Towards a Dependable Component Technology for Embedded System Applications

Mikael Åkerholm[1], Anders Möller[1,2], Hans Hansson[1], Mikael Nolin[1]

[1]Mälardalen Real-Time Research Centre, Mälardalen University, Sweden
[2]CC Systems, Sweden

E-mail: Mikael.Akerholm@mdh.se

Abstract

Component-based software engineering is a technique that has proven effective to increase reusability and efficiency in development of office and web applications. Though being promising also for development of embedded and dependable systems, the true potential in this domain has not yet been realized.

In this paper we present a prototype component technology, developed with safety-critical automotive applications in mind. The technology is illustrated by a case-study, which is also used as the basis for an evaluation and a discussion of the appropriateness and applicability in the considered domain. Our study provides initial positive evidence of the suitability of our technology, but also shows that it needs to be extended to be fully applicable in an industrial context.

1 Introduction

Software is central to enable functionality in mobile phones, cars, airplanes, medical systems, and other products. At the same time, software is also a source of quality problems and constitutes a major part of the development cost. These problems are further accentuated by the increasing complexity and product integration.

Improving quality of Embedded Computer Systems (ECS) is a prerequisite to increase, or even maintain, profitability. Similarly, there is a call for predictability in the ECS engineering processes; keeping quality under control, while at the same time meeting stringent cost and time-to-market constraints. This calls for new systematic engineering approaches to design, develop, and maintain ECS software. Component-Based Software Engineering (CBSE) is such a technique, currently used in office applications, but which is unproven for embedded dependable software systems. In CBSE, software is structured into components and systems are constructed by composing and connecting these components. CBSE can be seen as an extension of the object-oriented approach, where components may have additional interface types than the traditional method invocation of objects. Similarly to objects, simpler components can be aggregated to produce more complex components.

In this paper, we present the ongoing work of devising a component technology for distributed, embedded, safety critical, dependable, resource constrained real-time systems. Systems with these characteristics are common in the automotive, robotics, and automation industries. Hence, we cooperate with leading product companies (e.g. ABB, Bombardier and Volvo) and some of their suppliers (e.g. Arcticus Systems and CC Systems) in order to establish this novel component technology.

Outline: Section 2 provides background on CBSE for embedded systems. In Section 3 we present the current implementation of our component technology, and Section 4 provides an example application that illustrates its use. Based on experiences with the example application, we provide an evaluation of our technology in Section 5. Finally, in Section 6 we conclude and outline some future work.

2 CBSE for Embedded Systems

Research in the CBSE community is targeting theories, processes, technologies, and tools, supporting and enhancing a component-based design strategy for software. A component-based approach for software development distinguishes *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 concerned with assembling components into applications that meet the system requirements. The central technical concepts of CBSE in an embedded setting are:

- *Software components* that have well specified interfaces, and are easy to understand, adapt and deliver. Especially for embedded systems, the components must have well specified resource requirements, as well as specification of other, for the application relevant properties, e.g., timing, memory consumptions, reliability, safety, and dependability.
- *Component models* that define different component types, their possible interaction schemes, and clarify how different resources are bound to components. For embedded systems the component models should impose design restrictions so that systems built from components are predictable with respect to important properties in the domain.
- *Component frameworks*, i.e., run-time systems that supports the components' execution by handling component interactions and invocation of the different services provided by the components. For embedded systems, the component framework typically must be light weighted, and use predictable mechanisms. To enhance predictability, it is desirable to move as much as possible of the traditional framework functionality from the run-time system to the pre-run-time compilation stages.
- *Component technologies*, i.e., concrete implementations of component models and frameworks that can be used for building component-based applications. Two of the most well known component technologies are Microsoft's Components Object Model[1] (COM) for desktop applications, and Sun's Enterprise Java Beans[2] (EJB) for distributed enterprise applications.

Efficient development of applications is supported by the component-based strategy, which addresses the whole software life-cycle. CBSE can shorten the development-time by facilitating component reuse, and by simplifying parallel development of components. Maintenance is also supported since the component assembly is a model of the application, which is by definition consistent with the actual system. During maintenance, adding new, and upgrading existing components are the most common activities. When using a component-based approach, this is supported by extendable interfaces of the components. Also testing and debugging is enhanced by CBSE, since components are easily subjected to unit testing and their interfaces can be monitored to ensure correct behaviour.

CBSE has been successfully applied in development of desktop and enterprise business applications, but for the domain of embedded systems CBSE has not been widely adopted. One reason is the inability of the existing commercial technologies to support the requirements of the embedded applications. Component technologies supporting different types of embedded systems have recently been developed, e.g., from industry [11, 24], and from academia [4, 25] . However, as Crnkovic points out [3], there are much more issues to solve before a CBSE discipline for embedded systems can be established, e.g., basic issues such as light-weighted component frameworks and identification of which system properties that can be predicted by component properties.

Based on risks and requirements for applying CBSE for our class of applications, we have collected a check-list with evaluation points that we have used to evaluate our component technology in an industrial environment. In Section 5 we provide a summary of the evaluation. For more details we refer to [9].

3 Our Component Technology

Our component technology implements the SaveComp Component Model (SaveCCM) [6] and provides compile-time mappings to a set of operating systems, following the technique described in [19]. The component technology is intended to provide three main benefits for developers of embedded systems: efficient development, predictable behaviour, and run-time efficiency.

Efficient development is provided by SaveCCM's efficient mechanisms for developing embedded control systems. This component model is intended to be sufficiently expressive for the needs of embedded control designers, while at the same time being restricted enough to facilitate predictability, dependability, and analysis.

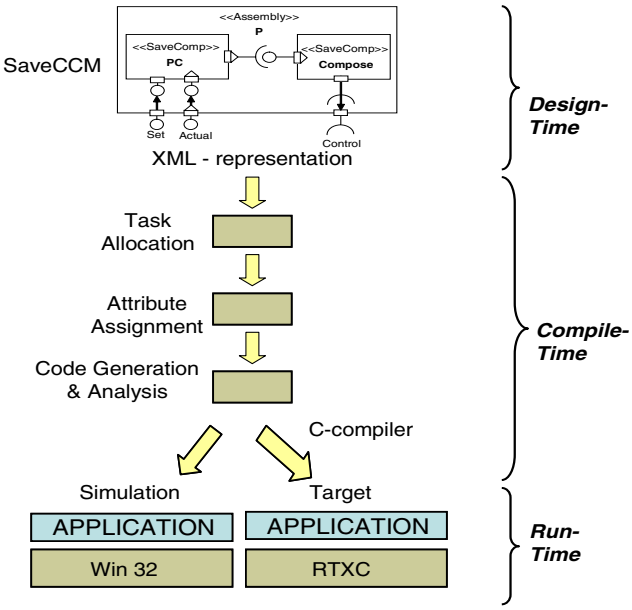


Figure 1. An overview of our current component technology

[1] Microsoft Corporation, The Component Object Model, www.microsoft.com

[2] Sun Microsystems, Enterprise JavaBeans Specification, www.sun.com

Predictable behaviour is essential for dependable systems. In our technology, predictability is achieved by systematic use of simple, predictable, and analysable run-time mechanisms; combined with a restrictive component model with limited flexibility.

Run-time efficiency is important in embedded systems, since these systems usually are produced in high volumes using inexpensive hardware. We employ compile-time mappings of the component-based application to the used operating systems, which eliminates the need for a run-time component framework.

As shown in Figure 1, three different phases can be identified, where different pieces of the component technology are used:

- Design-time – SaveCCM is used during design-time for describing the application.
- Compile-time - during compile-time the high-level model of the application is transformed into entities of the run-time model, e.g., tasks, system calls, task attributes, and real-time constrains.
- Run-time – during run-time the application uses the execution model from an underlying operat-ing system. Currently our component technology supports the RTXC [18] operating system and the Microsoft Win32 [15] environment . The Win32 environment is intended for functional test and debug activities, but it does not support real-time tests since executions are not timely accurate with target system executions.

3.1 Design-Time – The Component Model

SaveCCM is a component model intended for development of software for vehicular systems. The model is restrictive compared to commercial component models, e.g., COM and EJB. SaveCCM provides three main mechanisms for designing applications:

Components which are encapsulated units of behaviour.

Component interconnections which may contain data, triggering for invocation of components, or a combination of both data and triggering.

Switches which allow static and dynamic reconfiguration of component interconnections.

These mechanisms have been designed to allow common functionality in embedded control systems to be implemented. Specific examples of key functionality supported are:

- Support for implementation of feedback control, with a possibility to separate calculation of a control signal, from the update of the controller state. Something which is common in control applications to minimise latency between sampling and control.
- Support for system mode changes, something which is common in, e.g., vehicular systems.
- Support for static configuration of components to suit a specific product in, e.g., a product line.

3.1.1 Architectural Elements

The main architectural elements in SaveCCM are components, switches, and 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. We 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. SaveCCM distinguish between these two aspects, and allow three types of ports:

- Data ports are one element buffers that can be read and written. Each write operation to the port will overwrite the previous value stored.
- Triggering ports are used for controlling the activation of elements. An element may have several triggering ports. The component is triggered when all input triggering ports are activated. Several output triggering ports may be connected to a single input triggering port, providing "OR-semantics".
- Combined ports (data and triggering), combine data and triggering ports. Semantically the data is written before the trigger is activated.

An architectural element emits trigger signals and data at its output ports, and receives trigger signals and data at its input ports. Systems are built from the architectural elements by connecting input ports to output ports. Ports can only be connected if their types match, i.e. identical data types are transferred and the triggering coincides.

The basis of the execution model is a control-flow (pipes-and-filters) paradigm [21]. On a high level, an element is either waiting to be activated (triggered) or executing. In the first phase of its execution an element read all its inputs, secondly it performs all computations, and finally it generates outputs.

Components

Components are the basic units of encapsulated behaviour. Components are defined by an entry function, input and output ports, and, optionally, quality attributes. The entry function defines the behaviour of the component during execution. Quality attributes are used to describe particular characteristics of components (e.g. worst-case execution-time and reliability). 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.

Switches

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. Logical expressions (guards; one for each pattern), based on the data available at some of the input ports, are used to determine which connection pattern that is to be used.

Switches can be used for specifying system modes, each mode corresponding to a specific static configuration. By changing the port values at run-time, a new mode can be activated. By setting a port value to a fixed value at design time, the compiler can remove unused functionality.

Assemblies

Component assemblies allow composite objects to be defined, and make it possible to form aggregate components from groups of components, switches, and assemblies. In SaveCCM, assemblies are encapsulation of components and switches, having an external functional interface (just as SaveCCM-components).

3.1.2 SaveCCM Syntax

The graphical syntax of SaveCCM is shown in Figure 2, the syntax is derived from symbols in UML 2.0 [17] , with additions to distinguish between the different types of ports. The textual syntax is XML-based [26], and available in [9].

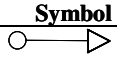
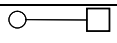
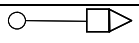
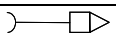
Symbol	Interpretation
	**Input port** - with triggering only
	**Input port** - with data only
	**Input port** – combined with data and triggering
	**Output port** - with triggering
	**Output port** - with data
	**Output port** - combined with data and triggering
<<SaveComp>> **<name>**	**Component** - A component with the stereotype changed to SaveComp corresponds to a SaveCCM component
<<Switch>> **<name>**	**Switch** - components with the stereotype switch, corresponds to switches in SaveCCM
<<Assembly>> **<name>**	**Assembly** - components with the stereotype Assembly, corresponds to assemblies in SaveCCM
	Delegation - A delegation is a direct connection from an input to –input or output to –output port, used within assemblies

Figure 2. Graphical syntax of SaveCCM

3.2 Compile-Time Activities

During compile-time, the XML-description of the application is used as input. The XML description contains no dependencies to the underlying system software or hardware. All code that is dependent on the execution platform is automatically generated in the compile-step. In the compiler, the modules (see Figure 1) that are independent of the underlying execution platform are separated from modules that are platform dependent. When changing platform, it is possible to replace only the platform dependent modules of the compiler.

The four modules of the compiler (task allocation, attribute assignment, analysis, and code generation) represent different activities during compile-time, as explained below.

3.2.1 Task Allocation

During the task-allocation step, components are assigned to operating-system tasks. This part of the compile-time activities is independent of the execution platform, and the algorithm used for allocation of components to tasks strives to reduce the number of tasks. This is done 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 [9].

3.2.2 Attribute Assignment

Attribute assignment is dependent on the task-attributes of the underlying platform, and possibly additional attributes depending on the analysis goals. In the current implementation for the RTXC RTOS and Win32, the task attributes are:

Period-time used during code generation for specifying the period time for tasks.

Priority used by the underlying operating system for selecting the task to execute among pending tasks.

Worst-case execution-time (WCET) used during analysis.

Deadline used during analysis.

The period time, deadline, and WCET are directly derived from the components included in each task. Priority is assigned in deadline monotonic order, i.e., shorter deadline gives higher priority.

3.2.3 Analysis

The analysis step is optional, and is in many cases dependent on the underlying platform, e.g., for schedulability analysis it is fundamental to have knowledge of the scheduling algorithm of the used OS. But analysis is also dependent on the assigned attributes (e.g., for schedulability analysis, WCET of the different tasks are needed).

Examples of analysis include schedulability analysis [1], memory consumption analysis [5], and reliability analysis [20].

Attributes that are usage and environment dependent cannot be analysed in this automated step, since it only relies on information from the component model. There are no usage profiles or physical environment descriptions included in the component model. Additional information is needed to allow such analysis, e.g., safety analysis [22]. Safety is an important attribute of vehicular systems, and we plan to integrate safety aspects in future extensions.

In the current prototype implementation, schedulability analysis according to FPS theory is performed [7].

3.2.4 Code Generation

The code generation module of the compile-time activities generates all source code that is dependent on the underlying operating system. The code generation module is dependent on the Application Programming Interface (API) of the component run-time framework. In the prototype implementation for the RTXC operating system (see Figure 3 a) and the Win32 operating system (see Figure 3 b), the code generation does not target any of the APIs directly. Instead, the automatic code generation generates source code for target independent APIs: the SaveOS and SaveIO APIs [9]. The APIs are later translated using C-style defines to the desired target operating system.

3.3 The Run-Time System

The run-time system consists of the application software and a component run-time framework. The application software is automatically generated from the XML-description using the SaveCCM Compiler. On the top-level, the run-time framework has a transparent API, which always has the same inter-face towards the application, but does only contain the run-time components needed (e.g. the SaveCCM API does not include a CAN interface [8], a CAN protocol stack or a device driver, if the application does not use CAN).

Pre-compilation settings are used to change the SaveCCM API behaviour depending on the target environment. If the application is to be simulated in a PC environment, in our case using CCSimTech [13], the SaveCCM API directs all calls to the SaveOS to the RTOS simulator in the Windows environment. If the system is to be executed on the target hardware using a RTOS (e.g. RTXC) the SaveCCM API directs all system calls to the RTOS.

The framework also contains a variable set of run-time framework components (e.g. CAN, IO, and Memory) used to support the application during execution. These components are hardware platform independent, but might, to some degree, be RTOS dependent. To obtain hardware independency, a hardware abstraction layer (HAL) is used. All communication between the component run-time framework and the hardware passes through the HAL.

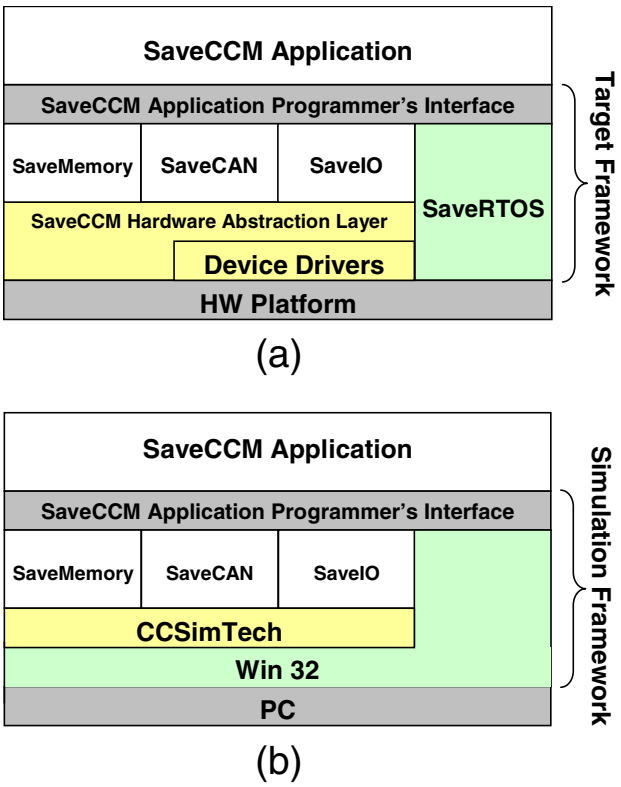


Figure 3. System architecture for target (a) and simulation (b)

The layered component run-time framework is designed to enhance portability, which is a strong industrial requirement [14].This approach also enhances the ability to upgrade or update the hardware and change or upgrade the operating system. The requirements on product service and the short life-cycles of today's CPUs also make portability very important.

4 Application Example

To evaluate SaveCCM and the compile-time and run-time parts of the component technology, we implemented a typical vehicular application: an Adaptive Cruise Controller (ACC). When designing the application, much focus was put on using all different possibilities in the component model (components, switches, assemblies, etc.) with the purpose to verify the usefulness of these constructs, the compile-time activities, and the automatically generated source code. In the remaining part of this section, the basics of an ACC system is introduced, and the resulting design using SaveCCM is presented.

4.1 Introduction to ACC functionality

An ACC is an extension to a regular Cruise Controller (CC). The purpose of an ACC system is to help the driver keep a desired speed (traditional CC), and to help the driver keep a safe distance to a preceding vehicle (ACC extension). The ACC autonomously adapt the distance depending on the speed of the vehicle in front, while keeping the gap large enough to aviod rear-end collisions.

To increase the complexity of a basic ACC system, and thereby exercise the component model more, our ACC system has two non-standard functional extensions; (1) the possibility for autonomous changes of the maximum speed of the vehicle depending on the speedlimit regulations, and (2) a brake-assist function, helping the driver with the braking procedure in extreme situations, e.g., when the vehicle in front suddenly brakes or if an obstacle suddenly appears on the road. Achieving (1) would require actual speed-limit regulations to be known to the ACC system by, e.g., by using transmitters on the road signs or road map information in cooperation with a Global Positioning System (GPS).

4.2 Implementation using SaveCCM

On the top-level, we distinguish between three different sources of input to the ACC application: (i) the Human Machine Interface (HMI) (providing e.g. desired speed and on/off status of the ACC system), (ii) the vehicular internal sensors (e.g. actual speed and throttle level), and, (iii) the vehicular external sensors (providing e.g. distance to the vehicle in front). The different outputs can be divided in two categories, the HMI outputs (returning driver information about the system state), and the vehicular actuators for controlling the speed of the vehicle.

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

Furthermore, there is a number of operational system modes identified, in which different components are active. The different modes are: Off, ACC Enabled and Brake Assist. Off is the initial system mode. In the Off mode, none of the control related functionality is activated, but system-logging, functionality related to determining distance to vehicles in front, and speed measuring are active. During the ACC enabled mode the control related functionality is active. The controllers control the speed of the vehicle based on the parameters: desired speed, distance to vehicles in front, and speed-regulations. In the Brake Assist mode, braking support for extreme situations is enabled.

The ACC system is implemented as an assembly ("ACC Application" in Figure 4 a) 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 4 b.

4.2.1 The ACC Application Assembly

The *Speed Limit* component calculates the maximum speed, based on input from the vehicle sensors (i.e. current vehicle speed) and the maximum speed of the vehicle depending on the speed-limit regulations. The component runs with 50 Hz and is used to trigger the *Object Recognition* component.

The *Object Recognition* component is used to decide whether or not there is a car or another obstacle in front of the vehicle, and, in case there is, it calculates the relative speed to this car or obstacle. 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 (ACC Enabled, Brake Pedal Used) and information from *Object Recognition*.

The *Brake Assist* component is used to assist the driver, by slamming on the brakes, if there is an obstacle in front of the vehicle that might cause a collision.

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 log-memory can be used for aftermarket purposes (black-box functionality), e.g., checking the vehicle-speed before a collision.

The *ACC Controller* assembly is built up of two cascaded controllers (see Figure 4, right), 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 the two controllers to deliver the response for the time-critical computation (throttle lever level) as fast as possible. Hence, the controllers firstly calculate their output values and after these values have been sent to the actuators, the internal state is updated (detailed presentation can be found in [9]).

4.3 Application Test-Bed Environment

In the evaluation, the RTXC operating system is used together with a Cross FIRE ECU[3]. RTXC is a pre-emptive multitasking operating system which permits a system to make efficient use of both time and system resources. RTXC is packaged as a set of C language source code files that needs to be compiled and linked with the object files of the application program.

The Cross FIRE is a C167-based[4] IO-distributing ECU designed for CAN-based real-time systems. The ECU is developed and produced by CC Systems, and intended for use by mobile applications in rough environments.

[3] CC Systems, Cross FIRE Electronic Control Unit, http://www.cc-systems.com.

[4] Infineon, C-167 processor, http://www.infineon.com/.

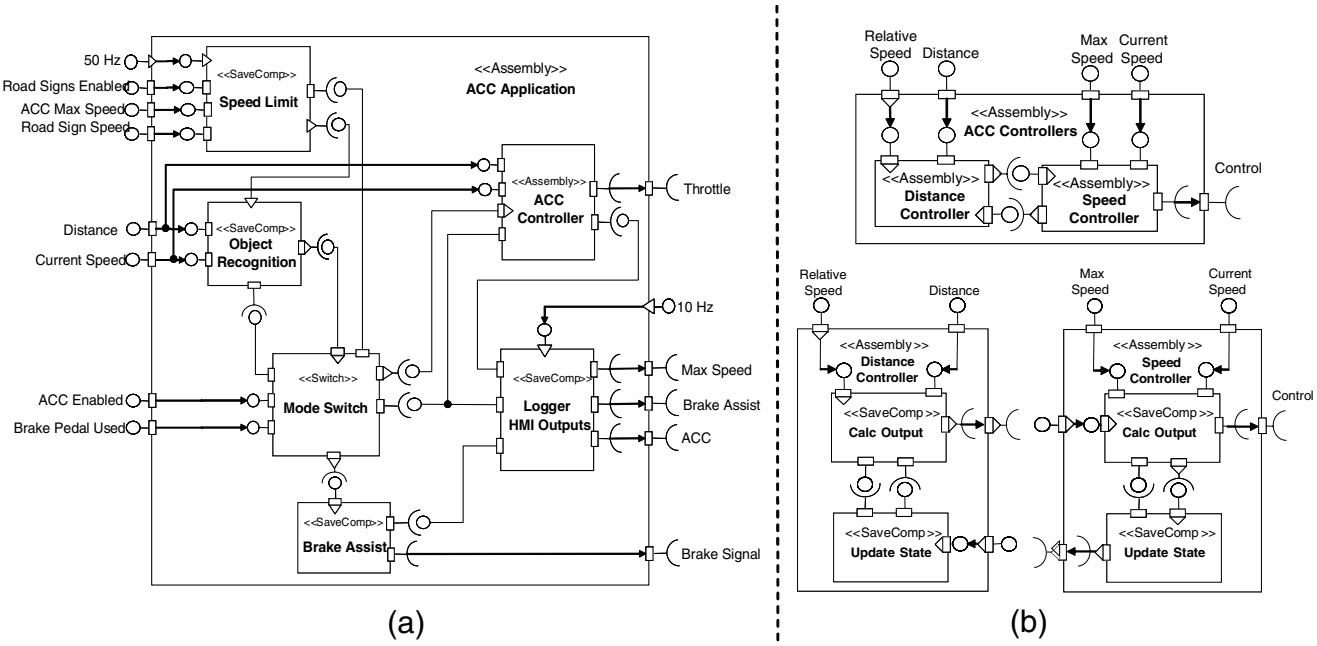


Figure 4. ACC implementation

During functional testing and debugging, CC Systems use a simulation environment called CCSimTech [13], which also was incorporated in this work. Developing and testing of distributed embedded systems is very challenging in their target environments, due to poor observability of application state and internal behaviour. With CCSimTech, a complete system with several nodes and different types of interconnection media, can be developed and tested on a single PC without access to target hardware. This makes it possible to use standard PC tools, e.g., for debugging, automated testing, fault injection, etc.

5 Evaluation and Discussion

CBSE addresses the whole life-cycle of software products. Thus, to fully evaluate the suitability of a component technology requires experiences from using the technology in real projects (or at least in a pilot/evaluation project) by representatives from the intended organisation, using existing tools, processes and techniques.

Our experiment was conducted using CC Systems' tools and techniques. However, we have not used the company's development processes. Hence, we can only give partial answers (indications) concerning the suitability our component technology.

Our evaluation focus on the following three types of properties:

Structural properties concerning the suitability of the imposed application structure and architecture, and the ease to define and create the desired behaviour using the supported design patterns.

Behavioural properties concerning the application performance, in terms of functional and non-functional behaviour.

Process properties concerning the ease and possibility to integrate the technology with existing processes in the organisation.

The adaptive cruise controller application represents an advanced domain specific function, which could have been used as a pilot study at the company. The hardware, operating system, compilers, and the simulation technique, have been selected among the company's repertoire, and are thus highly realistic.

The implementation of the application has not been done according to the process at the company, rather as an experiment by the authors. Thus, it is mainly the structural and behavioural properties that can be addressed. However, to evaluate the process related issues, senior process managers at the company have helped to relate the component technology to the development processes normally used.

The evaluation is conducted using a check-list assembled from requirements for automotive component technologies by Möller et al. [14], risks with using CBSE for embedded systems by Larn and Vickers [12], and from needs identified by Crnkovic [3].

5.1 Structural Properties

Based on the experiment performed we conclude that the component model is sufficiently expressive for the studied application, and that it allows the software developer to focus on the core functionality when designing applications. The similarities with UML 2.0 provide important benefits

by allowing us to use a slightly modified UML 2.0 editor for modelling applications. Also, issues related to task mapping, scheduling, and memory allocation are taken care of by the compilations provided by the component technology, something which gives developers possibilities to concentrate more on application functionality.

Modifications of components are straightforward, since the components have visible source code, and since all bindings between components are automatically generated. However, there is not yet any specific support for maintenance in the component technology.

The ACC system is directrly compilable for both Win32 on a regular PC and RTXC on a Cross FIRE ECU. This is an indication of the portability of our technology across hardware platforms and operating systems. As a consequence, components can be reused in different applications regardless of which RTOS or hardware is used.

Configurability is essential for component reuse, e.g., within a Product Line Architecture (PLA) [2]. In SaveCCM, components can be configured by static binding of values to ports. However, there is currently no explicit architectural element to specify this. In our experiment, we could however achieve the same effect by directly editing the textual representation. For instance, a switch condition can be set statically during design-time, and partially evaluated during compile-time, to represent a configuration in a PLA. A future extension of SaveCCM is to add a new architectural element that makes it possible to visualise and directly express static configurations of input ports. This will additionally facilitate version and variant management.

5.2 Behavioural Properties

With respect to behavioural properties, our component technology is quite efficient. The run-time framework provides a mapping to the used OS without adding functionality, and the compile-time mechanisms strive to achieve an efficient application by allocating several components to the same task. Some data about our case-study:

- The compilation resulted in four tasks: one task including components speed-limit, object recognition, and mode-switch; one task including logger HMI outputs; one task including brake assist; and one task including the four components in the ACC controller.
- The CPU utilisation in the different application modes are 7%, 12%, 15%, respectively for the Off, Brake Assist, and ACC modes, respectively.
- The total application size is 114 kb, of which 104 kb belongs to the operating system, and 10 kb to the application. The application part consists of 2 kb of components code, and 8 kb run-time framework and compiler generated operating system dependent code.

To allow analysis it is essential to derive task level quality attributes from the corresponding component level attributes. In our case-study this was straight-forward, since the only quality attribute considered is worst-case execution time, which can be straightforwardly composed by addition of the values associated to the components included in the task.

Furthermore, the CCSimTech simulation technique proved very useful for verification and debugging of the application functionality.

5.3 Process Related

The process related evaluation concerns the suitability to use the component technology in conjunction with existing processes and organisation, when developing component-based applications. Though process related issues are not directly addressable by our experiment, based on a set of interviews company engineers have expressed the following:

- The RTOS and platform independence is a major advantage of the approach.
- The integration with the simulation technique, CCSimTech, used in practically all development projects at CC Systems, will substantially facilitate the integration of SaveCCM in the development process.
- The maintainability aspects of CBD are attractive, since changes are simplified by the tight relation between the applications description and the source code.
- The tools included in the component technology, as well as the user-documentation, have not reached an acceptable level of quality for use in real industry projects.

6 Conclusions and Future Work

We have described the initial implementation of our component technology for vehicular systems, and evaluated it in an industrial environment, based on requirements identified in related research.

The evaluation shows that the existing parts of the component technology meet the requirements related to them. However, to meet overall requirements, extensions to the technology are needed.

Plans for future work include extending the component technology with support for multiple nodes, integration of legacy-code with the components [10], run-time monitoring support [23], and a real-time database for of shared data [16]. Implementation of more types of automated analysis to determine system attributes from component attributes is also a target for future work. Furthermore, to make the prototype useful in practice, our technology needs to be integrated with supporting tools, e.g., automatic generation of XML descriptions from UML 2.0 drawings, and integration with configuration management tools.

A final indication of the potential of our component technology, and CBSE for embedded systems development in general, is that the company involved in the case-study finds our technology promising and has expressed a keen interest to continue the cooperation.

Acknowledgements

We would like to thank CC Systems for inviting and helping us to realise this pilot project. Special thanks to J. Hansson and K. Lindfors for invitation and to J. Strandberg and F. Löwenhielm for their support with all kinds of technical issues. We would also like to thank Sasikumar Punnekat for valuable feedback on early versions of this article.

References

[1] G. Butazzo. *Hard Real-Time*. Kluwer Academic Publishers, 1997. ISBN: 0-7923-9994-3.

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

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

[4] M. de Jonge, J. Muskens, and M. Chaudron. Scenario-Based Prediction of Run-Time Resource Consupmption in Component-Based Software Systems. In *Proceedings of the 6^{th} International Workshop on Component-Based Software Engineering*, May 2003. Portland, Oregon, USA.

[5] A. Fioukov, E. Eskenazi, D. Hammer, and M. Chaudron. Evaluation of Static Properties for Component-Based Architetures. In *Proceedings of 28^{th} Euromicro Conference*, September 2002. Dortmund, Germany.

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

[7] M. Harbour, M. Klein, and J. Lehoczky. Timing analysis for Fixed-Priority Scheduling of Hard Real-Time Systsems. *IEEE Transactions*, 20(1), January 1994.

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

[9] M. Åkerholm, A. Möller, H. Hansson, and M. Nolin. SAVE-Comp - a Dependable Component Technology for Embedded Systems Software. Technical report, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-165/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, December 2004.

[10] M. Åkerholm, K. Sandström, and J. Fredriksson. Interference Control for Integration of Vehicular Software Components. Technical report, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-162/2004-1-SE, MRTC, Mälardalen University, May 2004.

[11] K.L. Lundbäck, J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. In *Real-Time in Sweden – Presentation of Component-Based Software Development Based on the Rubus concept, Arcticus Systems: http://www.arcticus.se*. Västerås, Sweden.

[12] W. Lam and A. Vickers. Managing the Risks of Component-Based Software Engineering. In *Proceedings of the 5^{th} International Symposium on Assessment of Software Tools*, June 1997. Pittsburgh, USA.

[13] A. Möller and P. Åberg. A Simulation Technology for CAN-based Systems. *CAN Newsletter*, 4, December 2004.

[14] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proceedings of the 7^{th} International Symposium on Component-Based Software Engineering*. 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.

[15] M. MSDN. Win32 Application Programmer's Interface. http://msdn.microsoft.com/.

[16] D. Nyström. COMET: A Component-Based Real-Time Database for Vehicle Control Systems. Technical report, Technology Licentiate Thesis No.26, ISSN 1651-9256, ISBN 91-88834-41-7, Mälardalen Real-Time Reseach Centre, Mälardalen University, May 2003. Mälardalen University Press.

[17] Object Management Group. UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003. http://www.omg.com/uml/.

[18] Quadros Systems Inc. RTXC Kernel User's Guide. http://www.quadros.com/.

[19] K. Sandström, J. Fredriksson, and M. Åkerholm. Introducing a Component Technology for Safety Critical Embedded Real-Time Systems. In *Proceedings of th 7^{th} International Symposium on Component-Based Softwrae Engineering*, May 2004. Edinburgh, Scotland.

[20] H. Schmidt and R. Reussner. Parameterized Contracts and Adapter Synthesis. In *Proceedings of the 5^{th} International Conference on Software Engineering, Workshop on Component-Based Software Engineering*, May 2001. Toronto, Canada.

[21] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall; 1 edition, 1996. ISBN 0-131-82957-2.

[22] D. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. ASQ Quality Press, 2nd Edition, 2003. ISBN 0-87389598-3.

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

[24] R. van Ommering et al. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.

[25] 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.

[26] World Wide Web Consortium (W3C). XML - the Extensible Markup Language. http://www.w3.org/XML/.