utilised the integrated simulation technique CCSimTech. This enabled execution of the application in a Windows environment, testing and debugging being then performed with observability higher than when using the target hardware. However, to be able to test the whole ACC application it was necessary to develop an environment model and a control panel. The control panel was used to give stimuli (such as accelerator position, brake pedal and ACC settings) to the running system. The environment model was used to simulate the physical behaviour of the system, such as the braking behaviour. Using this test platform, application bugs could be found and eliminated at an early stage.

When moving to the CrossFire MX hardware, we used the compiler with no optimisations. The whole target system was about 115 kb of which approximately 10% was required by the application and the rest by the operating system. The CPU utilisation in the different application modes was 7%, 12% and 15%, respectively.

5.5. Evaluation

Although the interaction between the ACC and the rest of the system is simplified in comparison with a real vehicular system, we believe that the example is sufficiently complex to illustrate key aspects of our approach. Designing the ACC application according to component-based principles was relatively straightforward, and SaveCCM proved sufficiently expressive for this type of system. In particular, the separation of triggering and data connections proved very suitable for control loops, since it was easy to build loops with synchronized forward and backward activities.

The close integration of analysis tools, exemplified by LTSA and Times, enabled us to derive a number of non-trivial properties automatically or with little manual intervention. In particular, the high predictability ensured by the SaveCCM semantics allowed analysis of properties crucial to ensure correct real-time behaviour, such as end-to-end response times. The integration of CCSimTech also provided adequate support for testing.

The resulting system is sufficiently resource efficient. It utilises only a small part of the available capacity of the target hardware, which is approximately the utilisation expected for this application in combination with state-of-practice programming methods (i.e., C and C^{++}). The explicit triggering allows the synthesis mechanism to minimize communication overhead by identifying static triggering patterns. In the ACC example we note, e.g., that the four components in the time-critical *ACC Controller* are bundled up in a single task, with the result that the communication between them is achieved by ordinary function calls, without calls to OS functionality such as semaphores or message queues.

6. Conclusions

We have presented SaveCCT, a component technology supporting component-based development of vehicular systems. Typical application requirements within this domain include resource-efficiency, predictability, and safety. We believe that such cross-cutting concerns should be considered early in the software process and taken into consideration at all stages in the process. This is supported in SaveCCT by enabling easy usage of analysis and verification methods during the whole software development phase, through automated connectivity to tools for analysis and testing. We have illustrated the suitability of SaveCCT through an example application developed in cooperation with our industrial partners. The adaptive cruise controller application has been a recurring example throughout the development of SaveCCT, and has been used for evaluation and guidance for improvements.

The expressiveness of the component model (SaveCCM) seems to be sufficient for efficient application of component-based principles in the domain of vehicular systems. SaveCCM is based on a control-flow (pipes-and-filters) interaction model, combined with additional support for domain-specific key functionality, e.g., feedback control, system mode changes, and static configuration. SaveCCM is predictable enough to permit derivation of specialised formal models, which enables the automated integration of analysis tools. This is an important advantage in the domain, due to the safety-critical nature of vehicular systems.

Resource efficiency is of high importance in embedded systems, and SaveCCT addresses this by an efficient synthesis mechanism. The dynamic component binding of general-purpose component technologies, which allows changes to components and connections at run-time, has been discarded in favour of a more rigid approach where dynamicity is achieved by explicit switch elements. This permits the synthesis mechanism to simplify component communication at compile-time, so that resource efficient run-time platforms can be utilised without additional overhead.

Our future work includes evaluating the usefulness of SaveCCT in a more extensive industrial case-study, and investigating how compatible it is with embedded systems outside the vehicular domain. We also want to extend the number of integrated analysis tools to cover more widely the various requirements in different phases of the development process. Future research in other directions includes integrating the technology with a real-time database mechanism for structured handling of shared data, and with run-time monitoring support.

References

Åkerholm, M., Möller, A., Hansson, H., Nolin, M., 2005. Towards a dependable component technology for embedded system applications. In: Tenth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005). IEEE.

Alur, R., Dill, D.L., 1994. A theory of timed automata. Theoretical Computer Science 126 (2), 183–235.

Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W., 2003. Times: a tool for schedulability analysis and code generation of real-time systems. In: Proceedings of First International Workshop on Formal Modeling and Analysis of Timed Systems. LNCS. Springer-Verlag.

Bengtsson, J., Griffioen, W.D., Kristoffersen, K.J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W., 1996. Verification of an audio protocol with bus collision using Uppaal. In: Proceedings of CAV'96 LNCS. Springer-Verlag, pp. 244–256.

Carlson, J., Håkansson, J., Pettersson, P., 2005. SaveCCM: an analysable component model for real-time systems. In: Proceedings of the Second Workshop on Formal Aspects of Components Software (FACS 2005). Electronic Notes in Theoretical Computer Science. Elsevier.

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

David, A., Yi, W., 2000. Modelling and analysis of a commercial field bus protocol. In: Proceedings of the 12th Euromicro Conference on Real Time Systems. IEEE Computer Society, pp. 165–172.

Dimov, A., Punnekkat, S., 2005. On the estimation of software reliability of component-based dependable distributed systems. In: Reussner, R. et al. (Eds.), International Conference on Quality of Software Architectures (QoSA), LNCS, vol. 3712. Springer-Verlag.

Elmqvist, J., Nadjm-Tehrani, S., Minea, M., 2005. Safety interfaces for component-based systems. In: Winther, R., Gran, B.A., Dahll, G. (Eds.), SAFECOMP, LNCS, vol. 3688. Springer, pp. 246–260.

Fersman, E., Pettersson, P., Yi, W., 2002. Timed automata with asynchronous processes: Schedulability and decidability. In: Katoen, J.-P., Stevens, P. (Eds.), Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol. 2280. Springer-Verlag, pp. 67–82.

Fröberg, J., 2004. Engineering of Vehicle Electronic Systems: Requirements Reflected in Architecture. Licentiate Thesis No. 26, Mälardalen University, Sweden.

Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural mismatch or why it's hard to build systems out of existing parts. In: Proceedings of the 17th International Conference on Software Engineering.

Hansson, H., Lawson, H., Bridal, O., Norström, C., Larsson, S., Lönn, H., Strömberg, M., 1997. Basement: an architecture and methodology for distributed automotive real-time systems. IEEE Transactions on Computers 46 (9), 1016–1027.

Hansson, H., Åkerholm, M., Crnkovic, I., Törngren, M., 2004. SaveCCM – a component model for safety-critical real-time systems. In: Proceedings of 30th Euromicro Conference, Special Session Component Models for Dependable Systems.

Havelund, K., Skou, A., Larsen, K.G., Lund, K., 1997. Formal modelling and analysis of an audio/video protocol: an industrial case study using Uppaal. In: Proceedings of the 18th IEEE Real-Time Systems Symposium, pp. 2–13.

IEC, 1992. International Standard IEC 1131, Programmable controllers.

Larsen, K.G., Pettersson, P., Yi, W., 1997. Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer 1 (1–2), 134–152.

Lindahl, M., Pettersson, P., Yi, W., 2001. Formal design and analysis of a gearbox controller. International Journal on Software Tools for Technology Transfer 3 (3), 353–368.

Lundbäck, K.-L., Lundbäck, J., Lindberg, M., 2004. Development of dependable real-time applications. Arcticus Systems.

Magee, J., Kramer, J., 1999. Concurrency: State Models & Java Programs. John Wiley & Sons, Inc., New York, NY, USA.

Möller, A., Engblom, J., Nolin, M., 2005a. Developing and testing distributed can-based real-time control-systems using a single PC. In: Tenth International CAN Conference, Roma, Italy.

Möller, A., Peake, I., Nolin, M., Fredriksson, J., Schmidt, H., 2005b. Component-based context-dependent hybrid property prediction. In: ERCIM Workshop on Dependable Software Intensive Embedded Systems, Porto, Portugal.

Nierstrass, O., Arevalo, G., Ducasse, S., Wuyts, R., Black, A., Müller, P., Zeidler, C., Genssler, T., van den Born, R., 2002. A Component Model for Field Devices. In: Proceedings of the First International IFIP/ACM Working Conference on Component Deployment.

OMG, 2002. CORBA components. Tech. Rep., Object Management Group, formal/02-06-65.

Pernebo, L., Hansson, B., 2002. Plug and play in control loop design. In Preprints Reglermöte 2002, Linköping, Sweden.

Roll, W., 2003. Towards model-based and CCM-based applications for real-time systems. In: Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. IEEE Computer Society, pp. 75–82.

Sangiovanni-Vincentelli, A., October 2000. Automotive electronics: Trends and challenges. In: Convergence 2000. SAE.

Schmidt, H., 2003. Trustworthy components: compositionality and prediction. Journal of Systems and Software 65 (3), 215–225.

Schmidt, D.C., Levine, D.L., Mungee, S., 1998. The design of the TAO real-time object request broker. Computer Communications 21 (4).

Tivoli, M., Fredriksson, J., Crnkovic, I., 2005. A component-based approach for supporting functional and non-functional analysis in control loop design. In: Tenth International Workshop on Component-Oriented Programming, Glasgow, Scotland.

van Ommering, R., van der Linden, F., Kramer, K., Magee, J., 2000. The Koala component model for consumer electronics software. Computer 33 (3), 78–85.

Wallnau, K.C., 2003. Volume III: A Component Technology for Predictable Assembly from Certifiable Components. Tech. Rep., Software Engineering Institute, Carnegie Mellon University.

Wolf, W., 2002. What is embedded computing? IEEE Computer 35 (1), 136–137.

Yovine, S., 1997. KRONOS: a verification tool for real-time systems. International Journal on Software Tools for Technology Transfer 1 (1–2), 123–133.