Mälardalen University Press Dissertations
No. 60

REUSABILITY OF SOFTWARE COMPONENTS IN THE VEHICULAR DOMAIN

Mikael Åkerholm

2008

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISSN 1651-4238
ISBN 978-91-85485-82-6
Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations

No. 60

REUSABILITY OF SOFTWARE COMPONENTS IN THE VEHICULAR DOMAIN

Mikael Åkerholm

Akademisk avhandling

som för avläggande av Teknologie doktorsexamen i Datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras torsdagen, 29:e maj, 2008, 14.00 i Gamma, U, Högskoleplan 1, Västerås.

Fakultetsopponent: Dr. Lars Grunske, Member of the ARC Centre for Complex Systems, The University of Queensland, Australia.

MÄLARDALEN UNIVERSITY
SWEDEN

Akademin för innovation, design och teknik

Abstract

Component-based software engineering is concerned with enabling software to be assembled through systematic (re)use of carefully built software elements denoted components. In this thesis we describe how reusability benefits of component-based software engineering can be utilized for organizations acting in the vehicular domain. Attractive benefits with this approach include managing complexity through an architecture divided in components and avoidance of large monolithic structures; reduction of time-to-market since applications ideally can be assembled from pre-existing components; increased quality when applications are built from components already proven in use; and cost amortization through investment payoff by each reuse of a component.

Successful deployment of component-based development is however not simple - it depends on many strategic, technical, and business decisions. Furthermore the domain of vehicular systems represents a class of systems where component-based principles have had a limited success, in comparison to the domain of PC applications where the approach has emerged. The major reason to this is a number of important qualities that leaven all through the software life-cycle, e.g., safety, reliability, timing, and resource efficiency.

We have developed a prototype component technology tailored for the vehicular domain. The technology is based on a proposed component-model defining how component-based applications should be built and modelled in the context of vehicular systems. Our solution includes analysis tools and mechanisms supporting the process of maintaining important quality attributes in the life-cycle of software components.

Furthermore, we have used the technology to develop a typical vehicular application, demonstrated its integration with a component repository for vehicular components, and also studied real cases to evaluate our results in cooperation with industry. The results confirm the suitability of component-based principles for the domain, and also show the potential in further development of component technologies for vehicular systems.

ISSN 1651-4238

ISBN 978-91-85485-82-6

To Jenny, Lucas, and Amanda
O.A.V.

Acknowledgements

I have a lot of people to thank for making this big day of my life possible! First of all, I thank my supervisors Ivica Crnkovic and Kristian Sandström for all their time, valuable guidance, friendship, and for giving me the opportunity to perform this studies.

I have had the opportunity to share my time between Ph.D. studies and work in industry. Thank you Jörgen Hansson, Ken Lindfors, and Stefan Rönning, for believing in me, giving me this opportunity, and letting me implement and test my results at CC Systems.

I also want to thank all my co-authors for your guidance, collaboration, and nice trips around the globe. This thesis had not been possible without you. Thanks to my very engaged and pleasant colleagues, Jan Carlson, Igor Cavrak, Radu Dobrin, Johan Fredriksson, Joakim Fröberg, Hans Hansson, Andreas Hjertström, John Håkansson, Rikard Land, Lennart Lindh, Anders Möller, Mikael Nolin, Thomas Nolte, Christer Norström, Peter Nygren, Dag Nyström, Irena Pavlova, Paul Pettersson, Massimo Tivoli, Martin Törngren, Tobias Samuelsson, Johan Stärner, and Mario Zagar.

For fruitful discussions of many new research ideas which obviously not always resulted in publications, help with various tasks, and laughs shared around the coffee table I want to thank all colleagues at the university, especially: Jakob Axelsson, Tomas Bures, Marcus Bohlin, Per Branger, Baran Çürüklü, Harriet Ekwall, Sigrid Eldh, Andreas Ermedahl, Daniel Flemström, Ewa Hanssen, Joel Huselius, Kaj Hänninen, Damir Isovic, Johan Kraft, Magnus Larsson, Stig Larsson, Tomas Lenvall, Markus Lindgren, Åsa Lundkvist. Frank Lüders, Goran Mustapic, Jukka Mäki-Turja, Jonas Neader, Anders Petterson, Larisa Rizvanovic, Severine Sentilles, Daniel Sundmark, Henrik Thane, Peter Wallin, Monica Wasell, Gunnar Widforss, and Aneta Vulgarakis.

Many thanks goes to my workmates at CC Systems' Västerås office for taking good care of me when I was new and inexperienced, for being so good colleagues, for sharing a lot of cookies during the years, and for your feedback on my research when needed: Johnnie Blom, Jonas Ehlin, Carl Falk. Per Görling, Mats Kjellberg, Mattias Lång, Jörgen Martinsson, Andreas Olevik, Malin Olsson, Johan Persson, Göran Sohlman, Ulf Sporrong, Jochen Wendebaum, David Wretling, and Anders Öberg.

I want to thank my mother Margareta and my father Jan-Erik, for your true love, for always believing in me, and for encouraging me to continue studying several times in life. Thanks also to all my relatives and friends that have supported me during this journey, and showed interest in my work that has been really motivating. Finally, I will always be thankful to my family for letting me finish this thesis. It can not always be easy to have a husband or father, who is an industrial Ph.D. student. You have inspired me, and to be honest, the completion of this thesis had not been possible without your advice Jenny! I dedicate this thesis to my loved wife Jenny, and my wonderful kids Lucas and Amanda.

This work was partially supported by the Swedish Foundation for Strategic Research (SSF), via the projects SAVE and PROGRESS, the Knowledge Foundation (KKS) via the graduate school SAVE-IT, and by CC Systems.

Mikael Åkerholm
Västerås, April, 2008

List of Publications

Included publications

Paper A *SaveCCM - a component model for safety-critical real-time systems*, Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, Martin Törngren, *30th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Special Session Component Models for Dependable Systems*, IEEE, Rennes, France, September, 2004

Paper B *Towards a Dependable Component Technology for Embedded System Applications*, Mikael Åkerholm, Anders Möller, Hans Hansson, Mikael Nolin, *10th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, IEEE, Sedona, Arizona, January, 2005

Paper C *The SAVE approach to component-based development of vehicular systems*, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, Massimo Tivoli, *Journal of Systems and Software*, vol 80, nr 5, Elsevier, May, 2007

Paper D *A Model for Reuse and Optimization of Embedded Software Components*, Mikael Åkerholm, Joakim Fröberg, Kristian Sandström, Ivica Crnkovic, *29th International Conference on Information technology Interfaces (ITI 2007)*, IEEE, Cavtat, Croatia, June, 2007

Paper E *Introducing Component Based Software Engineering at an Embedded Systems Sub-Contractor*, Mikael Åkerholm, Kristian Sandström and Ivica Crnkovic, *submitted for publication*

Related publications

These publications are selected among the other not included publications, since they are more related to the thesis. They are frequently refered in the first part, and contributes to the thesis result.

- *Industrial Grading of Quality Requirements for Automotive Software Component Technologies*, Anders Möller, Mikael Åkerholm, Joakim Fröberg, Mikael Nolin, *Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th International Real-Time Systems Symposium*, 2005 Miami, Florida, December, 2005

- *Quality Attribute Support in a Component Technology for Vehicular Software*, Mikael Åkerholm, Johan Fredriksson, Kristian Sandström, Ivica Crnkovic, *Fourth Conference on Software Engineering Research and Practice in Sweden*, Linköping, Sweden, October, 2004

- *Evaluation of Component Technologies with Respect to Industrial Requirements*, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, *30th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering Track*, IEEE, Rennes, France, August, 2004

- *Introducing a Component Technology for Safety Critical Embedded Real-Time Systems*, Kristian Sandström, Johan Fredriksson, Mikael Åkerholm, *International Symposium on Component-based Software Engineering (CBSE7)*, Springer, Edinburgh, Scotland, May, 2004

Other publications

- *Key Factors for Achieving Project Success in Integration of Automotive Mechatronics*, Joakim Fröberg, Mikael Åkerholm, Kristian Sandström, Christer Norström, *Journal of Innovations in Systems and Software Engineering*, vol 11334, Springer, March, 2007.

- *INCENSE: Information-Centric Run-Time Support for Component-Based Embedded Real-Time Systems*, Andreas Hjertström, Dag Nyström, Mikael Åkerholm, Mikael Nolin, *Proceedings of the Work-In-Progress session, 14th Real-Time and Embedded Technology and Applications Symposium*, Seattle, Washington, April, 2007

- *Handling Subsystems using the SaveComp Component Technology*, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, Mikael Nolin, Thomas Nolte, John Håkansson, Paul Pettersson, *Workshop on Models and Analysis for Automotive Systems in conjunction with the 27th IEEE Real-Time Systems Symposium (RTSS06)*, Rio de Janeiro, Brazil, December, 2006

- *Integration of Electronic Components in Heavy Vehicles: A Study of Integration in Three Cases*, Joakim Fröberg, Mikael Åkerholm, *Proceedings from Systems Engineering/Test and Evaluation Conference*, Melbourne, Australia, September, 2006

- *Application of Built-In-Testing in Component-Based Embedded Systems*, Irena Pavlova, Mikael Åkerholm, Johan Fredriksson, *The Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, ACM, Portland, Maine, July, 2006

- *Building Distributed Embedded Systems from Large Software Components*, Mikael Åkerholm, Thomas Nolte, Anders Möller, *Proceedings of the 2nd Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th International Real-Time Systems Symposium (RTSS'05)*, Miami, Florida, December, 2005

- *A Software Component Technology for Vehicle Control Systems*, Mikael Åkerholm, *5th Conference on Software Engineering Research and Practice in Sweden*, Västerås, Sweden, October, 2005

- *Optimizing Resource Usage in Component-Based Real-Time Systems*, Johan Fredriksson, Kristian Sandström, Mikael Åkerholm, *the 8th International Symposium on Component-based Software Engineering (CBSE8)*, Springer, St. Louis, Missouri, May, 2005

- *An event algebra extension of the triggering mechanism in a component model for embedded systems*, Jan Carlson, Mikael Åkerholm, *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA)*, ENTCS, Edinburgh, Scotland, April, 2005

- *Calculating Resource Trade-offs when Mapping Component Services to Real-Time Tasks*, Johan Fredriksson, Mikael Åkerholm, Kristian Sandström, *Fourth Conference on Software Engineering Research and Practice in Sweden (SERPS)*, Linköping, Sweden, October, 2004

- *Software Component Technologies for Real-Time Systems - An Industrial Perspective*, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, *Work-in-progress session of Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December, 2003

- *Attaining Flexible Real-Time Systems by Bringing Together Component Technologies and Real-Time Systems Theory*, Johan Fredriksson, Mikael Åkerholm, Kristian Sandström, Radu Dobrin, *Proceedings of the 29th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Component Based Software Engineering Track*, IEEE, Belek, Turkey, September, 2003

- *A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software*, Tobias Samuelsson, Mikael Åkerholm, Peter Nygren, Johan Stärner, Lennart Lindh, *International Workshop on Advanced Real-Time Operating System Services (ARTOSS)*, Porto, Portugal, July, 2003

- *On the Teaching of Distributed Software Development, Ivica Crnkovic, Igor Cavrak, Johan Fredriksson*, Rikard Land, Mario Zagar, Mikael Åkerholm, *25th International Conference Information Technology Interfaces (ITI)*, IEEE, Cavtat, Croatia, June, 2003

Contents

I

Thesis

Chapter 1

Introduction

The vast majority of new innovative functions in modern vehicles are realized with software! Software controlled functions continue to increase their importance in modern vehicles through improving and replacing electro-mechanical functions and enabling new innovative functions that previously were not imaginable. However, software engineering in the vehicular domain is not without problems. The systems are complex, some high-end cars have about 1400 more or less interconnected software controlled functions distributed on 80 embedded computers interconnected with 5 different networks [13, 49]. In combination with high demands on quality attributes such as safety, reliability, resource efficiency, and timing, the engineering challenges for the software developers are hard to beat. Thus, to handle future and present challenges the vehicular industry needs improved software engineering approaches that will enable increased development efficiency, give support for handling the complexity, and facilitates safety, reliability, and timing assessment.

Component-Based Software Engineering (CBSE) [43, 84, 24] is a promising approach, in which software is composed through (re)using well-defined components. The main benefits are managing complexity through a well-defined architecture divided in components where each component provides certain service for the system, increased development efficiency through reusing already existing components, and quality improvements when components are mature and well-proven. However, new software engineering approaches evolved within the general software engineering community can often not be used without modification for vehicular systems. The general software engineering domain focus on software for PCs, entertainment, the Internet, and office applications, where other quality attributes as security, flexibility, and look and feel might be the most important. Technologies and methodology evolved with these quality attributes in mind must be approached with care by the vehicular industry which today relies on static, simple, robust, light-weighted and well-proven technologies to be able to assess the important domain quality attributes.

In this thesis we investigate and demonstrate how CBSE can be enabled for vehicular control systems. We propose methods that together form a prototype component technology that facilitates important quality attributes of vehicular systems. The technology is demonstrated and evaluated by means of experiments and case-studies in close cooperation with CC Systems [15], a company acting as a sub-supplier in the domain of embedded vehicular systems.

The work has been carried out within the SAVE [77], and SAVE-IT [76], and PROGRESS [70] projects. The main goal with SAVE is to begin establishing an engineering discipline for systematic development of component-based software for safety-critical embedded systems, focusing on a single application area (vehicular systems).

The two subsequent sections provide introductions to basic concepts of CBSE, and vehicular systems, as a foundation for reading the rest of the thesis. The introduction focus on presenting terminology used in the reminder of the thesis, and thus other important parts of CBSE and vehicular systems are intentionally left out.

1.1 Component Based Software Engineering

Research in the Component-Based Software Engineering (CBSE) community is concerned with developing theories, processes, technologies, and tools supporting and enhancing a component-based design strategy for software.

In an idealized view of traditional software development, the software is developed in a sequential process from requirement definition to delivery. CBSE, on the other hand, includes two separate development processes. A component-based approach 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.

1.1.1 Component-Based Processes

A very important characteristic that distinguish component based development from other types of development is separated development processes for system development and component development, which is illustrated in Figure 1.1. In system development, specialized products are built through reuse. While in component development, general components are built for reuse.

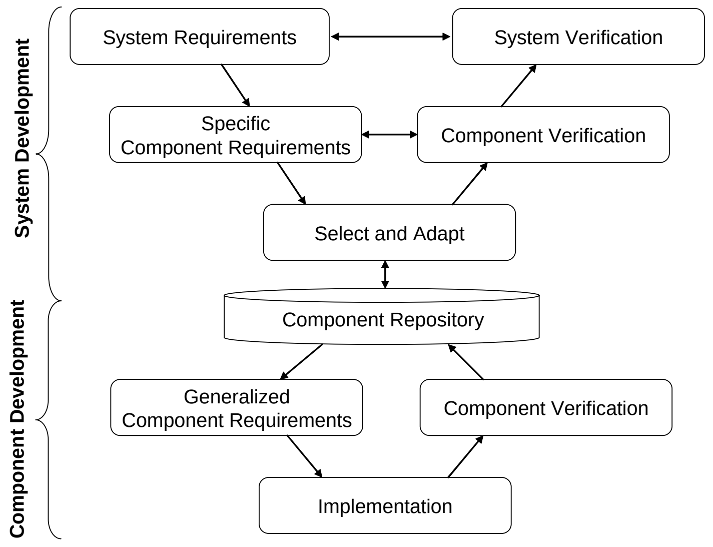


Figure 1.1: A system development process based on the V-model, and its interface to a component development process

The focus to develop reusable components is stressed through the separated component-development process, which ends with delivery to a common component repository where reusable components are stored. Mili et al. [60] provide an elegant definition of reusability of a software components as the aggregation of the quality attributes *usability*, and *usefulness*.

- Usability refers to the ease of reusing the component. Usability is increased by low coupling and well defined carefully documented interfaces, as well as comprehensive testing for different use-cases prior to reuse.

- Usefulness is in this case a measure of how often the software component is expected to be reused. This is clearly depending on the generality of the component.

These attributes must be first class goals for the component-development in order to maximize reusability of the components. The equation to get return of investment in reusable components is depending on the number of times the components are successfully reused. The investment of developing reusable components in comparison to software dedicated to a certain task is hard to estimate in the general case. Studies indicate all from 1.5 to 5 times the effort [64, 60, 22, 84].

In a component-based system development process, the development team focuses on efficient development of a system dedicated for a certain purpose. The goal of the system development process can thus be compared to the goal of a traditional software systems development process, but the major difference is the method to realize the system. As shown in figure 1.1, in a component-based process the lowest step in the V-model is select and adapt, while in a traditional processes the lowest step would be implementation.

If we focus on the difference the select and adapt step usually starts with some steps related to finding components that might provide the required functionality for some part in the project. However, it also requires that the application intended to be built have been divided in suitable component abstractions. This division should preferably be done with reuse of specific components in mind. The process should also be supported by possibilities to query or browse available components.

If candidate components for reuse are found, the selection of which components to reuse is done according to some established criteria. Assessment of candidate components to reuse may be necessary to select between alternatives.

Adaptation might be necessary to fit reuse in the specific context. Many techniques are proposed to support this process, e.g., configuration parameters [17], wrappers [12], adaptors [90], and connectors [51].

1.1.2 Component Technologies

A component technology provides support for assembling component-based software. The overall principles of CBSE are realised through component technologies. A component technology provides support for assembling component-based software. It includes models for how components can be assembled, as well as the necessary run-time support that includes component deployment and interoperation between components. Some of the most widely known component technologies are COM [11] and .NET [16] from Microsoft, and Enterprise JavaBeans [63] from Sun Microsystems.

A component technology often contains various development tools for simplifying the engineering process, it provides the necessary run-time support for the components, and imposes certain patterns for assembling components. Figure 1.2 illustrates the basic concepts of a component technology. It is a photograph of a table top in a playground, on which is placed a tray on which different building blocks can be arranged in different combinations. Besides the tray there is a box where the building blocks are stored, and besides the table there is a chair on which a user of the playground can sit comfortably. This playground will be used as a metaphor for a component technology in the following description of technical concepts.

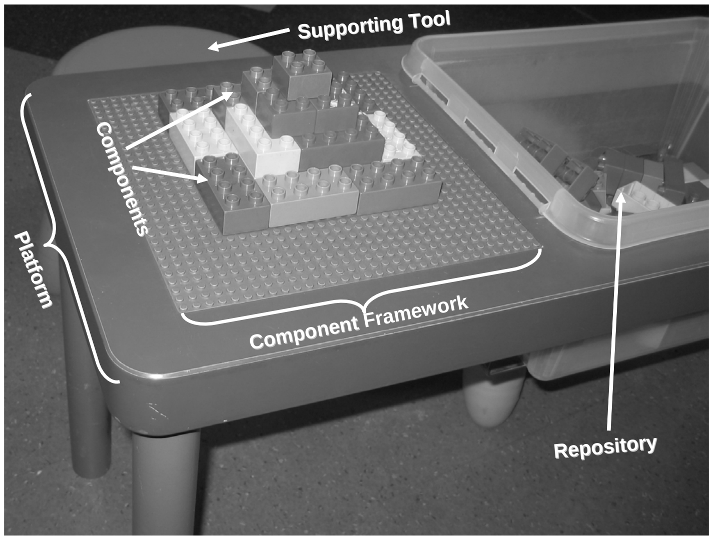


Figure 1.2: A Component Technology for Building Arbitrary Shapes

One of the most important parts of a component technology is the *component framework*, which provides the necessary run-time support for the components not provided by the underlying execution platform (i.e., operating system or similar). In the playground table metaphor, the blocks represent the components, the tray on which they stand represents the framework which provides the components with support, and the table on which the tray stands represents the execution platform. In the metaphor the component framework mainly provides strength to the construction that is not offered by the underlying execution platform. While for software components, the component framework typically handles component interactions, and the invocation of services provided by the components, in addition to providing services frequently used within the application domain targeted by the technology. For example, Enterprise JavaBeans targets distributed enterprise applications and the framework then provides support for database-transactions, and persistence [63]. Component frameworks are often implemented as a layer between the operating system and the component-based application.

A component technology is a concrete realisation of a *component model*. A component model defines the standards and conventions imposed on users of a component technology. It defines different component types that are supported by the technology, possible interaction schemes between components, and clarifies how different resources are bound to components. Compliance with a component model distinguishes a component from other forms of packaged software [5, 24]. In our playground example, the component model is the abstract standards and conventions that the children must follow when assembling blocks because the blocks can only be assembled in a certain pattern. The supplier of the blocks must also follow the component model when manufacturing the blocks, to ensure that the blocks are compatible with each other and the tray on which they are supported.

To be able to efficiently develop component-based applications a *component repository* that is easy to access and browse is necessary. In our playground the repository is ideally placed very close to the tray where the assembly takes place. In contrast to this consider the extra overhead caused by having the building blocks spread out over the floor or having the repository in another room. A repository as closely integrated in the development environment as possible is crucial for development efficiency, and of course, the contents in the repository are very important.

Moreover, *supporting tools* simplifying the work is necessary. Having a chair at the table can be seen as a supporting tool in the playground. The presence of graphical composition languages instead of textual, code-generators, test tools, and analysis tools are examples of supporting tools in the software case.

Finally, *software components* themselves are of basic importance. In the playground example, it is obvious that the blocks represent the components but even in this simple playground metaphor there are philosophical issues which can be subjects of discussion. For example, do several components assembled together to build an element (such as a wall), make a new component or should they be treated as a set of assembled components? This and similar questions continue as subjects of discussion within the CBSE community. Even the definition of a software component remains unclear to date. In Szyperki's book, his attempt to develop a general definition of a software component is compared with no less than fourteen other attempts [84]. One can always question the need for one common component definition, since the component model defines components for a particular technology. Technologies might in turn be intended for different purposes, and as a consequence, different types of components might be suitable. Heineman and Councill propose the following definition, extracted from other definitions, which try to be consistent with the majority of the other ones [43]:

> *A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
>
> Heineman and Councill

From a practical point of view, components should have well-specified interfaces and be easy to understand, adapt and transfer between organisations. The interfaces must handle all properties that lead to inter-component dependencies, since the rest of the component is often hidden from the developer. Components should also be easy to understand, since once created, they are intended to be reused by other developers. The possibilities of reuse of a component are enhanced if it is easy to adapt the component for use in different environments and in combination with different software architectures and other components.

1.1.3 Challenges with Reusing Components

In order to successfully implement component-based reuse in an organization, a component-based process suitable for the business case should be established. A component-technology fulfilling the needs of the domain should be selected or developed. Furthermore, awareness of the known challenges will be valuable in order to maximize the benefits of CBSE in the organization. The overall challenges with reusing software components have been subject for many investigations, e.g., [74, 31, 85, 18, 35, 60, 23, 87, 54, 29, 80, 78, 24, 21]. Below follows a summary of the challenges, mixing different types of challenges as technical and business challenges:

- *Component abstraction* - Defining components so that they can be successfully reused with high benefits in many applications is non-trivial. A big software component would imply a higher benefit associated with reuse, however, a big software component is also typically very information rich which makes it hard to reuse in different contexts. A small software component will be easier to reuse, but the gain with reusing it will be smaller.

- *Component interoperability and composability* - The basic requirement for smooth composition of components is that interacting components have compatible interfaces, e.g., that data types are compatible when data exchanges occurs. However, even if the signatures from syntactical point of view between components are compatible, architectural mismatches may happen due to incompatible semantics, or requirements that are beyond the interface specification. In particular for the domain of embedded systems, safety, reliability, timeliness, and resource efficiency are often very important properties that are hard to determine and often unknown or unspecified on the component level in practice. Models for analysis of system properties based on component properties and properties of the execution environment exists within academia and continuous progress of composability are made.

- *Initial investment need* - There is an investment need associated with developing reusable components in comparison to software in any form for a specific system. When reusable components are developed in the context of a project delivering a software system, these investments are irrational for the project team with the focus on budget and delivery time for the development of a specific product. For a durable focus on reuse the corporate management, even on the very top-levels, must be convinced that developing reusable components will give return on investment, and appropriate taxation schemes to fund groups investing development in reusable components must also be established

- *Additional administration need* - Successfully maintaining a repository with reusable components requires additional administration efforts. Configuration management must be used to keep track of different versions and perhaps variants of components. The components must also be retrievable across multiple business units within large organizations. Problems to locate reusable components outside immediate workgroups, often results in re-implementations of the same functionality since developers simply are unaware of other components.

- *Quality and trust assessment* - Trust becomes important in the decision to reuse a component developed by third-parties. The reason is inability to have a full control of third-party components (due to partial specification, or only partial test coverage). Trustworthiness is not only related to the technical characteristics of a component but also to the component producers, ability to maintain, to provide service, to improve the component in the future. Thus, the challenge is to accurately determine if a component or component supplier is trustworthy and dependable. One possibility might be to rely on certification, another to assess operational records.

- *Responsibility and maintenance* - The responsibility for failures in software systems built from components might be target for discussions, since the origin of the problem is often not in a specific component but a combination of components and usage profile. Maintenance needs also become more complicated when third-party or commercially available components are used.

- *Organizational challenges* - Negotiations regarding technical details between component and system developers might result in reflection of specific system development needs in components, it is a challenge in practice to provide the necessary conditions for component developers to focus fully on reusability. Organizations introducing reuse of components might also need to adapt to put more emphasize on architecture and testing than before and less on implementation.

- *Social challenges* - Experience shows that some programmers seems to have a resistance to reuse other developers solutions. Team rivalry and competitions among business units also complicate reuse of components. A resistance to reuse or accept components developed by other units may be present, which tend to aggravate administration and synchronization problems creating information islands resulting in re-implementations.

As presented above there are several challenges, technical, business-related, and organizational, that are important to be aware of in the efforts to maximize benefits of reusing components. These challenges are also typical targets for research within the CBSE community.

1.2 Vehicular Systems

The application domain in this thesis is embedded control systems in modern vehicles, e.g., forest machines, construction equipment, trucks, and other vehicles. Figure 1.3 gives an overview of the different types of software systems in modern vehicles.

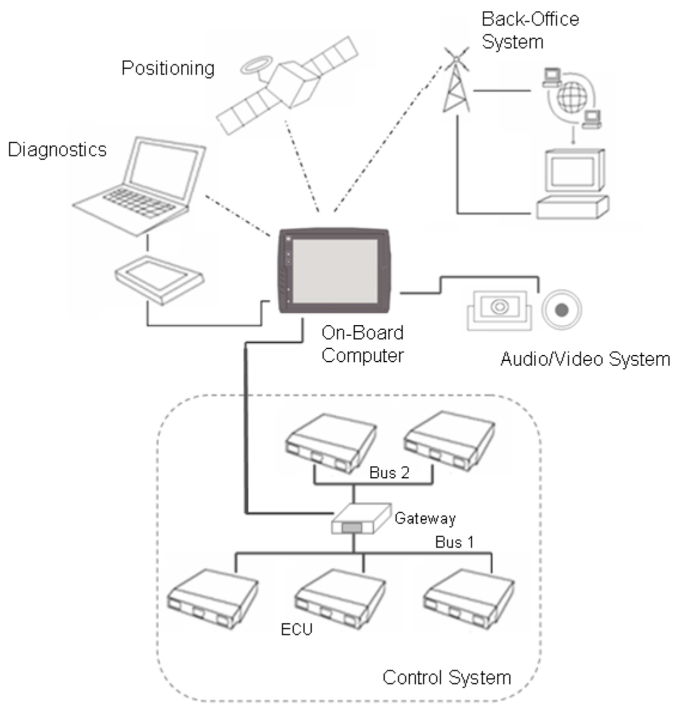


Figure 1.3: Overview the different parts of modern vehicular electronics

- *Control Systems* - These include systems that are embedded and highly critical for the vehicles core functionality controlling, e.g., engine, brakes, steering, and body functions as buckets or cargo platform. These are characterised by high demands on safety, reliability, and hard real-time constraints. There are also some less critical control systems, e.g., electrically powered windows, and air-conditioning. Vehicular control systems can also be further divided into power-train systems, chassis systems, and cabin systems [75].

- *On-Board Computers* - Powerful computers with graphical displays are used as the vehicles main interface for the driver, providing interaction with, and monitoring of, the control systems. They also provide possibilities for information processing as data collection for system diagnostics and prognostics, and communication with others outside the vehicle. These systems are not closely integrated with the core functionality of the vehicle and are easier to replace supplement and remove than control systems.

- *Audio/Video Systems* - Includes systems as reversing and surveillance cameras, and audio for communicating messages to passengers when present and entertainment. Such systems can be after-market mounted but are often integrated with the rest of the electronic systems, e.g., through open standard communication protocols against on-board computers.

- *Diagnostics* - A substantial part of the functionality in the control systems is related to diagnostics, data related to, e.g., service needs and usage are constantly monitored and stored in the vehicle. Thypically it exists an interface for connecting computers that are able to download and process this data, to show operational profiles, and overall service needs.

- *Positioning* - Positioning services are becoming more and more common, for navigation and for storing and communicating positions of, e.g., cargo that shall be picked up, or where it have been left. Positioning also includes real-time traffic information enabling vehicles to avoid traffic jams, and thus also reduce accidents. It is also a core part of fleet-management systems used to synchronize and monitor the operation of vehicle fleets.

- *Back Office Systems* - Back-office systems are systems that are used to administrate vehicles in some form. It may be production data for working machines, position monitoring for trucks, etc. These systems are mainly located in offices, but parts of the systems that communicate with the back office applications must reside in the vehicles.

We limit the scope of this thesis to control systems. These systems are the most critical for the functionality of the vehicle, with maximum demands on qualities beyond functionality, such as timeliness, safety, and reliability. It is these quality attributes that are not addressed by existing mature component technologies (e.g., .NET [16], and Enterprise JavaBeans [63]), and consequently vehicular systems cannot be developed with these existing commercial component technologies. We observe however that the existing technologies might be well-suited in the development of other types of applications in and outside the vehicles, which are more similar to, or might even be, office- and web-applications. The computer nodes in the control systems are designated Electronic Control Units (ECUs), and are often developed by different vendors and use different hardware. The ECUs are interconnected by one or several networks, and often different network technologies are used within the same vehicle. As examples, Volvo Truck Corporation uses two different network technologies and has six to eight external suppliers of ECUs, depending on the type of vehicle, and Volvo Car Corporation uses four different network technologies, depending on model, and has more than ten suppliers of ECUs [33].

1.2.1 A Small Example System

We present typical functionality in a brake system of a modern vehicle as an example of a sub-system to the overall control system. It is a so called brake-by-wire system, which provides several innovative functions in comparison to older brake systems:

- *Anti-lock Braking System* - (ABS) prevents the wheels from locking and simplifying the vehicle steering during heavy braking.

- *Brake blending* - when possible some of the brake force are actuated through engine braking to spare the mechanical brakes. This is especially efficient for vehicles with electric motors, when reversing electrical engines electricity may be generated.

- *Cornering control* - distributes the brake request evenly in corners so that the vehicle behaves neutral even during braking in corners and follows the direction of the steering wheel.

- *Diagnostics* - the performance of the entire brake function is continuously monitored, so that the driver can be aware of malfunctioning and worn parts.
- *Feedback control* - generally improved brake performance, e.g., even braking at all wheels independent of individual differences or worn mechanical parts.
- *Force distribution* - brake force is distributed according to axle pressure, implying an optimized brake request with respect to speed and road conditions with the goal to give the lowest possible stopping distance.
- *Hill hold* - when the driver holds the vehicle with the throttle in a hill, the system automatically switches from using the engine and transmission to using the brakes.
- *Pre-fill* - when the driver suddenly releases the throttle fully. The brake linings are moved closer towards the brake discs to decrease the response time of an expected panic-brake situation.
- *Stability control* - uses the brakes to prevent wheel spin and helps the driver maintain control under heavy acceleration and skids.

The very basic need for a brake system with the functionality presented above would be sensors at the brake pedal to determine brake request from the driver, sensors at the brakes to sense the actual brake pressure, sensors at the wheels to determine individual wheel speed, and actuators for actuating the brake commands. The brake system would also typically need to interact with the transmission control system to know when engine braking is possible, with the suspension to know the axle pressure, with the steering system to know the steering commands, and with the throttle to determine sudden releases of it. Note also that the brake system itself might be distributed over several ECUs.

However, the quality attributes safety, reliability, and timeliness are the most important quality attributes in the system. Achieving these is more important than realizing the value adding functions, and this is enforced through laws and standards, valid for different types of vehicles and markets. Safety must be met through providing evidence that the brake system reduces all potential hazardous events that might lead to accidents to a tolerable level. The reliability of the system to provide the brake function must be must be proven to be above a certain level. Likewise timing requirements must be met for certain functions, e.g., the response-time from pressed brake pedal until actuated brake command is regulated through laws for different markets.

It is value adding functions as presented above that improves brake systems of modern vehicles. Some of them are practically impossible to realize without the use a software based brake system, and interaction with other intelligent software based control systems. But the most important requirements are to provide safe, reliable, and timely operation.

1.3 Outline of Thesis

Chapter 2 gives a summary of the research, and presents the method and results. Related work is presented in Chapter 3. Possible continuations in future work are discussed in Chapter 4. Finally the first part of the thesis is concluded in Chapter 5. Chapters 6-10 contains the included papers.

Chapter 2

Research Summary

This chapter contains a description of the research questions that we attempt to answer in this thesis. In addition we describe the research methods we have used to address them and the results we have achieved. An overview of the research process we have used is shown in figure 2.1. As indicated in the figure, we have followed an iterative approach, since research is generally not a straight forward process. Different phases in the process (i.e., I-V) can be distinguished, and the dotted arrows represents that we have been switching to earlier phases and revised that work based on knowledge and insights gained in later stages. The process is adopted from the methodology described by Shaw in [81], where a method suitable for performing research targeting real-world problems is presented. The method is for the most parts deductive, accordingly the validation becomes a matter of collecting evidence through testing deduced methods in practice. The main activities are:

I Identification of research problems from real-world software engineering. Such problems are often complex, and not suitable subjects for direct research. The research problem is discussed in section 2.1. This is the problem that the contributions of this thesis help to solve.

II Transfer the problem to a research setting, which is a limitation and idealization of the real-world problem, often focusing on certain aspects of the problem. There are several different classes of research settings, each associated with different types of problems, e.g., determining the feasibility of an approach, finding methods to accomplish some goal, or selection between alternative approaches. We have built a research set-

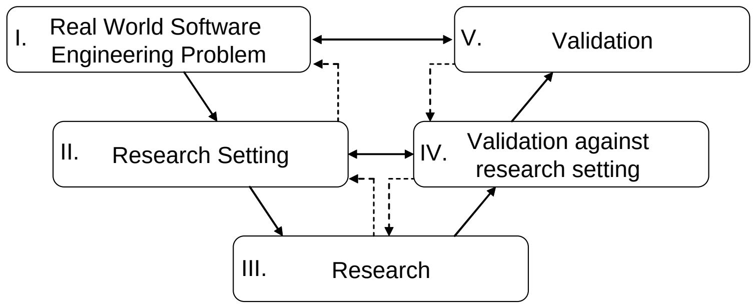

I. Real World Software Engineering Problem
V. Validation
II. Research Setting
IV. Validation against research setting
III. Research

Figure 2.1: Different stages in the applied research process

ting by defining a set of research questions. The details are discussed in section 2.2.

III Performing the research addressing the research setting. In this phase, the work is aimed at targeting a well-defined problem suitable for research, and, depending on the nature of the problem, different methods can be selected. There is a wide range of different methods, from descriptive models of observations to the development of new techniques. Section 2.3 describes the methods and the answers obtained in this thesis.

IV The first step to validate the research results is by demonstrating that the results satisfactorily answer the research questions. This can be done in different ways, e.g., by formal proofs, by implementation of a prototype, by demonstrating the suitability of a method or solution, through controlled experiments, or by persuasion through argumentation. This is described in 2.4.

V The ultimate goal for our validation is to demonstrate that the results can be applied on the real world problem. The only way to achieve this is to solve the real world problem by implementing the results in practice, study and observe the usage, and evaluate the outcome. How, and to which extent, this has been performed is described in section 2.5.

2.1 Problem Definition

The problem we address is to enable efficient usage of CBSE for the domain of vehicular systems, which we see as a matter of devising a suitable component technology. We have introduced the basic concepts of vehicular systems and CBSE in Chapter 1. The introduction shows that CBSE have been developed within the software-engineering community targeting general software as desktop, internet, and entertainment applications. This class of software has fundamentally different important requirements, which makes technologies developed for these systems hard to apply in development of vehicular system with, e.g., safety, timeliness, and low resource consumption as first class requirements.

Efficiently solving this problem is not trivial, as its success depends on the important selection or development of a component technology that can be efficiently used in the development and maintenance process, and at the same time satisfy the run-time and dependability requirements of the vehicular domain. Such a technology would provide the basic needs for the domain to efficiently practice CBSE. However, using CBSE efficiently will also depend of the implementation of processes, business cases, and many other important factors in each specific case.

Many component technologies that might be suitable exists within academia and some are to a limited extent used within industry, e.g., Koala [86] used internally at Philips, Rubus [57] used by some Swedish vehicle manufacturers, and different implementations of the IEC61131-3 standard [46]. However, as pointed out in [19], there is currently no de-facto standard component technology within the domain of vehicular systems; although CBSE seams to get a lot of attention from the industry, e.g., East [26] and Autosar [4]. Thus, it is definitely plausible that the lack of an adopted component-technology is a major reason for the limited success of CBSE in the domain. We have been working under the assumption that one of the reasons for the lack of an adopted component technology is the inability of existing commercial technologies to support the requirements of embedded vehicular applications.

2.2 Research Setting

The research contributes towards a solution to the problem described in the preceding section. We have limited the complexity of the real-world problem, through a research setting defined by research questions.

The first limitation of the scope is to focus on the control related part of the systems. This focus is chosen because CBSE have had a limited success for development of such systems. These systems are the most critical for the overall vehicle functionality, with maximum demands on qualities such as timeliness, safety, and reliability. It is also known that these qualities are not addressed by most existing commercial component technologies, and consequently these systems cannot be developed with such component technologies.

The second limitation is to focus on one node in the distributed control system. This limitation is present to limit the complexity in the research setting; it would probably be desirable to also consider networked control for applicability on the real-world problem. However, this limitation is in-line with state-of-practice. Today the different nodes of a networked control system are often built separately, with asynchronous interaction between the nodes.

As described in the preceding section, the assumption we have been working under is that one of reasons for the lack of an adopted component technology, which should also be one of the reasons for the limited success of CBSE, is the inability of existing commercial technologies to support the requirements of embedded vehicular applications. This leads to our main questions (Q), which have been formulated as follows:

Which should be the key characteristics of a component technology successfully tailored for vehicular applications?

(Q)

This is a broad question, which we intend to answer by separately addressing a number of sub-questions concerning the different parts constituting a component technology, i.e., component models, frameworks, and component storage.

A component model defines abstract rules for how components interact and what they are. In the context of vehicular systems we investigate the answer to the following question:

What should characterize a component model suitable for vehicular systems?

$(Q1)$

This question calls for the key characteristics of a model defining components and possibilities for component interaction, with respect to ease of implementing vehicular control systems, and support for important quality attributes in the domain.

A component framework provides necessary run-time support for the components, threatened by the following question:

Which should be the key characteristics of a component framework suitable for vehicular systems?

$$(Q2)$$

A component framework implements the key mechanisms responsible for many of the run-time properties of applications built on top of the framework. Thus, the component framework has a substantial impact on the overall quality attributes of the application.

Component management covers the management around components that is not part of the component model that has to be set up for organization to be able to work with a component-based strategy, e.g., configuration management and component retrieval. To support these processes the component repository is central, thus, we seek an answer to:

Which should be the characterizing requirements on a component repository suitable for organizations developing vehicular systems?

$$(Q3)$$

The component repository and tools related to the repository can be seen as part of the technology. The question is if the important quality attributes of vehicular systems places special requirements on these tools.

2.3 Results

The main results of the thesis are different parts of a component technology suitable for embedded vehicular systems. Paper A presents the suggested component model SaveCCM. Paper B presents the part of the SaveCCT component technology connecting the design-time models and the run-time models. Paper C outlines a wider view of the SaveCCT component technology, where analysis tools are integrated. Paper D describes a strategy for component management based on component variants in the component repository. Finally Paper E presents experiences from observing some of the results in practice.

The papers have been published and presented in international scientific journals, conferences, and workshops; except paper E, which is not yet published but have been submitted for publication.

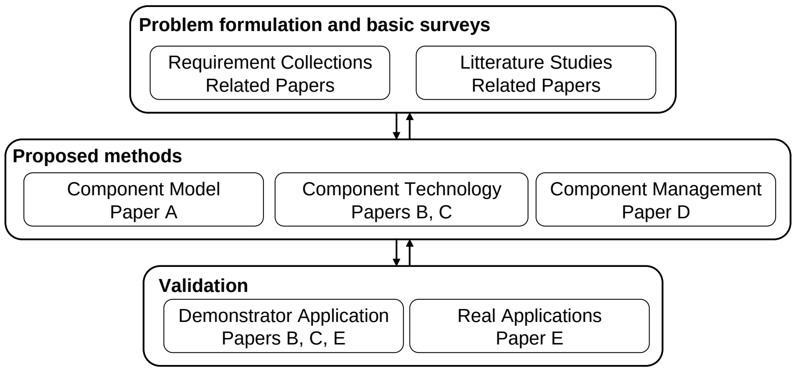


Figure 2.2: The relations between the included papers

An overview of the compound result of thesis is provided in Figure 2.2. As shown in the figure, the result is based on literature surveys and requirement investigations reported in related papers. These are not included in the thesis. The requirements are collected through investigations at companies acting in the vehicular domain. Qualitative interviews with experts at two companies based on open-ended interviews [61], and quantitative form-based surveys with five companies [2]. Important for all researchers is literature studies surveying related work in the research community, e.g., [1, 66], however, monitoring progress within the research community is also a continuous activity not practical to always report in separate surveys.

Paper A

SaveCCM a component model for safety-critical real-time systems, Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, Martin Törngren, *30th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Special Session Component Models for Dependable Systems*, IEEE, Rennes, France, September, 2004

Paper A presents a component model called SaveCCM, intended for embedded control applications in vehicular systems. SaveCCM was specified as an attempt to define an ideal component model for the vehicular domain.

SaveCCM is a simple model in which flexibility is limited to facilitate the analysis of real-time and dependability. The architectural elements are components, switches, and assemblies. Components are the basic units in a design, and shall give the desired functionality. Switches are special components used to statically (during design-time), or dynamically (during run-time), (re)configure component interconnections. Assemblies represent sub-systems and are aggregated behaviour from a set of components, switches, and possibly other assemblies.

The interface of all architectural elements is defined as a set of ports, which are points of interaction between the elements. The interaction is based on the pipes-and-filters pattern, and we distinguish between input- and output-ports, and the complementary aspects data transfer (data-flow) and execution triggering (control-flow). These decisions are an effort to allow easy construction of the key functionality in vehicular control systems, in combination with support for analysis of real-time and dependability properties. Some specific examples of key functionality are: feedback control, system mode changes, and static configuration for product-line architectures.

Method

The research method is development of a component model demonstrating the feasibility of meeting the most important requirements of vehicular applications in a component model. The most important requirements were identified in related papers [2, 62]. The design of the component-model was formed through surveying state-of-the-art, discussions within the Save project, and discussions with other researchers. The concept is demonstrated by the means of a qualitative experiment of modelling a fictive, but realistic, application with the component model; on which the application of real-time analysis is illustrated.

My Contribution

My contribution is active and influencing participation in the forming of the component model and in the writing process. But the model is based on consensus within the Save project headed by the first author Prof. Hans Hansson.

Paper B

Towards a Dependable Component Technology for Embedded System Applications, Mikael Åkerholm, Anders Möller, Hans Hansson, Mikael Nolin, *10th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, IEEE, Sedona, Arizona, January, 2005

In this paper we demonstrate a method allowing both expressive component-based design-time models and resource-effective run-time models by statically resolving resource usage and timing during compile-time. The compile-time step generates the necessary glue code to connect components instead of run-time binding. In order to improve resource efficiency components are mapped to a minimized number of run-time entities, through allocating interacting components to the same run-time entity when possible. The explicit triggering mechanism and the switch elements are key elements to allow this resource efficient mapping. Static triggering patterns can easily be recognized by means of the static triggering chains, and should then be mapped to the same execution entity minimizing the number of context switches and need for communication across task boundaries. Switches clearly show where the application requires dynamicity, and thus the need of division in different executable entities. Attribute assignment is the final step in the compile-time mapping where attributes to the run-time entities is assigned so that high level constraints expressed in the component model are met.

In the prototype implementation we use a commercial resource efficient and predictable real-time operating system as component framework. To further improve resource-efficiency, the compile-time mapping can be further optimised, and more efficient platforms (operating systems) will result in more efficient applications. A drawback with the method as with all compilation is that traceability between the application behaviour during run-time and the design description is decreased, since the compile-time method transforms the component-based design to the execution model of the underlying operating-system. However, this problem is solvable with the techniques used today by debuggers to link execution behaviour to source code.

Method

The research method is development of a prototype demonstrating the feasibility of the proposed model transformation techniques. The purpose is to avoid unpredictable and costly run-time mechanisms in the component framework. The key concept is the clear distinctions between design-time, compile-time, and run-time. The prototype is validated through a qualitative experiment in industry.

My Contribution

My contribution is realization and forming of the model transformations in the compile-time step and the definition of the SaveCCM graphical and textual representation. The implementation of the application, the evaluation work, and the writing is equally shared with the second author Ph. Lic. Anders Möller.

Paper C

The SAVE approach to component-based development of vehicular systems, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, Massimo Tivoli, *Journal of Systems and Software*, vol 80, nr 5, Elsevier, May, 2007

Paper C, presents an overview of a component technology called SaveCCT. Where SaveCCM from Paper A is used, together with the compile-time techniques from Paper B, integrated with existing analysis tools. To efficiently incorporate these tools, as much as possible of the translation from the SaveCCM model to the model required by the desired analysis tool are automated through compile-time techniques in-line with compile-time techniques as introduced in Paper B.

The major improvement over Paper B is in the development-phase, where developers can use a number of available analysis tools with automated connectivity to the design tool where SaveCCM based applications is assembled. In this study we prove the concept of integrating state-of-the-art analysis tools from the research community by the means of automated integration of the Labelled Transition System Analyser (LTSA) [58] and Times [3].

We also stress that designs should be analysed and realizations must be tested. Analysis tools should be used on models during design-time to early get feedback on how different design decisions affects the system. This must also be complemented by testing on the real-system, which we enable at early stages in the project through replacing hardware, run-time platforms, and missing parts of the system with hardware emulation supporting simulation. 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 differ.

Method

This research is a further development of the results in Paper A and Paper B, and it also serves as proof of the concepts proposed in these papers. The SaveCCT technology combines component-based development with related work in the domain of model-based development [48]. We work on models during design, starting from a component-based model in SaveCCM, where transitions to other models is automated, i.e., timed-automata, finite state processes, simulation model, and run-time model. The importance of model transitions in the domain of model-based development is stressed in, e.g., [79]. The feasibility is proven through a qualitative study of a fictive application in cooperation with industry.

My Contribution

My contribution is suggestion of the conceptual integration, through the model generation to integrate the analysis tools in the SaveCCT technology. I also took initiative to write the article. The writing process was administrated by me and Ph.D. Jan Carlson. The detailed parts and application of the analysis tools are shared between the authors.

Paper D

A Model for Reuse and Optimization of Embedded Software Components, Mikael Åkerholm, Joakim Fröberg, Kristian Sandström, Ivica Crnkovic, *29th International Conference on Information technology Interfaces (ITI 2007)*, IEEE, Cavtat, Croatia, June, 2007

Paper D outlines a model that supports component adaptation for scenarios where components must be highly specialized for specific applications. The paper shows how component variants can be used for these application scenarios. The key concept is to store traceability information of the internal realization of components in metadata associated with each component. Using metadata associated with components is an emerging approach within the CBSE community, and a good survey on the topic is presented by Lau and Ukis [52]. In our work we use metadata for maintaining traceability information, which are used for performing impact analysis of desired adaptations. The impact analysis gives guidance to the adaptation work. It gives information of which parts of the component that should be modified. It also specifies which test case that should be used for regression testing after the change, i.e., points out which test cases that must produce the same results.

This method is suggested to be used in combination with a start and a completion step in a component-design process. The completion-phase provides automatic detection of accidentally introduced side effects in redesign. The starting phase supports the selection of the best matching candidate from a repository of components given a set of requirements, and provides guidance for the necessary adaptation work.

Method

The research method is development and proposition of an approach supporting component adaptations taking the requirements of the vehicular domain in consideration. Through surveying existing methods for component adaptation, we found no existing method that was suitable for the problem. The feasibility of the proposal is demonstrated through application on a limited example.

My Contribution

My contribution is the suggested method, with feedback and discussions within the group of authors. The writing process was administrated by me, with review and feedback from the other authors.

Paper E

Introducing Component Based Software Engineering at an Embedded Systems Sub-Contractor, Mikael Åkerholm, Kristian Sandström and Ivica Crnkovic, *submitted for publication*

We present experiences from four cases in this paper. One case, SaveCCT, is a study where a group of researchers demonstrate a prototype component technology in a real industrial environment. The second case, CrossTalk, utilises CBSE principles for realizing a software platform supporting "any" system consisting of the company's hardware. The third case, CC Components, make use of a component repository when possibilities to create or reuse components arise in the development projects. Finally, the fourth study is an evaluation of a method supporting the sometimes necessary work with adaptation of components to fit usage in different development projects.

Our findings indicate that CBSE principles are suitable for vehicular systems, but also that it might be harder to practice CBSE as sub-contractor than as product owner. The most technical needs of expressiveness in the component models, resource efficiency of component based applications, and analysis possibilities can be considered possible to fulfil with a combination of the contents in the different cases. According to our studies the most important need is related to resource efficiency. Resource efficient component frameworks with mature tools together with support for adaptation of software components themselves are needed.

Method

The method is a multiple case study of four cases, where the observer has been participating in the work. The cases, however, differs too much to be able to generalize all observations across all cases, but when applicable the multiple study setup is utilized and observations are validated across cases.

My Contribution

My contribution is study and evaluation of the different cases through participation. The findings are based on feedback from CC Systems and the group of authors. The writing process was administrated by me, with review and feedback from the other authors.

2.4 Questions Revisited

In this section we show how the results provide answers to the research questions, we also discuss certain limitations, which are suitable for further research. There are more detailed presentations of the contributions in the included papers referred to in the presentation.

Q1

What should characterize a component model suitable for vehicular systems?

Based on the SaveCCM component model suggested in Paper A, we have implemented and evaluated a prototype component technology, which have been used to implement a fictive but realistic application. The prototype evaluations are reported in Paper B, C, and E. The following key characteristics are our answer to this question.

The component model should be limited and restrictive to support important quality attributes, e.g., safety, real-time, and reliability properties. Through surveying the industry we have showed that the most important technical requirements in development of vehicular systems are related to dependability characteristics, such as safety, reliability, and testability, e.g., [2, 61, 62]. Today, these quality requirements can only be met by systematically favouring simplicity and predictability in all design decisions. This is obvious by studying, e.g., the main safety standard for electronic programmable systems IEC61508 [45], where all dynamic constructs as pointers, dynamic run-time binding, and artificial intelligence is considered unsuitable for building safe and reliable systems. We have demonstrated that with SaveCCM it is possible to create usable component models that are predictable enough to allow derivation of specialised formal models, which enables automated integration of analysis tools, e.g., Times [3] and LTSA [58]. The suitability of our component model as base for applying state-of-the-art analysis techniques is also proven through the work by Grunske [37], where safety properties are evaluated using SaveCCM.

The pipes-and-filters interaction style is a suitable basis for a component model targeting the vehicular domain. The pipes-and-filters interaction style was chosen to give good support for expressing the key functionality of control systems. This interaction style is similar to the commonly used block-diagrams in control system models, e.g., in the function blocks of the IEC61131-3 standard [46]. This interaction mechanism is also successfully used in the vehicular industry in the Rubus component model [57], which have been a major source of influence. Designing the fictive application with SaveCCM was relatively straight-forward, and SaveCCM proved sufficiently expressive for this type of system. The key functionality demonstrated in the application was feedback control, system mode changes (dynamic configuration), and static configuration.

Q2

Which should be the key characteristics of a component framework suitable for vehicular systems?

Paper B and Paper C present the component framework of SaveCCT that we have used as basis for the answers to this question.

In contrast to the flexibility offered by component frameworks for desktop and entertainment applications, a component framework for the vehicular domain should be based on simple, predictable, static run-time mechanisms. It is known that quality attributes are interdependent, and that some contradictory attributes such as flexibility and predictability are difficult to support simultaneously [6, 40]. The component framework has a substantial impact on the overall quality attributes of the application. The most important technical requirements in the domain are related to dependability characteristics [2, 61], which must be supported by the component framework. These qualities are cross-cutting concerns that must be considered in all parts of an application. Dynamic constructs as pointers, dynamic run-time binding is unsuitable for building safe and reliable systems [45]. For reliable support of analysis of the run-time behaviour, it is important to base the framework on predicable and deterministic mechanisms. In the SaveCCT prototype, this is achieved using a predictable real-time operating system (RTOS) as underlying platform.

The component framework should use resources efficiently. By resources we mean shared limited run-time resources, e.g., processor and memory capacity. Resource-efficiency, (the consumption of a minimum of resources in achieving an objective) is a quality attribute that is important to stress when mentioning component frameworks and vehicular systems, it is recognized as a current need in the commercially available technologies [20]. Most mature commercial frameworks in component technologies are developed for use with personal computers and network servers, where resource consumption is a minor concern, e.g., Enterprise JavaBeans in the J2EE 1.3.1 release requires 128 MB RAM [59], while typical hardware used in vehicular systems can have 128 kB [67]. Paper B and Paper C demonstrates our approach in meeting resource efficiency through code-generation 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.

Q3
Which should be the characterizing requirements on a component repository suitable for organizations developing vehicular systems?

Paper D and Paper E present our findings related to component storage and repositories.

In comparison to storage of software components for desktop and entertainment applications, vehicular systems should have support for creating and storing specialized variants of components. The foundation of CBSE is that general components are reused in many applications. However, requirements of vehicular systems make it harder to reuse the same component in different projects. Software must often be optimized and tailored for each application [20], due to high volumes implying that smaller memory capsules and cheaper processors has high impact on the total production cost. But also due to safety-requirements expressing that no "dead code" should be present in source files or on target for certification according to the higher SIL levels of, e.g., IEC61508 [45]. In practice components suitable for reuse without modification repeatedly in project after project is rare under these circumstances, a component must each time it is used solve the intended task as efficient as possible, and not contain any unused functionality. In paper D, a method supporting the need to create specialized variants of components are suggested. The method causes no overhead in the internal realisation of a software component, allowing the components to be highly specialized for every scenario

2.5 Validation

Validation has been performed through four case studies, reported in Paper E. Two of them are real industrial cases and the other two are demonstrations and evaluations by researchers in industrial environments. The studies have been performed at CC Systems' engineering sites in Finland and Sweden. CC Systems is acting in the vehicular domain, and develops electronics targeting vehicles and machines in rough environments. From software and hardware controlling safety critical by-wire functions, to software and hardware for powerful on-board display based information systems with back-office connections. We have according to the scope of the thesis, focused on control related parts of the systems, and have not considered networked control.

- Case 1, is a continuous study of the usage of SaveCCT in an adaptive cruise controller application, which has been continuously adjusted and evaluated. From demonstrator application in Paper A, through implementation and realisation in an environment representative for projects at CC Systems in Paper B and Paper C.
- Case 2, is a real case driven by CC Systems where a component-based strategy is used to realize product-line architecture for control system platforms.
- Case 3, is also a real case at CC Systems where a company-wide component repository for reusable software components is used.
- Case 4, is an evaluation of a prototype supporting the work related to adaptation of software components, which have been integrated in the repository of Case 3.

Paper E presents the experiences from participating and observing these cases. These studies strengthen some of the answers to the questions Q1-Q3, which leads to the following answer to the main question Q.

Q
Which should be the key characteristics of a component technology successfully tailored for vehicular applications?

Under the limitations in this thesis, and based on the findings reported in the thesis. We conclude that a component technology for the vehicular domain should have the following key characteristics:

- The component model should be limited and restrictive to support important quality attributes, e.g., safety, real-time, and reliability properties. These qualities are most important and can only be met by systematically favouring simplicity and predictability in all design decisions. In both Case 1, and Case 2, component models that are limited and restrictive are used. We have demonstrated that it is possible to create a usable component model that is predictable enough, even for automated application of state-of-the-art analysis tools from the research community. The analysis capabilities is so far not proven in real projects with real applications.

- A good choice is to base the component model on the pipes-and-filters interaction style. The style has been chosen to give good support for expressing the key functionality of control systems, in SaveCCM. Designing the fictive application in Case 1 according to component-based principles was relatively straight-forward. Studying Case 2 fortify our conclusion that this is a suitable component interface. Numerous control systems have successfully been built for vehicles and machines based on the CrossTalk concept. However, note that we cannot show, and do not claim, that any other interaction style is unsuitable.

- The component frameworks must use resources efficiently, mature commercial component technologies with resource efficient component frameworks is identified as a need for the domain today [20]. In Case 1 we have demonstrated a possible approach for far more efficient component frameworks than available in most commercial technologies. This is achieved through code-generation during compile-time, where the explicit triggering of the design-time component model allows efficient transition to the run-time model, e.g., in our case a real-time operating system with tasks, system calls, and task attributes. Studying Case 2, where a mature commercial technology is used, for some projects it can be concluded that it should be possible for more resource efficient implementations with lower level programming of the hardware.

- The component frameworks must rely on simple, predictable, static run-time mechanisms. Achieving safety, timelines, and reliability are cross-cutting concerns that must supported in the component framework. Dynamic constructs as pointers, dynamic run-time binding are unsuitable for building safe and reliable systems. Analysis applied in Case 1, rely heavily on the run-time mechanisms in the framework.

- To improve reusability of software components for vehicular control systems, support for creating specialized variants of components is needed. Instead of being based on general components, applications must be dedicated and specialized to its task for high volume products. Safety critical applications are even worse since no "dead code" is allowed in source files or on target to achieve higher safety integrity levels. It was demonstrated in Case 4, how the adaptation needs of Case 3 can be satisfied. We note that the adaptation needs might be higher for the business case of a sub-contractor company, in comparison to product owners which have the advantage of being able to plan new products based on reuse. However, it can be concluded that the adaptation needs is higher for safety critical embedded systems than office or entertainment systems without resource constraints and safety requirements.

The validation activities are so far limited to observations at a single company. However, they have been performed at a sub-contractor company working in close cooperation with many different customers in different countries. This implies that projects are influenced by the specific requirements of the different customers, major customers requirements also affects internal processes. Thus we claim that the validation activities have also been target for some form of variations within the domain.

We have not used the prototype realizations of the results in real projects; the prototypes have been proven and demonstrated together with tools, processes, software, and hardware from the company's repertoire. We are also aware of limitations of functional capabilities of our demonstrator application; a real adaptive cruse controller should most certainly need to be more capable in, e.g., diagnostics, fault management, and calibration. Thus, the modelling capabilities, the model transitions, and application of the different analysis techniques have so far been proven on limited examples. To initiate validation of our proposed technology in the real world, we have exploited similarities with parallel cases used in real projects.

In general the results are independent of the vehicular domain, control applications, and business situation. The focus is on facilitating quality attributes that are important in the vehicular domain, e.g., safety, reliability, timeliness, and resource efficiency. Thus the technology as such should be attractive to any component-based initiative where these quality attributes are important. Probably the technology suits several applications in the wider domain of embedded control systems. The focus on the vehicular domain is according to the strategy of the Save project, to start by focusing on a limited domain.

Chapter 3

Related Work

Component technologies are developed for a wide range of applications, a broad survey of different technologies is compiled by Lau and Wang [53], which is concluded by a classification of the different types of component models based on run-time or design-time composition and whether the components in the model have an explicit relation to a repository. Here we relate our proposed technology to component technologies targeting embedded systems, together with other research that relates to the results in this thesis.

The Rubus Component Technology [57], which originates from the research around Basement [39], is commercially available and 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. In relation to our proposed component technology Rubus main limitations are that the static scheduling approach only supports periodic activation and that timing aspects are the only extra-functional properties considered. However, current research, contemporary to our work, have recently resulted in version 3 of the Rubus component model with major enhancements [41] where these shortcomings have been targets for improvements.

From Koala [86], we have adopted the idea of 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 developed by Philips, and used as input for further development in the projects Robocop [73] and Space4U [83] with Philips and Eindhoven Technical University as main actors. The focus of these projects is on, e.g., analysis, fault prevention, power management, and terminal management. But compared to the technology in this thesis they are geared towards less safety-critical applications, such as consumer electronics.

An ongoing project with similar goals, but in comparison to the work in this thesis has taken a different approach, is Predictable Assembly from Certifiable Components (PACC) [69] at the Software Engineering Institute. The project focuses on adaptation of component technologies to achieve predictable assemblies. Their concept of Prediction Enabled Component Technologies (PECT) [88] describes a concept for integration of component technologies and analysis techniques. Rather than being a concrete technology, PECT describes possibilities to restrict 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. Within this project the PIN component technology [44] has also been developed, targeting safety-critical embedded real-time systems. A key difference in comparison to our technology is in the design choices that forms the run-time attributes of the technology, PIN components are Dynamic Link Libraries (DLLs) on Windows NT, Windows CE or Unix. Thus the possible application domain is on larger systems, but not for hard real-time systems.

PECOS [65] is one of the component technologies targeting the automation industry. It emerged from a joint ABB and academia project focusing on developing a component technology especially for field-devices, i.e., small reactive embedded systems. PECOS is similar to our technology in the sense that it considers extra-functional properties in order to enable analysis. PECOS provides means for specifying component properties, but does not provide support for analysis of these properties on component or system level.

The IEC61131-3 standard [46], defines a graphical language that can be used for composition of components. The language uses the same pipes-and-filters interaction model between components as we propose, but support for analysis of extra-functional properties does not exist in the standard, e.g., the semantics of the different elements is not formally defined. An addition to the standard released 2005 is IEC61499 [47], which extends the function blocks to allow encapsulation of functions created with IEC61131 and languages as C and Java. It also introduces separation of control flow and data flow, similar to the component model proposed in this thesis.

Another approach to create a component technology for vehicular systems would be to take an existing mature technology from the PC domain, and modify it to meet the requirements of the vehicular domain, e.g., Lüders et. al. shows that COM can be customized to maintain important quality attributes of industrial control systems on a satisfactory level [55, 56]. Another example is

Think [28], a C implementation of Fractal [14] with focus on component-based assembly of operating system kernels for embedded systems.

Important in our proposed concept is analysis of system level properties, based on properties of components and the execution environment. This type of analysis has many related works within the CBSE community; it is part of the important compositionality goal of CBSE research. Larsson et al. [50, 25] divides a large number of quality attributes based on how they can be predicted on the system level; here it is motivated that some attributes can be directly composable from component attributes while others requires usage profiles or system properties of the execution environment. The specification of quality attributes associated with components becomes an elementary need. Shaw propose to specify quality attributes as tuple with type, value, and credibility [82], which is the approach we have adopted. Grunske presents a framework for applying quantitative analysis of several quality attributes of component-based architectures [38]; the framework exploits commonalities between the different analysis methods and utilises that quality attributes are interdependent through exchanging information between different analysis methods. A key attribute to apply real-time analysis is Worst Case Execution Time (WCET). This attribute is clearly depending on the execution environment and can be refined when taking into consideration a usage profile. Fredriksson et al. show how WCET can be packaged and reused for different usage-profiles, and used to achieve less pessimistic and tighter analysis [34]. Safety is one of the most important quality attributes in vehicular systems. In this thesis there is however no results related to assessment of safety. Grunske has demonstrated a method to the assessment of safety properties in combination with the component model proposed in this thesis [37]; the method is based on assigning failure propagation model for each element of the SaveCCM specification. Elmqvist and Nadjm-Tehrani [27] propose the use of safety interfaces to achieve compositionality of safety attributes without need to assess the internal realization of components during composition. Reussner et al. [72] show how reliability can be calculated using Markov chains using a compositional approach, so that the system reliability can be analyzed by using and reusing the reliability information of the components. Furthermore, among other contributions in the field of quality attributes of component-based applications, real-time attributes are considered in [9, 92], and memory consumption attributes in [32, 41].

We also found that components might need to be adapted to suit different applications within the vehicular domain, where applications often require a high degree of specialization. This problem has also been recognized in related research, and a classification of different techniques is presented in [42].

Common for many of these techniques is the support for configuration of components, e.g., [10, 17, 71, 91]. However, the flip-side is that future scenarios must be predicted, and that the configuration code increase complexity and thereby resource usage. The other main principle for existing techniques is to apply external adaptation through wrappers [12], or adaptors [90]. The main limitation here is that optimization of the component's internal realization is not possible, e.g., it is not possible to remove functionality. Another drawback is that the adaptor, or wrapper, code must also be incorporated in the system level analysis [8]. Thus, these techniques are not suitable for resource constrained embedded systems.

Our proposed method to achieve support for adaptation of components is based on storing traceability information of the internal realization of components in metadata. Using metadata associated with components is common in many component technologies [52], e.g., the MS .Net component model [16] uses metadata for certain run-time properties. Our proposal is inspired by the work by Orso et.al. [68] suggesting to (re)use component metadata supporting software engineering tasks. They demonstrate how metadata can be used to improve the test phase. In our method we also reuse test specifications in metadata, this is similar to the ideas around Built-In-Test (BIT) [36, 30, 89, 7], however, we rely on specifications in metadata instead of executable test cases embedded in the components mainly due to needs for high resource efficiency.

Chapter 4

Future Work

For future work it would be interesting to explore more about the impact from the business situation on CBSE. In the domain of control systems for vehicles and machines we can identify three major business situations *sub-suppliers on contract basis*, *COTS suppliers*, and *product owners*. Note that it might be possible to study all these within a single company, such as e.g., CC Systems, hopefully with increased possibilities to limit influences from other differences. This could be done by extending the model presented by Mili et al. [60], shown to the left in Figure 4.1 to also cover embedded systems and possibly also sub-contractors as showed to the right in the figure.

According to Mili et al a typical software system consists of 20 % reusable generic components, which are reusable across several domains. These are typically de-facto standard libraries that can be treated almost as extensions to the programming language. Then 65 % is typically domain specific components that are reusable within the domain, e.g., if the domain is data storage then database components and query languages are typically very reusable within the domain. The 15 % at the top is application specific software that is hard to reuse. How would this apply for the domain of embedded systems? Due to the high degree of application orientation and specialization for certain tasks, it can be expected that there is an embedded factor that can be added to the top-most 15 %. Likewise in the business situation of sub-contractors in general, the reusability of components might in practice be dependent on new orders from specific customers, this might also be modelled as a factor adding to the top most 15 %.

It is also necessary to deploy more of the results in this thesis in practice,

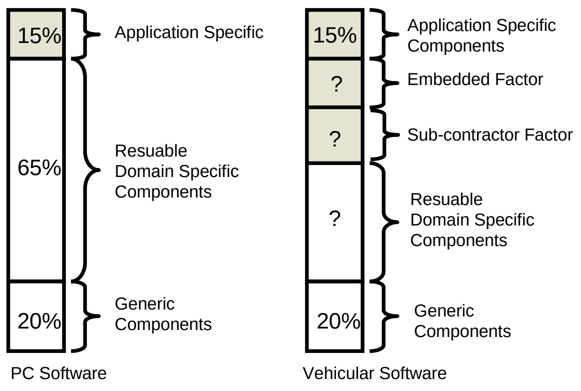


Figure 4.1: Speculation on reusability of software components in the PC domain compared to the sub-contractors of embedded systems

to continue the validation activities. As well as continue to studying ongoing CBSE implementation projects at several actors in the vehicular industry, to understand more of the needs of the domain.

The application of CBSE to distributed control was out of the scope for the thesis. This would, however, be an important area for future research. It would be valuable for the software engineering activities with transparent support for distributed control, preserving reliability, safety, and timeliness. This support would open possibilities for more freedom in deployment of components to ECUs, e.g., deployment to efficiently utilise available processor capacity all over the vehicle network. However, it might also enable new innovative functions, e.g., possibilities to enhance reliability through transferring functions of a certain important ECU to another ECU upon malfunctioning hardware etc.

The thesis focus on control systems, however, the diversity of software systems in modern vehicles will most likely also require that different technologies are used for different types of systems. Interoperability and some form of compositionality-"light" between components in different component models could perhaps be valuable for vehicular applications. An example could be compositional reasoning of interacting software components, where one adheres to the component model used in a control system ECU and the other adheres to the component model used on the on-board computer.

Chapter 5

Conclusions

The goal of this thesis has been to identify the most important characteristics of a component technology for vehicular control systems, to develop such a technology for validation in practice, and to define methods for its successful application to achieve efficient reuse of components. This knowledge is required to maximize the benefits of using component-based principles for this type of applications.

We have proposed a component model, a concept for integration of analysis tools, and other mechanisms, supporting the process of maintaining important quality attributes in the life-cycle of software components. Furthermore, prototypes have been implemented and evaluated, and real cases have been studied, all in cooperation with industry. The main findings are:

- A component model should be limited and restrictive to support important quality attributes, e.g., safety, real-time, and reliability properties. These qualities are most important and can today only be met by systematically favouring simplicity and predictability in all design decisions. We have demonstrated that it is possible to create a usable component model that is predictable enough, even for automated application of state-of-the-art analysis tools from the research community.

- A good choice is to base the component model on the pipes-and-filters interaction style. The style has been proven to give good support for expressing the key functionality of control systems.

- The component frameworks must ensure efficient use of resources. We have proposed an approach where the component-based application built during design-time is efficiently transformed into the execution model of an underlying standard real-time operating system during compile-time. Explicit notion of control-flow (triggering) in the component model allows this transition to efficiently handle static triggering chains (transactions).

- The component frameworks must rely on simple, predictable, static run-time mechanisms. Achieving safety, timelines, and reliability are cross-cutting concerns that must be supported in the component framework. Dynamic constructs such as pointers and dynamic run-time binding are unsuitable for building safe and reliable systems. The possibility to apply accurate analysis of run-time properties during design-time rely heavily on the run-time mechanisms in the framework.

- The repository and associated tools should have support for creating and storing specialized variants of components. In contrast to be based on general components, applications must for high volume products often use components specialized to its specific task. It is also highly recommended to avoid unused code in source files or on target to achieve higher safety integrity levels in certification. The thesis also proposes a method based on storing metadata with traceability information to perform impact analysis of necessary adaptations as support for the adaptation process.

Overall our findings indicate that CBSE principles are suitable for vehicular control systems, which in their fundamental characteristics are similar to control systems in other domains. It is possible to develop component technologies and methods that admit efficient reuse of components, supports implementation of key functionality in the domain, and compositional reasoning of system level quality attributes based on attributes of the components and the execution environment. There is a trade-off between functional capabilities and analysis capabilities, but we have demonstrated how this can be efficiently solved in the context of vehicular control systems.

Bibliography

[1] M. Åkerholm and J. Fredriksson. A sample of component technologies for embedded systems. Technical Report, Mälardalen University, November 2004.

[2] M. Åkerholm, J. Fredriksson, K. Sandström, and I. Crnkovic. Quality attribute support in a component technology for vehicular software. In *Fourth Conference on Software Engineering Research and Practice in Sweden*, Linköping, Sweden, October 2004.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *In Proceedings of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*. LNCS Springer, 2003.

[4] Autosar project. http://www.autosar.org/ (Last Accessed: 2008-04-25).

[5] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical concepts of component-based software engineering, volume ii. Technical report, Software Engineering Institute, Carnegie-Mellon University, May 2000. CMU/SEI-2000-TR-008.

[6] M. Barbacci, M. H. Klein, T. A. Longstaff, and C. B. Weinstock. Quality attributes. Technical report, Software Engineering Institute, Carnegie Mellon University, 1995.

[7] F. Barbier, N. Belloir, and J. Bruel. Incorporation of Test Functionality into Software Components. *Proceedings of the Second International Conference on COTS-Based Software Systems*, 2003.

[8] S. Becker and R. H. Reussner. The impact of software component adaptors on quality of service properties. In *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCATŠ04)*, Oslo, Norway, June 2004.

[9] E. Bondarev, J. Muskens, P. de With, M. Chaudron, and J. Lukkien. Predicting real-time properties of component assemblies: a scenario-simulation approach. In *Proceedings of the 30th Euromicro Conference*, Sep. 2004.

[10] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 5(41), 1999.

[11] D. Box. *Essential COM*. Addison-Wesley, 1998. ISBN: 0-201-63446-5.

[12] J. Brant, B. Foote, R. e. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings of 12th European Confernece on Object-Oriented Programming (ECOOP98)*, July 1998.

[13] M. Broy. Challenges in automotive software engineering. *International Conference on Software Engineering*, pages 33–42, 2006.

[14] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J. Stefani. An Open Component Model and its Support in Java. *Proceedings of the International Symposium on Component-based Software Engineering (CBSE2004), Edinburgh, Scotland*, 2004.

[15] CC Systems. http://www.CC-Systems.com (Last Accessed: 2008-04-25).

[16] J. Conard, P. Dengler, B. Francis, J. Glynn, B. Harvey, B. Hollis, R. Ramachandran, J. Schenken, S. Short, and C. Ullman. *Introducing .NET*. Wrox Press Ltd, 2000. ISBN: 1-861004-89-3.

[17] K. Cooper, J. Zhou, H. Ma, I. L. Yen, and F. Bastani. Code parameterization for satisfaction of qos requirements in embedded software. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003.

[18] B. Cox. Planning the software industrial revolution. *Software, IEEE*, 7(6):25–33, 1990.

[19] I. Crnkovic. Component-based approach for embedded systems. In *9th International Workshop on Component-Oriented Programming*, Oslo, June 2004.

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

[21] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle, pages. *Journal of Computing and Information Technology*, 13(4):321–327, November 2005.

[22] I. Crnkovic and M. Larsson. A case study: Demands on component-based development. In *Proceedings, 22th International Conference of Software Engineering*, Limerick, Ireland, May 2000. ACM, IEEE.

[23] I. Crnkovic and M. Larsson. Challenges of component-based development. *Journal of Software Systems*, December 2001.

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

[25] I. Crnkovic, M. Larsson, and O. Preiss. Concerning predictability in dependable component-based systems: Classification of quality attributes. Springer, LNCS 3549, 2005.

[26] EAST, Embedded Electronic Architecture Project. http://www.east-eea.net/ (Last Accessed: 2008-04-25).

[27] J. Elmqvist and S. Nadjm-Tehrani. Safety-Oriented Design of Component Assemblies using Safety Interfaces. *Electronic Notes in Theoretical Computer Science*, 182:57–72, 2007.

[28] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. *Proceedings of the General Track: 2002 USENIX Annual Technical Conference table of contents*, 2002.

[29] J. Favaro. What Price Reusability?: A Case Study. In *Proceedings 1st ACM First Symposium on Environments and tools for Ada*, 1990.

[30] K. Fernandes, V. Raja, and M. Morley. A System Level Testing Modeling Mechanism in a Reengineering Environment. *Journal of Conceptual Modeling*, 2001.

[31] R. Fichman and C. Kemerer. Object technology and reuse: lessons from early adopters. *IEEE Computer*, 30(10):47–59, 1997.

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

[33] J. Fröberg. Engineering of Vehicle Electronic Systems: Requirements Reflected in Architecture. Technical report, Technology Licentiate Thesis No.26, ISSN 1651-9256, ISBN 91-88834-41-7, Mälardalen Real-Time Reseach Centre, Mälardalen University, March 2004.

[34] J. Fredriksson, T. Nolte, M. Nolin, and H. Schmidt. Contract-based reusable worst-case execution time estimate. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, Daegu, Korea, August 2007.

[35] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: why reuse is so hard. *Software, IEEE*, 12(6):17–26, 1995.

[36] H. Groß, M. Melideo, and F. Barbier. Component+ Methodology. Technical report.

[37] L. Grunske. Towards an Integration of Standard Component-Based Safety Evaluation Techniques with SaveCCM. *Lecture Notes In Computer Science*, 4214:199, 2006.

[38] L. Grunske. Early quality prediction of component-based systems–A generic framework. *The Journal of Systems & Software*, 80(5):678–686, 2007.

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

[40] D. Haggander, L. Lundberg, and J. Matton. Quality attribute conflicts - experiences from a large telecommunication application. In *Proceedings of the 7th IEEE International Conference on Engineering of Complex Computer Systems*, 2001.

[41] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck. The rubus component model for resource constrained real-time systems. In *3rd IEEE International Symposium on Industrial Embedded Systems*, Montpellier, France, June 2008.

[42] G. T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

[43] G. T. Heineman and W. T. Councill. *Component-based Software Engineering, Putting the Pieces Together*. Prentice-Hall, 2001. ISBN: 0-201-70485-4.

[44] S. Hissam, J. Ivers, D. Plakosh, and K. Wallnau. Pin Component Technology (V1. 0) and Its C Interface. 2005.

[45] International Electrotechnical Commission IEC. Standard: IEC61508, Functional Safety of Electrical/Electronic Programmable Safety Related Systems. Technical report.

[46] International Electrotechnical Commission IEC. *International Standard IEC 61131, Programmable controllers*, 1992.

[47] International Electrotechnical Commission IEC. *International Standard IEC 61499, Function blocks, Part 1: Architecture*, 2005.

[48] A. Kleppe, W. Bast, and J. Warmer. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[49] I. Kruger, D. Gupta, R. Mathew, P. Moorthy, W. Phillips, S. Rittmann, and J. Ahluwalia. Towards a process and tool-chain for service-oriented automotive software engineering. *IEE Seminar Digests*, 33(2004), 2004.

[50] M. Larsson. *Predicting Quality Attributes in Component-based Software Systems*. PhD thesis, Mälardalen University, March 2004.

[51] K.-K. Lau, L. Ling, and Z. Wang. Composing components in design phase using exogenous connectors. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2006.

[52] K.-K. Lau and V. Ukis. Component metadata in component-based software development: A survey. Preprint CSPP-34, School of Computer Science, The University of Manchester, October 2005.

[53] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. on Software Engineering*, 33(10):709–724, October 2007.

[54] W. C. Lim. Effects of reuse on quality, productivity, and economics. *The Journal of Software Engineering*, 11(5):23–30, September 1994.

[55] F. Lüders. Use of component-based software architectures in industrial control systems. Technical report, Licentiate Thesis, Mälardalen University Press, December 2003.

[56] F. Lüders, I. Crnkovic, and P. Runeson. Adopting a component-based software architecture for an industrial control system - a case study. pages 232–248. Springer, LNCS 3778, 2005.

[57] K.-L. Lundbäck, J. Lundbäck, and M. Lindberg. Development of dependable real-time applications. Arcticus Systems, Dec. 2004.

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

[59] S. Microsystems. Java 2 platform, enterprise edition (j2ee). URL: http://java.sun.com/j2ee/index.jsp (Last Accessed: 2005-01-17).

[60] H. Mili, A. Mili, S. Yacoub, and E. Addy. *Reuse-based software engineering: techniques, organization, and controls*. Wiley-Interscience New York, NY, USA, 2001.

[61] A. Möller, M. Åkerholm, J. Fredriksson, and M. Nolin. Evaluation of component technologies with respect to industrial requirements. In *Euromicro Conference, Component-Based Software Engineering Track*, Rennes, France, August 2004.

[62] A. Möller, M. Åkerholm, J. Fröberg, and M. Nolin. Industrial grading of quality requirements for automotive software component technologies. In *Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th IEEE International Real-Time Systems Symposium*, December 2005.

[63] R. Monson-Haefel. *Enterprise JavaBeans, Third Edition*. O'Reilly & Assiciates, Inc., 2001. ISBN: 0-596-00226-2.

[64] M. Mrva. Reuse factors in embedded systems design. *IEEE Computer*, 30(8):93–95, 1997.

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

[66] M. Nolin, J. Fredriksson, J. Hammarberg, J. Huselius, J. Håkansson, A. Karlsson, O. Larses, , G. Mustapic, A. Möller, T. Nolte, J. Norberg, D. Nyström, A. Tesanovic, and M. Åkerholm. Component based software engineering for embedded systems - a literature survey. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-102/2003-1-SE, Mälardalen University, June 2003.

[67] D. Nyström, A. Tesanovic, C. Norström, J. Hansson, and N.-E. Bånkestad. Data management issues in vehicle control systems: a case study. In *Euromicro Real-Time Conference 2002*, June 2002.

[68] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontents to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance*, November 2001.

[69] PACC Project, Predictable Assembly from Certified Components. http://www.sei.cmu.edu/pacc (Last Accessed: 2005-04-15).

[70] PROGRESS Project. http://www.mrtc.mdh.se/progress/ (Last Accessed: 2008-04-25).

[71] R. Reussner. Automatic component protocol adaptation with the CoConut/J tool suite. *Future Generation Computer Systems*, 19(5):627–639, 2003.

[72] R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3):241–252, 2003.

[73] Robocop project. www.extra.research.philips.com/euprojects/robocop/ (Last Accessed: 2008-04-25).

[74] S. Rosenbaum and B. du Castel. Managing software reuse - an experience report. *Proceedings of the 17th international conference on Software engineering*, pages 105–111, 1995.

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

[76] Save-IT Project. http://www.mrtc.mdh.se/projects/save-it/ (Last Accessed: 2008-04-25).

[77] Save Project. http://www.mrtc.mdh.se/SAVE/ (Last Accessed: 2008-04-25).

[78] D. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1), 1999.

[79] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[80] M. Shaw. Thruth vs knowledge: The difference between what a component does and what we know it does. In *Proceedings 8th International workshop on software specification and design*, 1996.

[81] M. Shaw. The coming age of software architecture resreach. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001.

[82] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Dicipline*. ISBN 0-13-182957-2. Prentice-Hall, 1996.

[83] Space4u project. www.extra.research.philips.com/euprojects/space4u/ (Last Accessed: 2008-04-25).

[84] C. Szyperski. *Component Software - Beyond Object-Oriented Programming, Second Edition*. Pearson Education Limited, 2002. ISBN: 0-201-74572-0.

[85] W. Tracz. The three cons of software reuse. *Proceedings of the Third Annual Workshop: Methods and Tools for Reuse*, 1990.

[86] R. van Ommering, F. van der Linden, K. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, march 2000.

[87] J. Voas. The Challenges of Using COTS Software in Component-Based Development. *Contact*, 31:44–45, 1998.

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

[89] Y. Wang, G. King, D. Patel, S. Patel, and A. Dorling. On coping with real-time software dynamic inconsistency by built-in tests. *Annals of Software Engineering*, 7(1):283–296, 1999.

[90] D. M. Yellin and R. E. Strom. Protocol specification and component adaptors. *ACM Trans. on Programming Languages and Systems*, 2(19):292–333, March 1997.

[91] J. Zhou, K. Cooper, H. Ma, and l Ling Yen. On the customization of components: A rule-based approach. *IEEE Transactions on Knowledge and Data Engineering*, 19(9), Sept. 2007.

[92] S. Zschaler. Formal specification of non-functional properties of component-based software. In *Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference 2004*, Sept. 2004.

II

Included Papers

Chapter 6

Paper A: SaveCCM a Component Model for Safety-Critical Real-Time Systems

Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren
In Euromicro Conference, Special Session Component Models for Dependable Systems, Rennes, France, September 2004

Abstract

Component-based development has proven effective in many engineering domains, and several general component technologies are available. Most of these are focused on providing an efficient software-engineering process. However, for the majority of embedded systems, run-time efficiency and prediction of system behaviour are as important as process efficiency. This calls for specialized technologies. There is even a need for further specialized technologies adapted to different types of embedded systems, due to the heterogeneity of the domain and the close relation between the software and the often very application specific system.

This paper presents the SaveCCM component model, intended for embedded control applications in vehicular systems. SaveCCM is a simple model in which flexibility is limited to facilitate analysis of real-time and dependability. We present and motivate the model, and provide examples of its use.

6.1 Introduction

Component-based development (CBD) is of great interest to the software engineering community and has achieved considerable success in many engineering domains. Some of the main advantages of CBD are reusability, higher abstraction level and separation of the system development process from the component development process. CBD has been extensively used for several years in desktop environments, office applications, e-business and in Internet- and web-based distributed applications. The component technologies used in these domains originates from object-oriented (OO) techniques. The basic principles of the OO approach, such as encapsulation and class specification, have been further extended; the importance of component interfaces has increased: a component interface is treated as a component specification and the component implementation is treated as a black box. A component interface is also the means of integrating the components in an assembly. Component technologies include the support of component deployment into a system through the component interface. On the other hand, the management of components' quality attributes has not been supported by these technologies. In the domains in which these technologies are widely used, the quality attributes have not been of primary interest and have not been explicitly addressed; they have instead been treated separately from the applied component-based technologies. In many other domains, for example embedded systems, CBD is utilized to a lesser degree for a number of different reasons, although the approach is as attractive here as in other domains. One reason for the limited use of CBD in the embedded systems domain is the difficulty to transfer existing technologies to this domain, due to the difference in system constraints. Another important reason is the inability of component-based technologies to deal with quality attributes as required in these domains. For embedded systems, a number of quality attributes are at least as important as the provided functionality, and the development efforts related to them are most often greater than the efforts related to the implementation of particular functions. For development of vehicular systems, CBD is an attractive approach, but due to specific requirements of system properties such as real-time, reliability and safety, restricted resource consumption (e.g., memory and CPU), general-purpose component models cannot be used. Instead new component models that keep the main principles of the CBD approach, but fulfil specific requirements of the domain, must be developed.

This paper discusses the component model SaveCCM, a part of SAVE-Comp, a component-based development framework being developed in the project SAVE (Component Based Design of Safety Critical Vehicular Systems). The basic idea of SAVEComp is to by focusing on simplicity and analysability of real-time and dependability quality attributes provide efficient support for designing and implementing embedded control applications for vehicular systems

The paper is organised as follows. Section 2 gives a short overview of different component models used in embedded systems. Section 3 briefly presents the SAVE project, and Section 4 outlines the characteristics of the considered application domain. In Section 5, our component model SaveCCM is presented, including textual and graphical syntax, as well as a few illustrative examples. A larger and more complete example from the vehicular domain is provided in Section 6, and in Section 7 we summarize and give an outline of future work.

6.2 Related Work

In addition to widely used component technologies, new component models appear in different application domains, both in industry and academia. We will refer to some of them: Koala and Rubus used in industry and the research models PECT, PECOS and ROBOCOP.

The Koala component technology [1] is designed and used by Philips for development of software in consumer electronics. Koala has passive components that interact through a pipes-and-filters model, which is allocated to active threads. However, Koala does not support analysis of run-time properties.

The Robocop component model [2] is a variant of the Koala component model. A Robocop component is a set of models, each of which provides a particular type of information about the component. An example of such a model is the non-functional model that includes modeling timeliness, reliability, memory use, etc. Robocop aims to cover all aspects of a component-based development process for embedded systems.

The Rubus Component Model [3] is developed by Arcticus systems aimed for small embedded systems. It is used by Volvo Construction Equipment. The component technology incorporates tools, e.g. a scheduler and a graphical tool for application design, and it is tailored for resource constrained systems with real-time requirements. In many aspects Rubus Component Model is similar to SaveCCM; actually some of the basic approaches from Rubus are included in SAVEComp. One difference is that SAVEComp is focused on multiple quality attributes and independences of underlying operating system.

PECT (Prediction-enabled Component Technology) from Software Engineering Institute at CMU [4, 5] focuses on quality attributes specification and methods for prediction of quality attributes on system level from attributes of components. The component model enables description of some real-time attributes. Compared with SAVECom, PECT is a more general-purpose component technology and more complex.

PECOS (PErvasive COmponent Systems) [6], developed by ABB Corporate Research Centre and academia, is designed for field devices, i.e. reactive embedded systems that gathers and analyze data via sensors and react by controlling actuators, valves, motors etc. The focus is on non-functional properties such as memory consumption and timeliness, which makes PECOS goals similar to SaveCCM.

These examples show that there are many similar component technologies for development of embedded systems. One could ask if it would not be more efficient to use a single model. Experiences have shown that for many embedded system domains efficiency in run-time resources consumption and prediction of system behaviour are far more important than efficiency in the software development. This calls for specialization, not generalization. Another argument for specialization is the typically very close relation between software and the system in which the software is embedded. Different platforms and different system architectures require different solutions on the infrastructure and inter-operability level, which leads to different requirements for component models. Also the nature of embedded software limits the possibilities of interoperability between different systems. Despite the importance of pervasiveness, dynamic configurations of interoperation between systems, etc. this is still not the main focus of vast majorities of embedded systems.

These are the reasons why different application domains call for different component models, which may follow the same basic principles of component-based software engineering, but may be different in implementations. With that in mind we can strongly motivate a need for a component technology adjusted for vehicular systems.

6.3 The SAVE Project

The long term aim of the SAVE [7] project is to establish an engineering discipline for systematic development of component-based software for safety critical embedded systems. SAVE is addressing the above challenge by developing a general technology for component-based development of safety-critical vehicular systems, including:

- Methodology and process for development of systems with components
- Component specification and composition, providing a component model which includes the basic characteristics of safety-critical components and infrastructure supporting component collaboration.
- Techniques for analysis and verification of functional correctness, real-time behaviour, safety, and reliability.
- Run-time and configuration support, including support for assembling components into systems, run-time monitoring, and evaluation of alternative configurations.

The main objective of SAVE is to develop SAVEComp - a component-based development (CBD) technology for safety-critical embedded real-time systems (RTS). The primary focus is on designing systems with components, based on component and system models. The ambition is to develop a method and infrastructure for CBD for safety-critical embedded RTS, corresponding to existing general component technologies, such as COM and JavaBeans.

6.4 Application Characteristics

As mentioned above, the considered application domain is vehicular systems. Within that domain we are mainly considering the safety-critical sub-systems responsible for controlling the vehicle dynamics, including power-train, steering, braking, etc.

The vehicular industry has a long tradition of building systems from components provided by different suppliers. In the past these components have been purely mechanical, but today many of the components include computers and software. The trend today is, on one hand, towards *intelligent* mechatronics *light weight nodes*, such as actuators including a microprocessor. On the other hand, there are trends towards more integrated and flexible architectures, where software components can be freely allocated to *heavy weight* computer units (Electronic Control Units; ECUs). One reason for this is that the number of ECUs is growing beyond control in a modern car (in the range of 100 in top of the line models). Letting SW from several suppliers, related to different sub-systems, execute on the same ECU has several benefits, including reduced number of ECUs, reduced cabling, reduced number of connection points (essential for system reliability), reduced weight, and reduced per-unit production cost. The downside is an increased risk of interference between the different sub-systems. Minimizing this risk and increasing efficiency and flexibility in the design process is the main motivation for SAVEComp and other efforts currently in progress (e.g. the EAST/EEA initiative [8]).

The safety-critical sub-systems we consider will in the foreseeable future have the following characteristics:

- Statically configured, i.e., the components used and their interconnections will essentially be decided at design or configuration time. Hence, the binding will be static, as opposed to the dynamic binding used in current component technologies.
- It will be essential to satisfy and provide proof of satisfaction of not only the functional behaviour, but also of timing and dependability quality attributes.
- The timing and dependability quality attributes will be strict, in the sense that they will be specified in terms of absolute bounds that must be satisfied.
- There will be additional, less critical, less static components executing on the same ECUs as the critical ones. The focus of SAVE is however not on these.
- The systems will be resource constrained, in the sense that the per-unit cost is a main optimization criterion, i.e., the use of computer and computing resources should be kept at a minimum.
- Due to the *product-line nature* of the industry, reuse of architectures, components and quality assessments should be supported.
- The contractual aspect of system and component models will in many cases be important as a tool for communication and ensuring quality in the integrator/supplier relation.

Looking more in detail at the timing quality attributes, SaveCCM should provide sufficient machinery to express and reason about the following types of timing attributes/requirements:

- End-to-end timing, i.e., it should be possible to determine (or guarantee) that the time from some event (e.g., sampling of a sensor value) to the time of some other event (e.g., providing a new control signal to an actuator) stays within specified bounds.

- Freshness of data, i.e., it should be possible to determine (or guarantee) that a datum has been generated no earlier than a specified bound before it is used by a specific component (e.g., that a sensor value has been sampled no earlier than 35ms before it is used by a specific component).
- Simultaneity, i.e., it should be possible to determine (or guarantee) that a set of data occur sufficiently close together in time (e.g., that the sampling of two sensors occur within 2ms).
- Jitter tolerances, i.e., it should be possible to determine (or guarantee) that the variation in latency between two events stay within specified bounds (e.g., that the variation in the time between subsequent (periodic) samplings of a sensor value stays within 2ms).

6.5 The SAVEComp Component Model

SaveCCM has its roots in previous models and design methods for embedded real-time systems, in particular Basement [9] and its extensions into the Rubus-methodology [10, 3]. SaveCCM, and its predecessors are designed specifically for the vehicular domain, which (in contrast with many of the current component technologies) implies that predictability and analysability are more important than flexibility. Hence, the model should be as restrictive as possible, while still allowing the intended applications to be conveniently designed. It is with this in mind we have designed SaveCCM.

6.5.1 Architectural Elements

SaveCCM consists of the following main elements:

Components which are basic units of encapsulated behaviour, that executes according to the execution model presented below.

Switches which provide facilities to dynamically change the component interconnection structure (at configuration or run-time).

Assemblies which provide means to form aggregate components from sets of interconnected components and switches.

Run-time framework which provides a set of services, such as communication between components. Component execution and control of sensors and actuators.

Both switches and assemblies can be considered to be special types of components. Due to the difference in semantics we will, however, treat them as separate elements. Below, we will elaborate on these elements, their properties, and their attributes.

Functional interface

The functional interface of all architectural elements is defined in terms of a set of associated 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: (1) data-only ports, (2) triggering-only ports, and (3) data and triggering ports.

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 components 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.

Data-only ports are one element buffers that can be read and written. Each write will overwrite the previous value stored. Output and input ports are distinct, in the sense that writing a datum to an output port does not mean that the datum is immediately available at the input port connected to the output port. This is to allow transfer of data between ports over a network or any other mechanism that does not guarantee atomicity of the transfer.

Triggering-only ports are used for controlling the activation of components. A component 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 an *OR-semantics*, in the sense that the input port is triggered if at least one of its connected output ports is activated. Note that the input triggering port is active from the time of activation (triggering) to the start of execution of the component. Activations cannot be cancelled, and activating an active port has no effect.

Data and triggering ports combine data-only and triggering-only ports in the obvious way.

Execution Model

Since predictability and analyzability are of primary concern for the considered application domain, the SaveCCM execution model is rather restrictive.

The basis is a control-flow (pipes and filter) paradigm in which executions are triggered by clocks or external events, and where components have finite, possibly variable, execution time.

On a high level, a component is either waiting to be activated (triggered) or executing. A component change state from waiting to executing when all input triggering ports are active.

In a first phase of its execution a component reads all its inputs. In its second execution phase the component performs all its computations based only on the inputs read and its internal state. In its third execution phase, the component generates outputs, after which it returns to its idle state waiting for a new triggering.

External I/O

Sensors and actuators (I/O) are accessed via enclosing components, in which the sensor/actuator values are part of the component's internal state.

Timing

Time is a first class citizen in SAVEComp. A global time base is assumed (a perfect clock). This perfect clock is accessed via special components, called triggers, which can trigger the activation of other components. To cater for the imperfection of real clocks, a triggering initiated at time t will arrive at the receiving component sometime in the interval $t + / - O$.

Switches

As mentioned above, a switch provides means for conditional transfer of data and/or triggering between components. Switches allow configuration of assemblies. A switch contains a connection specification, which 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 of the switch are used to determine which connection pattern that is in effect.

It should be noted that a pattern does not have to provide connections for all ports, it is sufficient to only connect some input and some output ports.

Switches can be used for pre-run-time static configuration by statically binding fixed values to the data in some of the input ports, and then use partial evaluation to reduce the alternatives defined by the switch.

Switches can also be used for specifying modes and mode-switches, each mode corresponding to a specific static configuration. By changing the port values at run-time, a new configuration can be activated, thereby effectuating a mode-shift.

Assemblies

As mentioned above, component assemblies allow composite behaviours to be defined, and make it possible to form aggregate components from components and switches. In SaveCCM, assemblies are encapsulations of components and switches having an external functional interface, just as SaveCCM-components. Some of the ports of components and switches are associated/ delegated to the external ports of the assembly.

Due to the strict (and restricted) execution semantics of SaveCCM components, an assembly does not satisfy the requirements of a component. Hence, assemblies should be viewed as a mechanism for naming a collection of components and hiding internal structure, rather than a mechanism for component composition.

Quality attributes

Handling of quality attributes, in particular those related to real-time and safety, is one of the main aspects of SaveCCM. A list of quality attributes and (possibly) their values is included in the specification of components and assemblies. In this paper we will only consider timing attributes. We will show how such attributes can be specified and used in analysis.

6.5.2 Specification and Composition Language

We will now outline the textual syntax used to define SaveCCM components and assemblies.

A SaveCCM system is an aggregate of component instances. A component instance is a named instance of a component type. A component type is either a basic component type or a component assembly type. A basic component type is defined as follows:

Components are specified by their interfaces, behaviour and (quality) attributes. Interfaces are port-based and they specify input and output ports. Behaviour identifies variables that express internal states, and actions that describe the component execution. Variables can be initiated by values from the input ports. Attributes describe different properties of the components. An attribute has a type, value and credibility (a measure of confidence of the expressed value). Credibility value, expressed in percentage is discussed in [11]. Ports include data or triggers or both. A simplified BNF specification of a component type is shown below. Actions are abstract specifications of the externally visible behaviour of the component.

```
<component> ::= Component <typeName> {<componentSpec>}
<componentSpec> :: =<Interface>  [<Behaviour>]  [<Attributes> ]
<Interface> ::= Inports: <port>[,<port>]+ ;
                Outports: <port>[,<port>]+ ;
<port> ::= <portName> : <portTypeName>;
<Behaviour> ::= Variables: <variables>+ Actions: <actions>+
<Variables> ::= <type> <name> [ = <value> | = <port_name> ] ;
<actions> ::= { <action-program> }
<Attributes> ::= Attributes <attributeSpec>+ ;
<attributeSpec> ::=  <type> <name> = <value> [:<credibility>]
<portType> ::= Port <Name> {<portSpec>};
<portSpec> ::= Data: <dataType|empty>;
               Trigger: <bolean> ;
```

Switches are specified as special types of components, however without actions and attributes. Depending on the switch state (condition) particular input and output ports are connected or disconnected.

```
<switch> ::= Switch <type> <name>{<swSpec>}
<swSpec> ::= <Interface>  <behaviour>
<Interface> ::= Inports: <port>[,<port>]+ ;
              Outports:  <port>[,<port>]+ ;
<port> ::= <portType> <portName> ;
<behaviour> ::= Switching: <cond>:<in-out-connect> [,<in-out-connect>];
<in-out-connect> ::= <portName> -> <portName> [,<portName> -> < portName>];
```

An assembly includes a set of components and switches that are *wired* together. Similar to components assemblies can be instantiated, which enables reusability on a higher level than the component level. However, the specification does not include a behaviour (variables and activities) part. Quality attributes are part of assemblies. The reason is that there are assembly properties which cannot be derived from the component properties but are applicable and can be measured on the assembly level.

```
<assembly> ::= Assembly  <assemblyType> {<assemblySpec>}
<assemblySpec> ::= <Interface>  <Behaviour>
                   [<Attributes> ]
<Interface> ::= Inports: <port>[,<port>]+ ;
                Outports: <port>[,<port>]+ ;
<port> ::= <portType> <portName> ;
<Behaviour> ::= Components: <componentName> [,<compomemtName >+]
<connections> ::=  Connections <singleConnection> [,<singleConnection>]+
<singleConnection> ::= <portName> -> <componentName.portName>
                       | <componentName.portName> -> <portName>
                       |<componentName.portName> -> <componentName.portName>
<Attributes> ::= Attributes <attributeSpec>+ ;
<attributeSpec> ::= <type> <name> = <value> [:<credibility>];
```

In modelling and building systems we must create instances of these types and associate instances to tasks that execute on target systems. We will, however, in this paper not discuss these issues further, though our examples will contain some instantiations that we hope will be intuitive enough to be understood without further explanations.

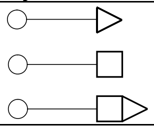
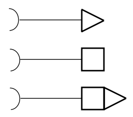
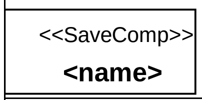
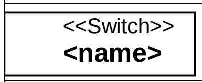
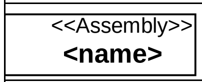
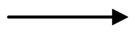
Symbol	Interpretation
	**Input ports** - The upper is an input port with a trigger, and no data. The middle symbol is an input port with data and no triggering, and the lower symbol is an input port with data and triggering.
	**Output port** - Similar to the input ports, the upper is symbol is an output port with triggering functionality but with no data. The middle symbol is an output port with data but with no triggering, and the lower symbols indicates an output port with both data and triggering.
	**Component** - A component with the stereotype changed to SaveComp corresponds to a SaveCCM component.
	**Switch** - components with the stereotype switch, corresponds to switches in SaveCCM.
	**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 6.1: Graphical Syntax of SaveCCM

6.5.3 Graphical Language

A subset of the UML2 component diagrams is adopted as graphical representation language. The interpretation of the symbols for provided and required interfaces, and ports are somewhat modified to fit the needs of SaveComp. The symbols in Figure 6.1 are used.

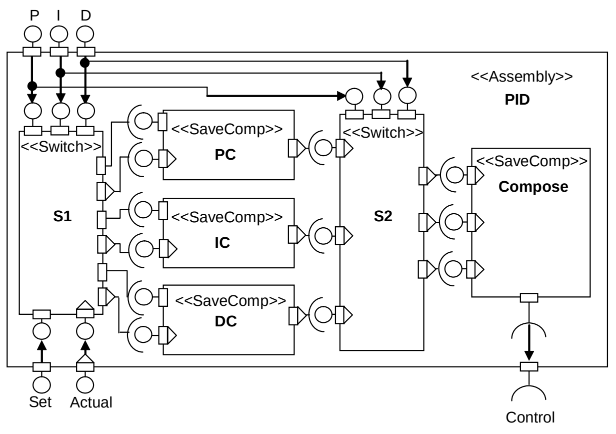


Figure 6.2: Generic PID Controller

6.5.4 Simple Examples

We will give a few examples to illustrate SaveCCM. In the examples we will use our graphical language, and for selected architectural elements also the textual format.

Static Configuration By static configuration we assume instantiation of assemblies and the included components. For example we specify a general controller, which can be configured to be a P, I, D, PI, PD, ID, or PID controller. Switches are used to express this. Graphically we can illustrate PID as in Figure 6.2.

The following is the same example as in Figure 6.2 expressed in the specification and composition language:

```
Assembly PID  {
   Inports: P:Pport, I:Iport, D:Dport,
   Set:Setport, Actual:Actualport;
   Outports: Control:Controlport;
   Components: PC:PCtype, IC:ICtype,
   DC:DCtype, Compose :Ctype, S1:S, S2:Z;
   PortConnect:
      P->{S1.P,S2.P}, I->{S1.I,S2.}, D->
      {S1.D,S2.D},Set->S1.setin, Actual->
      S1.actualin,S1.actualoutp->P.actual,
      S1.actualouti-> I.actual, S1.actualoutd->
      D.actual,S1.setoutp-> P.set, S1.setouti->
      I.set, S1.setoutd->D.set, P.control->S2.p,
      I.control->S2i, D.control-> S2.d, S2.pp->
      Compose.p, S2.ii->Compose.i, S2.dd->
      Compose.d, Compose.control->control
}
Switch S {
   Inports: P:Pport, I:Iport, D:Dport,
   setin:Setport, actualin:Actualport;
   Outports: actualp:Actualport,
   actuali:Actualport, actuald:Actualport,
   setoutp:Setport, setouti:Setport,
   setoutd:Setport
   Switching:
      P: setin->setoutp, actualin->actualp;
      I: setin->setouti, actualin->actuali;
      D: setin->setoutd, actualin->actuald;
}
Switch Z {
   Inports: P:Pport, I:Iport, D:Dport,
   p:Setport, i:Setport, d:Setport;
   Outports: pp:Setport, ii:Setort,
   dd:Setport;
   Switching:
      P: p->pp; I: i->ii; D: d->d;
}
```

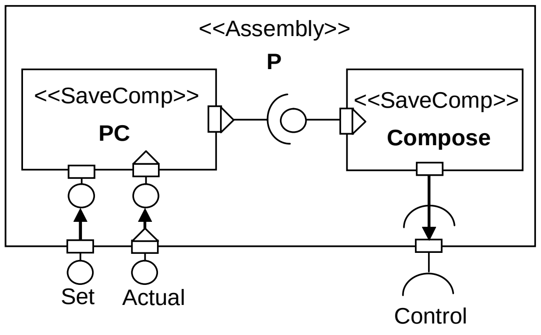


Figure 6.3: Generic PID, statically configured as a P controller

Like components, assemblies can be reused. When creating a component instance or an assembly we can statically bind port values to constants. For instance if the component type PID is instantiated with P set to true, and I and D set to false, we will (by partial evaluation) obtain the following component. This configuration is supposed to be done automatically by a configuration tools.

```
Assembly P:PID (P.val=true, I.val=false, D.val=false) {
   Inports: Set:Setport, Actual:Actualport;
   Outports: Control:Controlport:
   Components: PC:PCtype, Compose:Ctype;
   PortConnect:
      Set->P.set, Actual->P.actual,
      P.control->Compose.p,
      Compose.control->control;
}
```

The graphical interpretation is shown in Figure 6.3.

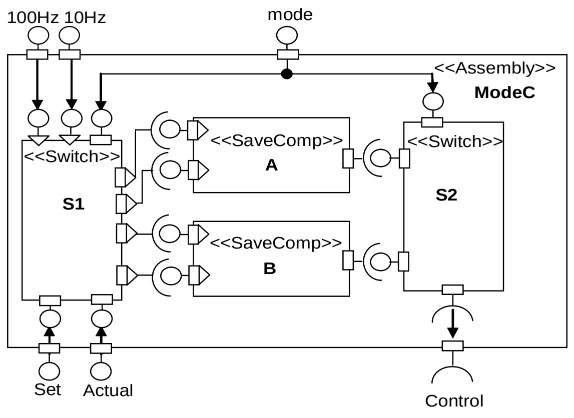


Figure 6.4: Switch effectuating mode-switches, with different execution rates

Mode shift We specify a component (ModeC) with two externally determined modes: idle and busy. In mode idle control algorithm A should run at 10Hz and in mode busy control algorithm B should run at 100Hz. Graphically we illustrate ModeC as in Figure 6.4.

6.6 The Cruise Control Example

To further illustrate the use of SaveCCM we demonstrate a simple design of an Adaptive Cruise Control system (ACC), as an example of an advanced function in a vehicle. An ACC system helps the driver to keep the distance to a vehicle in-front, i.e., it autonomously adapt the velocity of the vehicle to the velocity

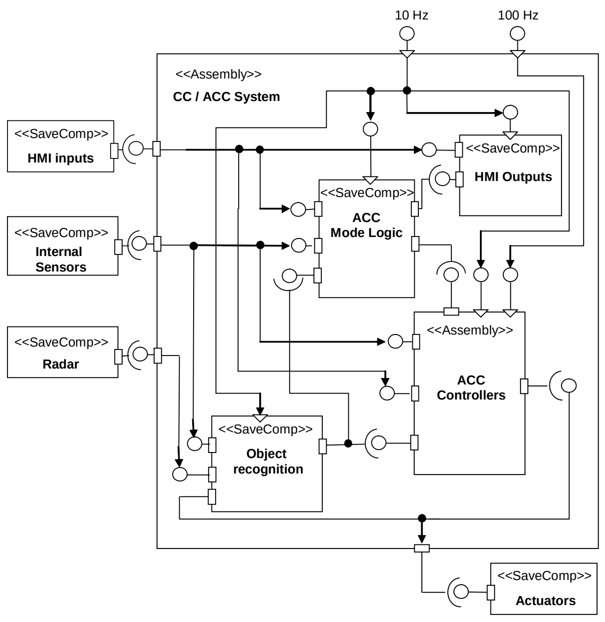


Figure 6.5: ACC system

and distance of the vehicle in front. Figure 6.5 visualises a suggested ACC system using SaveCMM.

The ACC system can be divided into three major parts: input, control, and actuate. Our focus will be on the control part that is encapsulated in the CC/ACC System assembly. The CC/ACC system consists of three components and a switch:

Object recognition is a component that has responsibility to determine if there is a vehicle in front and in that case estimate the distance and relative velocity. It is triggered by the CC/ACC 10 Hz triggering port, and has a Worst Case Execution Time (WCET) of 30 ms.

ACC controllers is an assembly implementing two cascaded controllers. The inner controller is for speed control and can be used for normal Cruise Control (CC), while the outer handles distance control. The assembly has two triggering ports, one for the inner loop, and one for the outer.

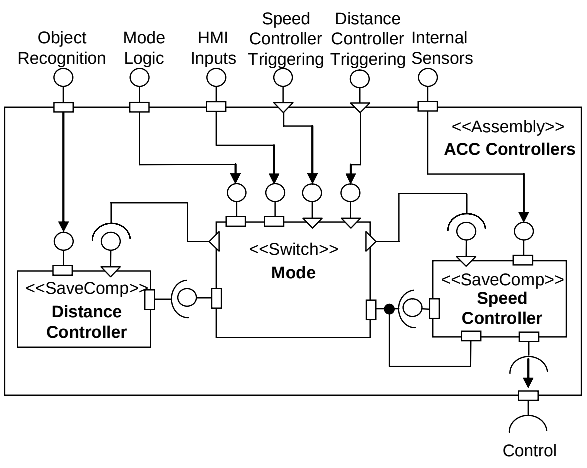


Figure 6.6: ACC controllers assembly

HMI outputs is a component that gives information to the driver through the vehicle computer display, e.g., information about the vehicle state and latest request. The component is triggered by the CC/ACC systems triggering port bound to 10 Hz. The WCET is 2 ms.

ACC mode logic is a component implementing the logic for shifting modes depending on the state of the vehicle, inputs by the driver and from the environment (vehicles in front). The different modes are CC, ACC, and standby. It is triggered by the 10 Hz port. The WCET is less than 1 ms.

A diagram showing the internal design of the assembly ACC Controllers is provided in Figure 6.6.

In Figure 6.6 the name of the component attached to in-ports is written above each port. A brief presentation of the different components in the assembly is given below.

Distance Controller is a pure controller component implementing a control algorithm; it handles distance control and is the component in the outer loop. The WCET is 20 ms, and it is triggered at 10 Hz.

Mode is a switch, which depending on the actual mode of the controller activates and deactivates the both controller components. The switch also switches the input of the speed controller, between HMI Inputs (CC functionality) and from the control signal of the outer loop controller (ACC functionality).

The speed controller executes with a rate five times faster than the rate of the distance controller due to faster dynamics, it control the speed of the vehicle. The WCET is 5 ms.

As illustrated by the example, SaveCCM is designed to seamless and easily support typical requirements that arise when designing advanced vehicular functionality, e.g., connections with data, triggering and both, assemblies, feedback, and mode changes.

As an illustration how the above SaveCCM specification can be used in analysis of timing properties, let us (somewhat simplified) assume that the CC/ACC System will be exclusively allocated to an ECU and that each component is allocated to a single task. We further assume that the tasks are executing under a fixed priority (FPS) real-time kernel, with a zero execution time overhead, and that the deadline attributes of the components are defined to be equal to the periods. Given this, and using deadline monotonic priority assignment, together with the execution time attributes of the components, we can derive the task set in Table 6.1 for the ACC mode.

Task	Period (ms)	WCET (ms)	Prio
Object Recognition	100	30	5
Mode Logic	100	1	4
HMI Outputs	100	2	3
Distance Controller	100	20	2
Speed Controller	20	5	1

Table 6.1: The resulting task set

The task set can be used as input to standard fixed-priority schedulability analysis tools (e.g. [12]). We can use such a tool to verify if the deadline attributes are satisfied. By applying this analysis we find that the all deadline attributes are satisfied, hence we can from now on treat these attributes as properties of the current configuration of the CC/ACC System.

6.7 Conclusions and Further Work

We have presented SaveCCM, a component mode intended for embedded control applications in vehicular systems. In contrast with most current component technologies, SaveCCM is sacrificing flexibility to facilitate analysis; in particular analysis of dependability and real-time. We illustrate SaveCCM by a simple example, where we also, as an example of timing analysis, show that SaveCCM models are amenable to schedulabilty analysis.

This paper covers only parts of the component specifications. In our future work we will provide a complete and formal definition of SaveCCM, as well as linking it to further methods and tools for both dependability and timing analysis. Parts of the specifications not discussed here include actions and attributes describing dynamic behaviour of the components and attribute values that are used for reasoning about system properties.

Bibliography

[1] R. van Ommering, F. van der Linden, and J. Kramer. The koala component model for consumer electronics software. In *IEEE Computer*, pages 78–85, March 2000.

[2] M. de Jonge, J. Muskens, and M. Chaudron. Scenario-based prediction of run-time resource consumption in component-based software systems. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering (CBSE6)*, May 2003.

[3] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N. Bånkestad. Experiences from introducing state-of-the-art real-time techniques in the automotive industry. In *In Eigth IEEE International Conference and Workshop on the Engineering of Compute-Based Systems Washington, US*. IEEE, April 2001.

[4] K. C. Wallnau and J. Ivers. Snapshot of ccl: A language for predictable assembly. Technical report, Software Engineering Institute, Carnegie Mellon University, 2003. CMU/SEI-2003-TN-025.

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

[6] P. Müller, C. Stich, and C. Zeidler. Components @ work: Component technology for embedded systems. In *Proceedings of the 27th International Euromicro Conference*, 2001.

[7] Save Project. http://www.mrtc.mdh.se/SAVE/ (Last Accessed: 2008-04-25).

[8] EAST, Embedded Electronic Architecture Project. http://www.east-eea.net/ (Last Accessed: 2008-04-25).

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

[10] C. Norström D. Isovic. *Building Reliable Component-Based Software Systems*, chapter Components in Real-time systems. Artech House Publishers, July 2002. ISBN 1-58053-327-2.

[11] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Dicipline*. ISBN 0-13-182957-2. Prentice-Hall, 1996.

[12] Mast - modeling and analysis suite for real-time applications. http://mast.unican.es/.

Chapter 7

Paper B: Towards a Dependable Component Technology for Embedded System Applications

Mikael Åkerholm, Anders Möller, Hans Hansson, and Mikael Nolin
In Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS2005), Sedona, Arizona, february, 2005

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 does also show that it needs to be extended to be fully applicable in an industrial context.

7.1 Introduction

Software is central to enable functionality in modern electronic products, but it is also the source of a number of quality problems and constitutes a major part of the development cost. This is further accentuated by the increasing complexity and integration of products. Improving quality and predictability of Embedded Computer Systems (ECS) are prerequisites 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 with a still unproven potential 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 interfaces compared to 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 most modern vehicles and in the robotics and automation industries. Hence, we cooperate with leading product companies (e.g. ABB, Bombardier and Volvo) and some of their suppliers (e.g. CC Systems) in order to establish this novel component technology.

Support for dependability can be added at many different abstraction levels (e.g. the source code and the operating system levels). At each level, different methods and techniques can be used to increase the dependability of the system. Our hypothesis is that dependability, together with other key characteristics, such as resource efficiency and predictability, should be introduced early in the software process and supported through all stages of the process. Our view is that dependability, and similar cross-cutting characteristics, cannot be achieved by addressing only one abstraction level or one phase in the software life-cycle. Further, we argue that dependability of systems is enhanced by systematic application of code synthesis. For code synthesis, models of component behaviour and their resource requirements together with application requirements and models of the underlying hardware and operating system are used. The models and requirements are used by resource and timing analysis algorithms to ensure that a feasible system is generated.

In this paper, we present the current implementation of our component technology (Section 7.3), together with an example application that illustrates its use (Section 7.4). Based on experiences with the example application, we provide an evaluation of the technology (Section 7.5).

7.2 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 intended 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 compile 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 (COM)[1] for desktop applications, and Sun's Enterprise Java Beans (EJB)[2] 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 supporting 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 [1, 2], and from academia [3], [4]. However, as Crnkovic points out in [5], 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 [6].

7.3 Our Component Technology

Our component technology implements the SaveComp Component Model [7] and provides compile-time mappings to a set of operating systems, following the technique described in [8]. The component technology is intended to provide three main benefits for developers of embedded systems: efficient development, predictable behaviour, and run-time efficiency.

[1]Microsoft Corporation, The Component Object Model, http://www.microsoft.com
[2]Sun Microsystems, Enterprise JavaBeans Specification, http://www.sun.com

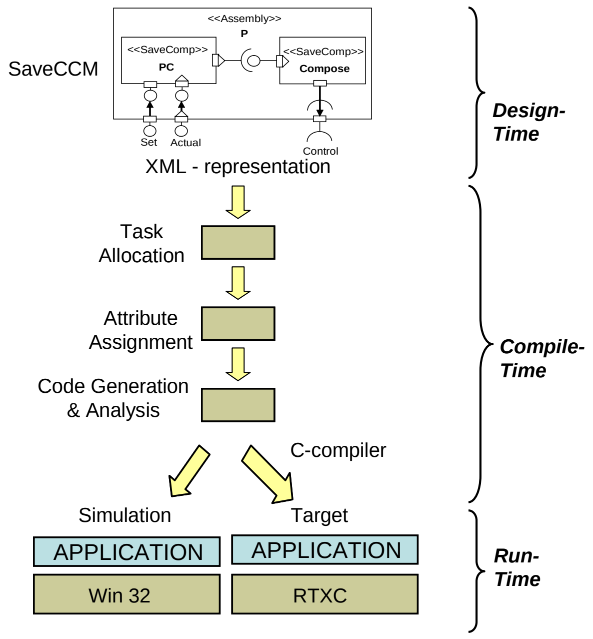


Figure 7.1: An overview of our current component technology

Efficient development is provided by the SaveComp Component ModelŠs efficient mechanisms for developing embedded control systems. This component model is restricted in expressiveness (to support predictability and dependability) but the expressive power has been focused to the needs of embedded control designers.

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 7.1, 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 operating system. Currently our component technology supports the RTXC operating system[3] and the Microsoft Win32 environment[4]. The Win32 environment is intended for functional test and debug activities (using CCSimTech [15]), but it does not support real-time tests.

7.3.1 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.

[3]Quadros Systems Inc, RTXC Kernel User's Guide, http://www.quadros.com
[4]MSDN, Win32 Application Programmer's Interface, http://msdn.microsoft.com/

- 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 a product line.

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 associated 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 [9]. 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

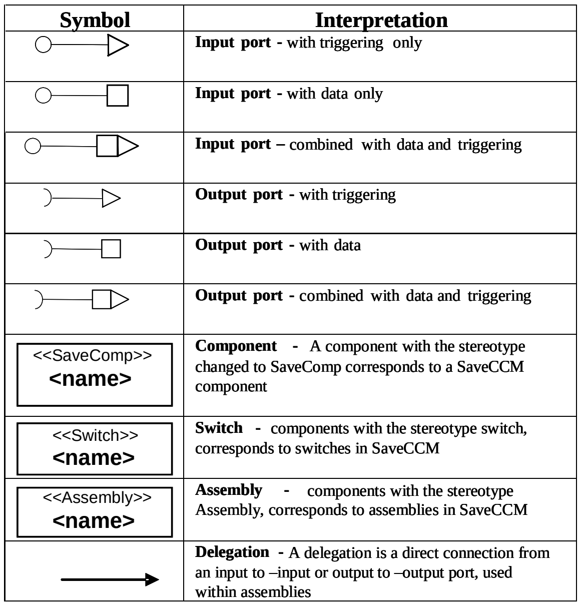
Component assemblies allow composite behaviours to be defined, and make it possible to form aggregate components from groups of components, switches, and assemblies. In SaveCCM, assemblies are encapsulation of components and switches, having an external functional interface (just as SaveCCM-components).

SaveCCM Syntax

The graphical syntax of SaveCCM is shown in 7.2, the syntax is derived from symbols in UML 2.0[5], with additions to distinguish between the different types of ports. The textual syntax is XML[6] based, and the syntax definition is available in [6].

[5]Object Management Group, UML 2.0 Superstructure Specification, http://www.omg.com/-uml/

[6]World Wide Web Consortium (W3C), XML, http://www.w3.org/XML/



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 7.2: Graphical syntax of SaveCCM

7.3.2 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 during the compile-step. In the compiler, the modules (see Figure 7.1) 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.

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 [6].

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 (T) during code generation for specifying the period time for tasks.

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

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

Deadline (D) 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.

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 [10], memory consumption analysis [11], and reliability analysis [12].

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 [13]. 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 [14].

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 7.3 right) and the Win32 operating system (see Figure 7.3 left), 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. The APIs are later translated using C-style defines to the desired target operating system.

7.3.3 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 interface towards the application, but does only contain the run-time components needed (e.g. the SaveCCM API does not include a CAN interface, 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 using CCSimTech [15], 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.

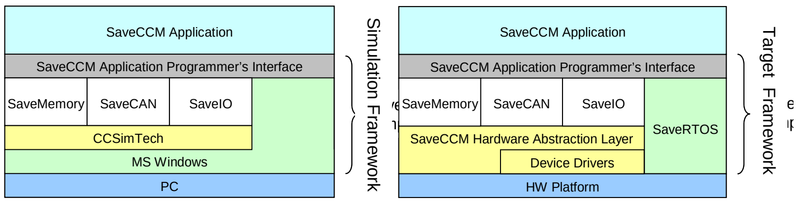


Figure 7.3: System architecture for simulation and target

The layered component run-time framework is designed to enhance portability, which is a strong industrial requirement [16].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 todayŠs CPUs also make portability very important.

7.4 Application Example

To evaluate SaveCCM and the compile-time and run-time parts of the component technology, a typical vehicular application was implemented. The application used for evaluation is an Adaptive Cruise Controller (ACC) for a vehicle. 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.

7.4.1 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 to 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. The gap between two vehicles has to be large enough to avoid 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. One extension is the possibility for autonomous changes of the maximum speed of the vehicle depending on the speed-limit regulations. This feature 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). The second extension is 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.

7.4.2 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) (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 (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 left part of Figure 7.4) 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 7.4, right.

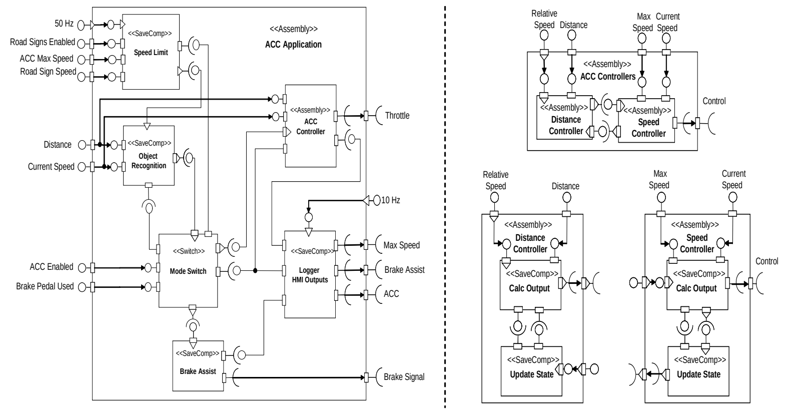


Figure 7.4: ACC Application implementation

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 trig 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 7.4, right), managing the throttle lever of the vehicle. This assembly has two sub-level assemblies, the *Distance Controller* assembly and the *Speed Controller* assembly.

The reason for using a control feedback solution between the two controllers is that since the calculation is very time critical, it is important to deliver the response (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 [6].

7.4.3 Application Test-Bed Environment

For the evaluation the RTXC operating system was used together with a Cross FIRE ECU[7]. 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[8] IO-distributing ECU (Electronic Control Unit) designed for CAN-based real-time systems. The ECU is developed and produced by CC Systems, and intended for use on mobile applications in rough environments.

During functional testing and debugging, CC Systems use a simulation environment called CCSimTech [15], 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.

7.5 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 rep-

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

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

resentatives 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.

We divide our evaluation in the following three categories:

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 ordered as a pilot study at the company. The hardware, operating system, compilers, and the simulation technique, have been selected among the companies 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 related evaluation that can be addressed by our experience. However, to evaluate the process related issues, senior process managers at the company have helped to relate the component technology to the processes.

The evaluation is conducted using a check-list assembled from requirements for automotive component technologies by Möller et al. [16], risks with using CBSE for embedded systems by Larn and Vickers [17], and from identified needs, by Crnkovic [5].

7.5.1 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 provided 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. Further allowing the developer to concentrate on application functionality.

Since the components have visible source code, and since all bindings between components are automatically generated, making modifications of components is facilitated, though there is not yet any specific support to handle maintenance implemented in the component technology.

It is straight forward to compile the ACC system 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) [18]. 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.

7.5.2 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, perecents 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, together with 8 kb run-time framework and compiler generated operating system dependent data and 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 straightforward, 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 provided very useful for verification and debugging of the application functionality.

7.5.3 Process Related

The process related evaluation concerns the suitability to use the existing processes and organisation, when developing component-based applications. So 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 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.

- The maintainability aspects of CBD are attractive, since changes are simplified by the tight relation between the applications description and the source code.

7.6 Conclusions and Future Work

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

The evaluation shows that the existing parts of the component technology meet the requirements and needs related to them. However, to meet overall requirements and needs, 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 [19], run-time monitoring support [20], and a real-time database for structured handling of shared data [21]. Implementation of more types of automated analysis to prove the concept of determining system attributes from component attributes is also a target for future work. However, there is also a need for methods to determine component attributes. Furthermore, to make the prototype useful in practice, there are needs for integrating our technology with supporting tools, e.g., automatic generation of XML descriptions from UML 2.0 drawings, and connectivity with configuration management tools.

An 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 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 Jörgen Hansson and Ken Lindfors for invitation and to Johan Strandberg and Fredrik Löwenhielm for their support with all kinds of technical issues. We would also like to thank Sasikumar Punnekkat for valuable feedback on early versions of this article.

Bibliography

[1] K.L. Lundbäck and J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. Arcticus Systems: http://www.arcticus.se (Last Accessed: 2005-01-18).

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

[3] 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 6th International Workshop on Component-Based Software Engineering*, May 2003.

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

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

[6] M. Åkerholm, A. Möller, H. Hansson, and M. Nolin. SaveComp - a Dependable Component Technology for Embedded Systems Software. Technical report, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-165/2004-1-SE, MRTC, Mälardalen University, December 2004.

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

[8] K. Sandström, J. Fredriksson, and M. Åkerholm. Introducing a Component Technology for Safety Critical Embedded Real-Time Systems.

In *Proceedings of th 7th International Symposium on Component-Based Software Engineering (CBSE7)*, May 2004.

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

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

[11] A.V. Fioukov, E.M. Eskenazi, D.K. Hammer, and M. Chaudron. Evaluation of Static Properties for Component-Based Architetures. In *Proceedings of 28th Euromicro Conference*, September 2002.

[12] H.W. Schmidt and R.H. Reussner. Parameterized Contracts and Adapter Synthesis. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, May 2001.

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

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

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

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

[17] W. Lam and A.J. Vickers. Managing the Risks of Component-Based Software Engineering. In *Proceedings of the 5th International Symposium on Assessment of Software Tools*, June 1997.

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

[19] 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.

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

[21] 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.

Chapter 8

Paper C: The SAVE approach to component-based development of vehicular systems

Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli

In Journal of Systems and Software, vol 80, nr 5, Elsevier, May, 2007

Abstract

The component-based strategy aims to manage complexity, reduce time-to-market, and decrease maintenance efforts by building systems from existing components. For embedded software, the full potential of this strategy has not yet been demonstrated, mainly due to specific requirements in the domain, e.g., 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.

8.1 Introduction

Component-Based Software Engineering (CBSE) is a young software engineering approach which has already shown successful in many software development projects. However, it has mainly been used 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 development of embedded vehicular control-system software. The underlying assumption is that one 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 process and treated through all stages in the process, since they are cross-cutting concerns that can not be achieved by addressing only one phase in the software life-cycle. To support domain requirements, the proposed component technology enables easy usage of analysis and verification methods during the whole software development process, through automated connectivity to test and analysis tools. Constructing formal models manually for analysis tools can be a time-consuming and demanding task, which often cannot be afforded as a repeated activity. Our work has been guided by continuous evaluation of its suitability in an running industrial case-study.

The research presented in this article has been carried out within the SAVE project[1], which has as long-term goal to establish an engineering discipline for systematic development of component-based software for safety-critical embedded systems. The focus of SAVE is on a single application area (vehicular systems), with the aim that results should be applicable to a wider area. The component technology presented here is one of the core parts of the project.

In the software engineering field, reuse has not been as successful as in other fields, 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 [1]. 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 [2]. One of the most central concepts in CBSE theory is *component technologies*. A component technology provides support for the composition of component-based software. It often contains various development tools for simplifying the engineering process,

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

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 elements to be classified as components, and certain patterns for assembling components. Component technologies for embedded systems should support general embedded domain characteristics, e.g., as described by [3]: 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 to verify 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 where SaveCCT has been used. Finally, Section 6 concludes the article.

8.1.1 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, e.g., engine, brakes, and steering. Other classes of electronic systems in modern vehicles include cabin systems, and infotainment systems [4].

The physical architecture of the electronic system in vehicles is a complex distributed computer system. The computer nodes are called Electronic Control Units (ECUs), and are often developed by different vendors and use different hardware. As an example, Figure 8.1 [**?**, from]]FrobergLicThesis:Froberg2004 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 for its ability to simplify reuse. Besides obvious advantages with reuse, CBSE is seen as a method that increases maintainability of software. Component based software is by definition modularised and changes can be isolated to a limited set of components. Thus, maintenance efforts can be decreased compared to when using monolithic software. Another important benefit compared to other

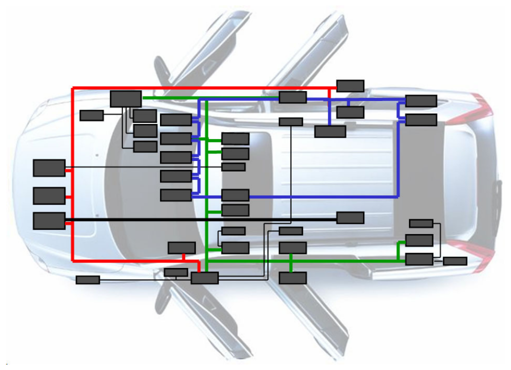


Figure 8.1: Overview of the electronic system architecture in Volvo XC90.

approaches is the support for product-line architectures, which is common for essentially all high volume products (e.g., vehicles). Product-line architectures are used to rapidly and costs effectively provide new products from a product family. The software is organised in a base-line variant, and new products are obtained by additions and/or replacements of components to the baseline. However, vehicular software has certain demands that must be considered when choosing development techniques and technologies, including:

- *Analysis* – Developers of vehicular software must at early stages in development perform analysis 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 extensive testing, complemented by formal methods. A component technology can improve testability by e.g., using well-defined and understandable run-time mechanisms, support for simulation to increase observability, and possibilities for mixed hardware-software tests.

- *Resource efficiency* – The software must use resources efficiently, since vehicles are produced in large volumes. This means that the software platform, development technologies, and system architecture must be chosen to serve the particular needs of embedded vehicular systems using resources efficiently.

8.1.2 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 [5], which originates from our previous work with Basement [6]. 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. Rubus main limitations are that the static scheduling approach only supports periodic activation and that timing aspects are the only extra-functional properties considered.

From Koala [7], SaveCCT has adopted the idea of 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 main actors. These projects focus on areas like analysis, fault prevention, power management, and terminal management; but compared to SaveCCT they are geared towards less safety-critical applications, such as consumer electronics.

An ongoing project with similarities in goals, but which compared to SaveCCT has taken a different approach, is Predictable Assembly from Certifiable Components (PACC)[4] at the Software Engineering Institute. The project focuses on how a component technology can be used and adopted to achieve predictable assemblies. Their concept of Prediction Enabled Component Technologies (PECT) [8] describes a concept for integration of component technologies and analysis techniques. Rather than being a concrete technology (as SaveCCT), PECT describes how to restrict 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 [9] is one of the component technologies targeting the automation industry. It emerged from a joint ABB and academia project focusing on devel-

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

oping a component technology especially 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 [10] defines a graphical language that can be used for composition of components. The language uses the same pipes-and-filters interaction model between components as SaveCCT, but analysis of extra-functional properties is not prioritised 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 [11], develops and implements a model for prediction and consistency checking of extra-functional properties relevant for distributed real-time control-systems. The main focus of the project is 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 running contemporary to our project, but public results in real context are still missing.

8.2 The SaveComp Component Technology

The SaveComp Component Technology (SaveCCT) is here described by distinguishing *manual design*, *automated activities*, and *execution*, which are discussed in the following sub-sections. Referring to Figure 10.1 which provides an overview of SaveCCT, the entry point for a developer is the design tool, where the application is created. During development a developer can utilise a number of available analysis tools with automated connectivity to the design tool. Analysis should be complemented by testing, which is possible already at early stages in the project through replacing hardware, run-time platforms, and missing parts of the system with simulated equivalencies. 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 differ.

8.2.1 Manual Design

During manual design, developers use a component-based strategy, supported by a set of tools for design and analysis. Practising CBSE means that developers distinguish *component development* from *system development*. Component

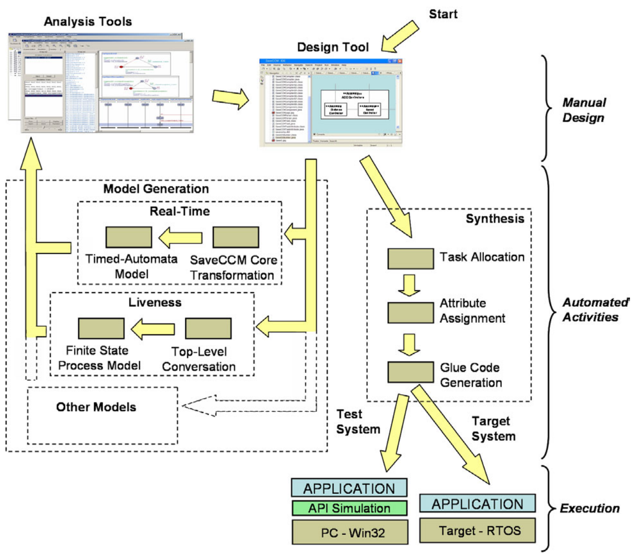


Figure 8.2: Overview of the SaveComp Component Technology.

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. Component development and system development are independent activities that can, e.g., be parallel or performed by different companies.

The development starts with identification of component requirements. This shall be done with respect to already available components; the remaining components have to be developed in parallel or bought from third parties. Then, the SaveCCT design tool provides support for graphical assembly of applications from existing components (i.e., system development). The tool allows designers to specify the component interconnection logics, and express high level constraints on the resulting application. Assembling components is done with respect to the rules of the SaveComp Component Model (SaveCCM) (see Section 3), which is 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. Some specific examples of key functionality are feedback control, system mode changes, and static configuration for variability within product-line architectures.

As shown in Figure 10.1, SaveCCT incorporates a number of analysis tools, which can be used for verifying specific attributes of the application, e.g., related to timeliness and safety. To efficiently incorporate an analysis tool, as much as possible of the translation from the model created with the design tool to the model required by the desired analysis tool should be automated. To date we have incorporated LTSA [12], and TIMES [13], described further in Section 8.4.

8.2.2 Automated Activites

Automated activities produce necessary code for the run-time system (i.e., glue-code), and different specialized 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), meaning that components are free from dependencies to the underlying platform. Furthermore, the code generation step statically resolves resource usage and timing, with the strategy to resolve as much as possible during compile-time instead of depending on costly run-time algorithms. Synthesis consists of four steps (task allocation, attribute assignment, analysis, and code generation), described in more detail by [14].

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.

8.2.3 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 underlying platform. The current implementation use 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 rough environments. Tasking[7] is integrated as the target compiler for the CrossFire MX.

To facilitate testing and debugging we incorporate CCSimTech [15], which is a simulation framework that offers simulated software equivalences as replacements for much common hardware in embedded systems, e.g., IO (digital and analogue), network technologies, and memories. This enables test and debug of distributed embedded control systems in a PC environment without access to target hardware. It enables easy unit-testing for the developers in their standard PC as well as test automation possibilities, and the tests can start even before the intended target hardware is available. Furthermore, CCSimTech provides support for mixed hardware-software tests, where some of the nodes in the distributed system is 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 utilsed. However, certain verification must be performed on the target hardware, e.g., timing related and acceptance tests in the intended environment.

8.3 The SaveComp Component Model

The SaveComp Component Model (SaveCCM) formalises the SaveCCT component concept, and defines how components can be combined to create systems [16]. To suit the domain of vehicular systems, the component model should support the development of resource-efficient systems, and thus the runtime 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

[5]www.quadros.com

[6]www.cc-systems.com

[7]www.tasking.com

timed automata with tasks. In this article, we use the graphical notation only, and present the semantics informally. The reader is referred to [17] for details on the formal semantics. The graphical notation is presented in Figure 8.3.

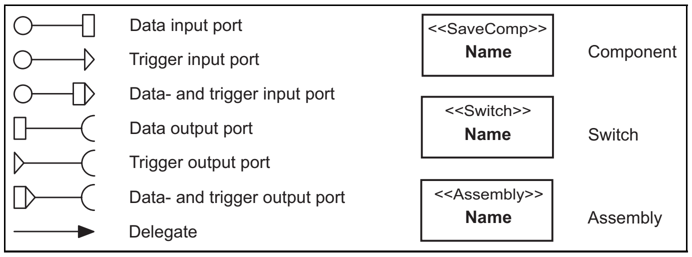


Figure 8.3: The graphical notation of SaveCCM.

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., which is crucial to ensure correctness of real-time systems.

8.3.1 Components

Components are the main architectural element in SaveCCM. In addition to input and output ports, the interface of a component contains a list 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 more complex components that consist of a number of possibly communicating tasks. In both cases, no intercomponent dependencies are allowed, except those explicitly captured by the ports.

A component is initially inactive. It remains in this state until all input triggering 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. Then, it performs the associated computations based only on this input 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 triggering 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.

8.3.2 Switches

The switch construct in SaveCCM is similar to that in Koala [7]. Switches provide 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 is the case of components. Instead, they respond directly to the arrival of data or a trigger signal to 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.

8.3.3 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, assemblies should be viewed as a mechanism for naming a collection of components and hiding internal structure, rather than a component composition mechanism. The SaveCCM semantics [17] also defines an encapsulation construct that do 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.

8.3.4 Ports and Connections

As mentioned above, we distinguish between input and output ports, and between trigger ports and typed data 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 where data never remains.

Connections come in two flavours: 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 residing 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 Figure 8.4 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.

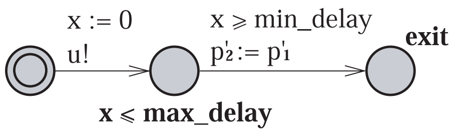


Figure 8.4: The behaviour of a complex connection with a non-deterministic delay in the interval [*min_delay*, *max_delay*].

Following 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.

8.4 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 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, UPPAAL and 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 analyses techniques have been investigated within the SAVE project. [18] 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 would permit tighter analysis in some cases, e.g., for components that behave differently in different operational modes. [19] define a safety analysis framework where 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 [20].

8.4.1 LTSA

LTSA (Labelled Transition System Analyser) is a verification tool for concurrent systems [12]. The tool is based on a process algebra notation (FSP) in which the system model is specified together with specifications of its intended behaviour. 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* to ensure that a specified action will always be performed at some point in the future, regardless of 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 regarding the component interaction within a system. The tool was originally incorporated in SaveCCT for analysis of control loops [21], but it can also be used to analyse general systems. The analysis is based on a 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 easily be 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.

8.4.2 The TIMES tool

The modelling language *timed automata* [22] is useful for modelling and analysis of real-time systems. A timed automaton is essentially a finite state automaton extended with real-valued clocks that can be tested and reset. The formalism has shown to be suitable for a wide range of real-time systems, model-checking tools such as UPPAAL [23] and Kronos [24] have been used to analyse many industrial size systems [25, 26, 27, 28].

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, called *timed automata with tasks* [29], 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 [13] that is 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 [17], 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 8.5.3 where we show how the transformation tool is applied to a concrete example system.

8.5 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 [14] 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 tools for analyses.

An Adaptive Cruise Controller (ACC) is an extension of a regular Cruise Controller. In addition to the conventional task of maintaining a constant velocity, an ACC also provides functionality to help the driver keeping the distance to a preceding vehicle, by autonomously adapting the velocity of the vehicle to the velocity and distance of the vehicle in front.

To increase the complexity of a basic ACC system, and thereby exercise the component model further, our ACC system has two non-standard functional extensions. One extension is the possibility for autonomous changes of the maximum speed of the vehicle depending on the speed-limit regulations. This feature would require that the ACC system have access to the actual speed-limit regulations, e.g., provided by transmitters on the road signs or road map information in cooperation with 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 reminder of this section, we describe the development of an ACC application using SaveCCT. The design is presented, 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.

8.5.1 System Design

The sources of input to the ACC application can be divided in 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.

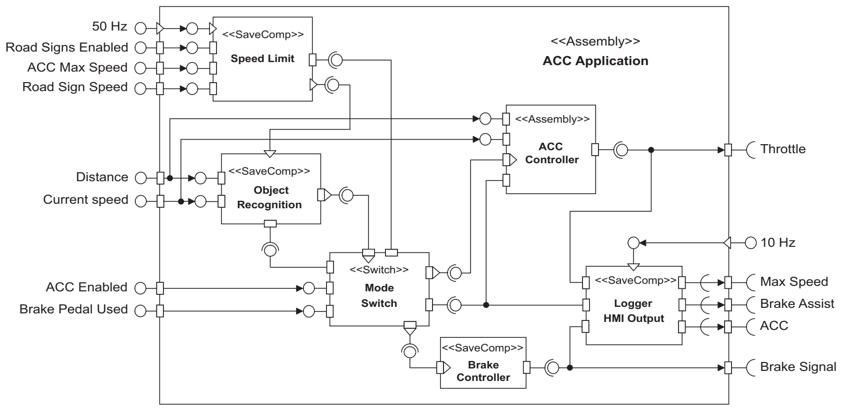


Figure 8.5: ACC application design.

The ACC system is designed as a SaveCCM assembly (*ACC Application* in Figure 8.5) built from four basic components, one switch, and one sub-assembly. The design of the sub-assembly (*ACC Controller*) is in turn shown in Figure 8.6. 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, in case there is, to calculate its speed relative to the vehicle. Based 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 settings of the ACC.
- The *ACC Controller* assembly manages the throttle lever of the vehicle, based on 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. Thus, the other components can be reused throughout the product-line, also 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.

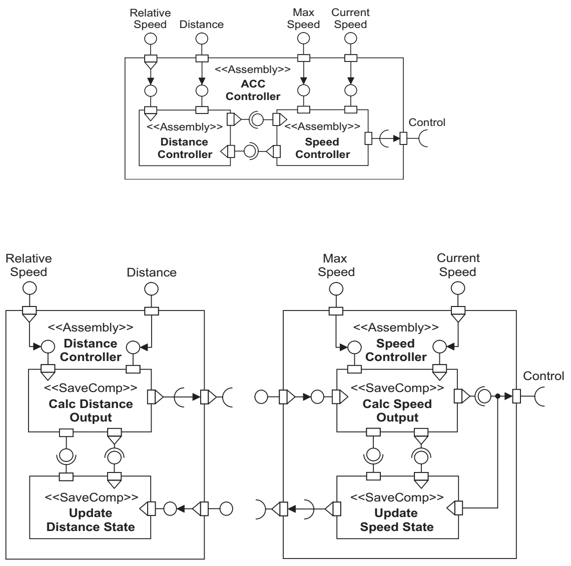


Figure 8.6: Internal design of *ACC Controller*.

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 lever level) as fast as possible is crucial. The assembly is built from two cascaded controllers (see Figure 8.6), represented by the sub-level assemblies *Distance Controller* and *Speed Controller*.

This design corresponds to the *control module* concept introduced by [30]. 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 depending on 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 like in *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.

8.5.2 LTSA Analysis

The LTSA tool described in Section 8.4.1 can be used to check a number of implicit properties, such as deadlock- and livelock-freeness. 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 exemplify 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 on 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 considered subsystem. In this case, the environment simply provides the data expected on 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 suffices to extract the FSP specification of *ACC Controller*, since this is a local property, independent from the interaction with the other components. We specify as valid behaviours of the system all those in which the *Update State* component will always read data from an input port only after that *Calculate Output* has written to its output port.

8.5.3 Analysis using the TIMES tool

As described in Section 8.4.2, 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 generated model of the ACC, we have produced an abstract model of the environment that non-deterministically stimulates 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 variables can be computed 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 preserved 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. The properties should normally be satisfied in any SaveCCM model. We have checked three kinds of properties of the ACC model and its environment. The first two were automatically generated, whereas the third was manually specified:

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 property must hold to ensure that no

[8] For more information about the UPPAAL tool, see the web site `www.uppaal.com`.

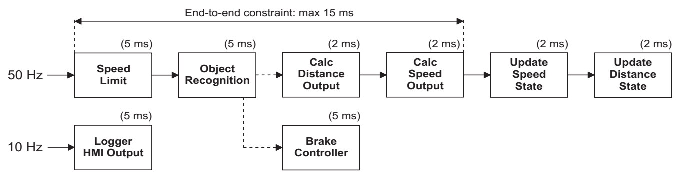


Figure 8.7: Component precedence relation, induced by triggering (computation time in parentheses). Solid arrows represent direct precedence, dashed arrows represent conditional precedence.

triggering is lost. The order of triggering within the ACC model is shown in Figure 8.7. 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 output to an actuator controlling the throttle lever. The constraint is of interest because a quick response to input is good for control stability. Checking such a constraint is done by annotating the model and introducing an extra clock, as described by [28].

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 component has the lowest priority. The computation times are shown in Figure 8.7. 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 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 eventually be executed. 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 seconds using less than 11 Mb of memory.

8.5.4 Synthesis

As described in Section 8.2.2, 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, meaning that testing and debugging could be performed with high observability compared to when using the target hardware. However, to be able to test the whole ACC application an environment model and a control panel had to be developed. The control panel was used to give stimuli (such as throttle lever, brake pedal and AC 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 early.

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 percent belongs to the application and the rest to the operating system. The CPU utilisation in the different application modes was 7, 12 and 15 percents, respectively.

8.5.5 Evaluation

Although the interaction between the ACC and the rest of the system is simplified compared to a real vehicular system, we believe that the example is complex enough 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 imposed by the SaveCCM semantics allowed analysis of properties crucial to ensure correct real-time behaviour, such as end-to-end response times. Likewise, the integration of CCSimTech provided good support for testing.

The resulting system is sufficiently resource efficient. It utilises only a small part of the available capacity in 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 minimise 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.

8.6 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 treated through 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, which has been used for continuous 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 allow derivation of specialised formal models, which enables 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, where changes to components and connections are allowed during run-time, has been discarded in favour of a more rigid approach where dynamicity is achieved by explicit switch elements. This allows 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 well it suits embedded systems outside the vehicular domain. We also want to extend the number of integrated analysis tools to better cover the various needs in different phases of the development process. Other future research directions include integrating the technology with a real-time database mechanism for structured handling of shared data, and with run-time monitoring support.

Bibliography

[1] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, April 1995.

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

[3] W. Wolf. What is embedded computing? *IEEE Computer*, 35(1):136–137, January 2002.

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

[5] Kurt-Lennart Lundbäck, John Lundbäck, and Mats Lindberg. Development of dependable real-time applications. Arcticus Systems, December 2004.

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

[7] Rob van Ommering, Frank van der Linden, Kramer Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, march 2000.

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

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

[10] International Electrotechnical Commission IEC. *International Standard IEC 61131, Programmable controllers*, 1992.

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

[12] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[13] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, 2003.

[14] Mikael Åkerholm, Anders Möller, Hans Hansson, and Mikael Nolin. Towards a dependable component technology for embedded system applications. In *10th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems*. IEEE, 1 2005.

[15] Anders Möller, Jakob Engblom, and Mikael Nolin. Developing and testing distributed can-based real-time control-systems using a single pc. In *10th international CAN Conference*, Roma, Italy, March 2005.

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

[17] Jan Carlson, John Håkansson, and Paul Pettersson. SaveCCM: An analysable component model for real-time systems. In *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[18] Anders Möller, Ian Peake, Mikael Nolin, Johan Fredriksson, and Heinz Schmidt. Component-based context-dependent hybrid property prediction. In *ERCIM Workshop on Dependable Software Intensive Embedded Systems*, Porto, Portugal, September 2005.

[19] Jonas Elmqvist, Simin Nadjm-Tehrani, and Marius Minea. Safety interfaces for component-based systems. In Rune Winther, Bjørn Axel Gran, and Gustav Dahll, editors, *SAFECOMP*, volume 3688 of *LNCS*, pages 246–260. Springer, 2005.

[20] Alexander Dimov and Sasikumar Punnekkat. On the estimation of software reliability of component-based dependable distributed systems. In Ralf Reussner et al., editors, *International Conference on Quality of Software Architectures (QoSA)*, volume 3712 of *LNCS*. Springer-Verlag, September 2005.

[21] Massimo Tivoli, Johan Fredriksson, and Ivica Crnkovic. 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, July 2005.

[22] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[23] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[24] Sergio Yovine. KRONOS: A verification tool for real-time systems. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, October 1997.

[25] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In *Proceedings of CAV'96*, pages 244–256, 1996.

[26] Alexandre David and Wang Yi. Modelling and analysis of a commercial field bus protocol. In *Proceedings of the 12th Euromicro Conference on Real Time Systems*, pages 165–172. IEEE Computer Society, 2000.

[27] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. 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*, pages 2–13, 1997.

[28] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gearbox Controller. *Int. Journal on Software Tools for Technology Transfer*, 3(3):353–368, 2001.

[29] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in LNCS, pages 67–82. Springer–Verlag, 2002.

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

Chapter 9

Paper D:
A Model for Reuse and Optimization of Embedded Software Components

Mikael Åkerholm, Joakim Fröberg, Kristian Sandström, and Ivica Crnkovic
In 29th International Conference on Information technology Interface (ITI 2007), IEEE, Cavtat, Croatia, June, 2007

Abstract

In software engineering for embedded systems generic reusable software components must often be discarded in favor of using resource optimized solutions.

In this paper we outline a model that enables the utilization of component-based principles even for embedded systems with high optimization demands. The model supports the creation of component variants optimized for different scenarios, through the introduction of an entrance preparation step and an ending verification step into the component design process. These activities are proposed to be supported by tools working on metadata associated with components, where the metadata is possible to automatically retrieve from many development tools.

This paper outlines the theoretical model that is the basis for our current realization work.

9.1 Introduction

Component-Based Software Engineering (CBSE), promises independent development and reuse of software components [1]. The foundation is that general components are reused in many applications, and that problems with architectural mismatches can be eliminated [2]. However, there are studies, e.g., [3, 4, 5] indicating that the development of reusable components in comparison with optimized components for certain applications requires up to five times the effort. A substantial part of the extra effort involves development addressing potential future usage scenarios.

Due to extra-functional requirements present in embedded systems, software must often be optimized and tailored for each application [6]. Embedded systems are often produced in high volumes, implying that smaller memory capsules and cheaper processors has high impact on the total production cost. To enable the verification of other extra-functional properties, e.g., reliability, safety, and timing; design choices must be simple in order to enhance predictability, testability, and analyzability. Thus, reusable components, which are bigger and more complex, are often discarded for optimized solutions.

There are several promising component technologies for embedded systems, e.g., the Rubus component technology [7], Koala [8], and our research prototype SaveCCT [9]. These technologies proves that different important needs for embedded systems can be satisfied, e.g., real-time support, and resource efficient run-time systems. However, in industrial case-studies where SaveCCT have been applied, we have found that much of the necessary support is provided (or possible to provide) but that the need to optimize components for certain applications remains a challenge [9].

The optimization problem has also been recognized in related research, and a classification of different techniques is presented in [10]. Common for many of these techniques is the support for configuration of components, e.g., [11, 12]. However, the flip-side is that future scenarios must be predicted, and that the configuration code increase complexity and thereby resource usage. The other main principle for existing techniques is to apply external adaptation through wrappers [13], or adaptors [14]. The main limitation here is that optimization of the componentŠs internal realization is not possible, e.g., it is not possible to remove functionality. Thus, these techniques it is not suitable for resource constrained embedded systems.

To address the problem, we are creating a framework supporting engineering activities related to optimization and adaptation of components. The framework should be used in combination with a component technology, in our case it will be a part of SaveCCT. In this paper we present the founding model for the framework, and this model is the contribution. The model is based on using component metadata, most of which can be automatically retrieved from development tools. Associating metadata with components is common, e.g., the MS .Net framework [15] uses metadata for certain run-time properties. In [16] it is showed how meta-data can be used to improve the test phase. In our work we use that idea and extend it to cover the whole component development phase. Similar to Built-In-Test (BIT) [17, 18, 19], our model includes reuse of tests, but as specifications and results in the metadata instead of executable test cases embedded in the components. In an initial phase of component design, our model supports preparation activities such as selection of a suitable candidate component to adapt, given a set of requirements forming a new usage scenario. This initial phase provides an estimate of the amount of specialization that must be performed. The need for similar component retrieval support has also been recognized in, e.g., [20, 21]. During component design our model collects key metadata from the tool-suite, in the design, realization, and test phases. At the end of the process the model supports verification activities such as detection of side-effects that have occurred during the specialization process.

In section 2, an overview of the proposed model is given. In section 3, the central distinction of components, variants, and versions is defined. Section 4 presents the metadata that is a core part of the model, while algorithms using the metadata are presented in section 5. Section 6 demonstrates the model by an example. Finally section 7 concludes the paper.

9.2 Model Overview

Figure 9.1, shows a schematic overview of the suggested model, fitted into a design process for software components. Characterizing for CBSE is that component development and system development (using components) are separated activities. It is important to be aware of that the focus in this work is on the component development process, and that the majority of the research targeting software components are concerned with system development using components Referring to the figure, the shown design process prior the integration of our model can be imagined as a waterfall model with four steps, design, realization, test execution, and finally delivery to the component repository. The main characteristics to emphasize after the introduction of our model are:

- There is a preparation step added as an entrance step into the process. At this stage, given the requirements forming a new usage scenario, the

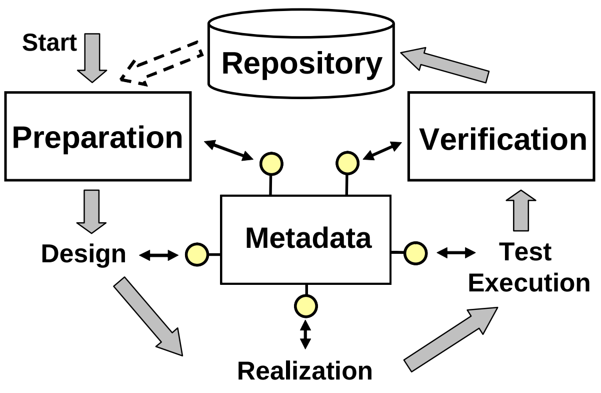


Figure 9.1: Overview of the model

decision to create a component from scratch or to select a component to specialize are taken through evaluation of the amount of work needed for specialization. The output from this step is a plan, or work-order, guiding design and verification efforts.

- There is an additional verification step at the end of the process. Here unplanned side-effects (not according to the plan from the work-order) are detected, e.g., functionality that has changed without intention in a specialization.
- The model is based on metadata, which is automatically retrieved in the design process. Given that tools are capable of exporting data, the need for manual intervention is small.
- Not shown in the figure, but also a central concept, is that the model distinguish components from variants and versions in the repository. This is described in next section.

9.3 Components, Variants, and Versions

In the repository a component may exist in several variants and versions. An overview of the repository is shown in Figure 9.2. The elements shown in the figure are defined below:

- The repository $Rep = \{C_1, \cdots, C_n\}$. The repository on the top-level stores all components in a flat set. The structure is flat since the components have no dependencies to each others in contrast to, e.g., object-oriented approaches were the inheritance relations may affect the storage structure.

- C_i is an abstract component. It is a root node in the repository representing all variants of the ith component in the repository. $C_i = \{C_{i1}, \cdots, C_{in}\}$. The structure is flat indicating no interdependencies between the different variants; they are separate units for usage and maintenance.

- Each variant may exist in several versions $C_{ij} = \{C_{ij1}, \cdots, C_{ijn}\}$. Versioning of the variants is handled according to the rules of common version management theory. The version created latest in time will have the highest version number.

- Referring to a component C_{ijk}, means version k of variant j of the i^{th} component in the repository. $Cijk$ is a concrete component in a component technology, e.g., [9, 8], according to common component definitions, e.g. [22].

Assume that the function $Req(x)$ gives the set of uniquely identified requirements fulfilled by element x. How this is realized is described in the next section. The following guarding conditions must be fulfilled for a software element to qualify as a variant, or version of a component respectively:

- *Commonality guard* - for all variants j and versions k of component i, $\{\cap_{jk} Req(C_{ijk})\} \neq \emptyset$. This implies that there must be at least one requirement in common between all variants and versions of a certain component. If this guard is not fulfilled, the variants and versions cannot be stored under same component.

- *Compatibility guard* - for a new version $k+1$ of variant k, $Req(C_{ijk}) \subseteq Req(C_{ijk+1})$. This implies that a new version of a variant should fulfill at least the same requirements as the previous version. When this strict guard is fulfilled the new version is backwards compatible with the older version, typically bug-corrections and improvements will sort under this category. In our model, if this guard is not fulfilled the component may be qualified as a new variant; otherwise a new component should be created.

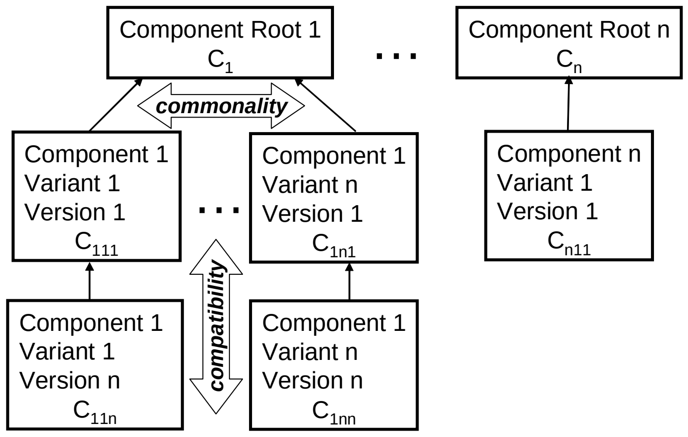


Figure 9.2: Repository Layout

9.4 Metadata Definition

Metadata units are associated with all concrete components, i.e., all versions of all variants of a component. The metadata manage requirements, elements of design and verification of the software in the repository.

An overview of the metadata is shown in Figure 10.4. The figure show more metadata compared to what will be formally defined in this paper; this is to give an idea of the overall concept. The core metadata (thicker lines in the figure), are the necessary parts required to provide the support that is emphasized in this paper. Non core parts may be useful when browsing the repository, e.g., containing abstract, keywords, usage statistics, and key design patterns practiced when the component was developed .

To define the core parts of the metadata, let $M_{ijk} = (S_{ijk}, G_{ijk})$ be the metadata associated with C_{ijk}. S_{ijk} is a specification of the component $S_{ijk} = (R_{ijk}, D_{ijk}, V_{ijk})$ where:

- R_{ijk} is a set of uniquely identified requirements $R_{ijk} = \{r_1, \cdots, r_n\}$. R_{ijk} contains all documented requirements that the software element tries to fulfill, including both functional and extra-functional requirements. The actual formulation or semantics of the requirement is not strictly required. The important matter is that a unique identity is asso-

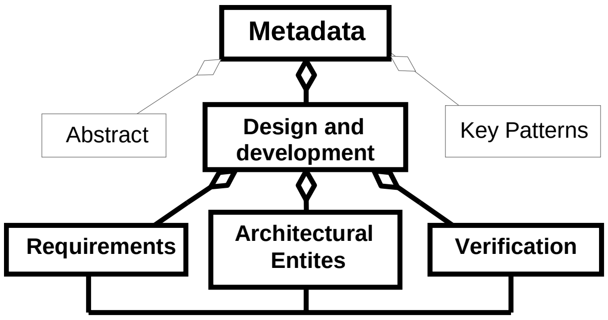


Figure 9.3: metadata associated with C_{ijk}

ciated with each requirement.

- D_{ijk} is a set of uniquely identified architectural entities $D_{ijk} = \{d_1, \cdots, d_n\}$. Depending on the realization of the software element, these design entities can be different artifacts, e.g., functions, data structures, objects, components or analysis. As for requirements, design entities must be associated with a unique identifier.

- V_{ijk} is a set of uniquely identified verification cases $V_{ijk} = \{v_1, \cdots, v_n\}$. V_{ijk} includes all test-cases, together with expected results, and also obtained results after the test phase. As for R_{ijk} and D_{ijk} each case needs to be represented. $G_{ijk} = (CR_{ijk}, VR_{ijk})$, contains manually defined relations, over the automatically derived sets D_{ijk}, R_{ijk}, and V_{ijk}.

- $CR_{ijk} \subseteq D_{ijk} \times R_{ijk}$, represents the causal relationships between elements of the design and their respective requirements. It represents the reason, or the cause, for design elements to exist.

- $VR_{ijk} \subseteq V_{ijk} \times (R_{ijk} \cup D_{ijk})$ represents the verify relationships from elements of the verification cases, to which requirements and/or design entities, each case verifies. Relations from V_{ijk} to D_{ijk} represent white-box test cases, while edges from V_{ijk} to R_{ijk} represent black-box cases.

9.5 Central Algorithms on the Metadata

Figure **??** emphasized the support provided first and last in the component design process, the algorithms applied in the two different stages are described in the following sub-sections.

9.5.1 Preparation

When a need for a new component is detected, a central decision is to decide if the new component should be obtained through adaptation of an existing component or if a new component should be developed. To support this decision, the metadata can be used to compare candidates for reuse and adaptation. The following expressions determine what requirements that are addressed by variants and versions of a specific component:

- SR_i derives all requirements shared by all variants and versions of a component. It is defined as the intersection of all requirements addressed by all entities of a certain component C_i: $SR_i = \{\cap_{jk} Req(C_{ijk})\}$
- NR_i is the set containing the requirements addressed only by a subset of the variants and versions of a certain component C_i. $NR_i = \{\cup_{jk} Req(C_{ijk})\} - SR_i$.
- $AR(r)$ gives the set of versions and variants that address the requirement r, of a certain component C_i, $AR(r) = \{C_{ijk} \mid \{r\} \subseteq Req(C_{ijk})\}$

The application of the expressions above provides overview information about the components. We can see divide requirements into those addressed by all versions and variants, and those requirements addressed by certain subsets. With this information developers are guided in the choice of candidate components to investigate in the work to find a suitable component to reuse.

It is possible to derive a work-order for each concrete component, i.e., C_{ijk}. Initially work-orders are used to estimate the amount of work to apply changes to certain concrete components to fit a new usage scenario. Thus, finding the most feasible candidate to adapt is supported by comparison of work orders. A component whose work order shows little need for adaptation is likely a suitable starting point for a new variant. Later, during the development, the work is guided by the work-order. For a certain concrete candidate C_{ijk}, and given the requirements forming a new usage scenario, the work order show what design entities and what test cases to reuse as-is, to change, and to remove.

It also shows what requirements that remains unimplemented and thus will require new development. The functions that are needed to be applied on the metadata are defined here.

An estimation of consequences of a changed requirement, r, in terms of the set of affected design entities, $AD(r)$, and set of affected test cases, $AT(r)$, is determined through:

- $AD(r) = \{x \mid CR_{ijk}(x, r)\}$
- $AT(r) = \{x \mid VR_{ijk}(x, r)\}$

The consequences of a removed requirement, r, in terms of affected design entities can similarly be determined by the same expressions. However, to determine if the deign entity or test case is not only affected, but according to the relationships expressed in the graphs can be removed, we must take the whole set of all removed requirements into consideration. Let RR be the set of requirements that is planned to be removed. Design entities that may be removed are determined through the function $RD(RR)$. Similarly test-cases that may be removed are derived by the function $RT(RR)$.

- $RD(RR) = \{x | \neg\exists r : [CR_{ijk}(x, r) \wedge r \in R_{ijk} - RR]\}$
- $RT(RR) = \{x | \neg\exists r : [VR_{ijk}(x, r) \wedge r \in R_{ijk} - RR]\}$

9.5.2 Verification

When a resulting variant or version is created based on reuse of another, it is possible to detect unplanned effects of the changes. To detect unplanned side-effects that may have occurred in the process, regression testing is applied based on information in the work order. The only allowed changes between the results of reused test cases are those we knew would be affected in the work order. If any other changes are detected, they must be investigated. There can be one of two reasons that must be corrected by the developers:

- Unplanned or unnecessary parts were changed during the development of the new variant, which must be found and corrected.
- Undocumented dependencies in the relations CR_{ijk} and/or VR_{ijk} should be updated and added to achieve a continuous improvement of the decision supporting relations. It may also be useful to store statistics when undocumented dependencies are discovered, to estimate a precision for work orders.

9.6 Usage Example

Now that we have defined the elements in the model we will demonstrate the support for design decisions. We do this through a simplified industrial case.

9.6.1 Initial Component

As a part of an order of a larger system, a component providing an interface to a CAN chip is ordered forming requirements R_{ijk} as below:

$$R_{ijk} = \left\{ \begin{array}{lc} (11, & Send), \\ (12, & Receive), \\ (13, & EnableRemoteReply), \\ (14, & worstcaselatencyforSend1ms) \end{array} \right\}$$

Given that this component is built from scratch, and stored in an empty repository, i.e., $Rep = \{C_1\}$, where $C_1 = \{C_{111}\}$. Depending on the developers design decisions, D_{111}, and V_{111} of the local metadata associated with C_{111} may have the following structure in the repository:

$$D_{111} = \left\{ \begin{array}{lc} (21, & FrameTypes), \\ (22, & ReceiveBuffer), \\ (23, & Send), \\ (24, & Receive), \\ (25, & EnableRemoteReply), \\ (26, & Analysisofsend, result500ms) \end{array} \right\}$$

$$V_{111} = \left\{ \begin{array}{lc} (31, & receiveBufTest, \\ (32, & sendTest, \\ (33, & receiveTest, \\ (34, & remoteReplyTest, \\ (35, & timeAnalysisSend, \end{array} \right.$$

$$\left. \begin{array}{cc} expected : oldestDropped, & observed : oldestDropped), \\ expected : allSent & observed : allSent), \\ expected : (2,3,66), & observed : (2,3,66)), \\ expected : remoteFrame2, & observed : remoteFrame2), \\ expected :< 500ms, & observed : 450ms) \end{array} \right\}$$

Notice that the requirements, design, and verification sets R_{111}, D_{111}, and V_{111} should be automatically created, given that it is possible to export data from the tools. However, the causal and verification relations CR_{111} and

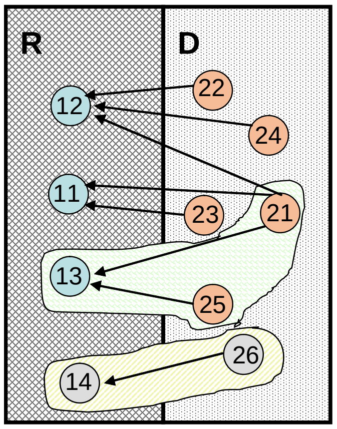


Figure 9.4: causal relations

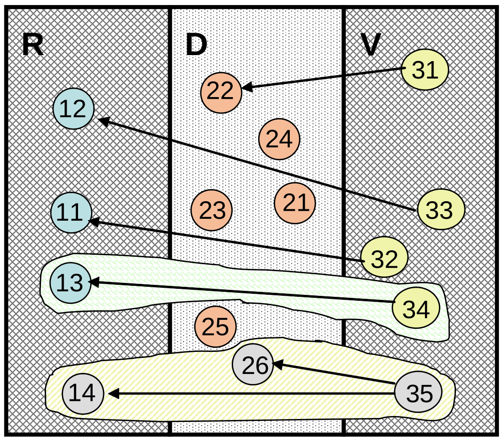


Figure 9.5: verify relations

VR_{111} remain to be manually defined. These relations can be presented and created graphically, through directed graphs. CR_{111} and VR_{111} for the case are defined below, and the corresponding graphs are shown in figure 9.4 and 9.5 respectively. For now, ignore the fields surrounding, e.g., nodes with id 14 and 26.

$CR_{111} = \{(21, 11), (21, 12), (21, 13), (23, 11), (22, 12), (24, 12), (25, 13), (26, 14)\}$

$VR_{111} = \{(31, 22), (32, 11), (33, 12), (34, 13), (35, 26), (35, 14)\}$

The metadata unit for C_{111} is now complete:

$M_{111} = (S_{111}, G_{111}) = ((R_{111}, D_{111}, V_{111}), (CR_{111}, VR_{111}))$

We have now the initial version of a variant of a component that can not only be reused. The component is also prepared for adaptation and specialization to form new variants addressing sets of requrements forming other usage scenarios.

9.6.2 New Component

In negotiation with another customer at a later point in time, the requirements on a similar component as a part of another system forms R_{ijk} as below. Requirement id 13 has changed, indicated here only by a '*', requirement id 14 has been removed, and requirement id 15 has been added.

$$R_{ijk} = \left\{ \begin{array}{lc} (11, & Send), \\ (12, & Receive), \\ (13, & EnableRemoteReply*), \\ (15, & GetRemoteFrameStatistics) \end{array} \right\}$$

Applying the expressions in section 5.1, the work-order contains the information in table 9.1. The results from the expressions are visualized in figure 4 and figure 5. The fields in the figures surrounding certain relations show the same as the table, e.g., that due to changes in requirement id 13, design entities 21, 25 may be affected as well as test case 35.

The process may proceed guided by the work order, eventually when tests are complete the results are verified according to section 5.2. In this case according to the work-order it is expected that test case 34 might show other results, and that observed results of cases 31, 32, 33 should be unchanged.

	Design Entities	Test Cases
Reuse ids	22, 23, 24	31,32,33
Affected ids	21, 25	34
Remove ids	26	35
Covered requirement ids: {11,12,13}		
Uncovered requirement ids: {15}		

Table 9.1: A work order for the specialization

9.7 Conclusions

We are convinced that component-based principles are beneficial for all types of software. Mature engineering disciplines always use standardized components. One of the most important prerequisites for component based principles is that components are general, so that they can be (re)used many times. This prerequisite has shown be hard to meet in development of certain software, e.g., embedded software with high specialization demands.

This paper introduces a model that supports developers of embedded software components in using optimized variants of components. The benefits are achieved by introducing a start and a completion step into a regular design flow. The completion-phase provides automatic detection of accidentally introduced side effects in redesign. The starting phase supports the selection of the best matching candidate from a repository of components given a set of requirements.

The model is based on associating metadata with components, and can be highly automated and integrated in an existing development tool-suite, given that it is possible to export data from the tools. An industrial case study is planned, where a prototype realization will be integrated in an existing tool-suite at a sub-contractor company. A sub-contractor company is often faced with challenges in adapting and customizing components to the different needs of customers with varying system architectures and choices in technology and standards. Managing adaptation and optimization of components is therefore a key value for sub-contractors.

Bibliography

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

[2] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the Seventeenth International Conference on Software Engineering*, April 1995.

[3] Ivica Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, December 2001.

[4] Ivica Crnkovic and Magnus Larsson. A case study: Demands on component-based development. In *Proceedings, 22th International Conference of Software Engineering*, Limerick, Ireland, May 2000. ACM, IEEE.

[5] M. Mrva. Reuse factors in embedded systems design. *IEEE Computer*, 30(8):93–95, 1997.

[6] W. Wolf. What is embedded computing? *IEEE Computer*, 35(1):136–137, January 2002.

[7] K.L. Lundbäck and J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. Arcticus Systems: http://www.arcticus.se (Last Accessed: 2005-01-18).

[8] R. van Ommering, F. van der Linden, and J. Kramer. The koala component model for consumer electronics software. In *IEEE Computer*, pages 78–85, March 2000.

[9] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The

save approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[10] G. T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

[11] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 5(41), 1999.

[12] K. Cooper, J. Zhou, H. Ma, I. L. Yen, and F. Bastani. Code parameterization for satisfaction of qos requirements in embedded software. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003.

[13] J. Brant, B. Foote, R. e. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings of 12th European Confernece on Object-Oriented Programming (ECOOP98)*, July 1998.

[14] D. M. Yellin and R. E. Strom. Protocol specification and component adaptors. *ACM Trans. on Programming Languages and Systems*, 2(19):292–333, March 1997.

[15] J. Conard, P. Dengler, B. Francis, J. Glynn, B. Harvey, B. Hollis, R. Ramachandran, J. Schenken, S. Short, and C. Ullman. *Introducing .NET*. Wrox Press Ltd, 2000. ISBN: 1-861004-89-3.

[16] A. Orso, M. J. Harrold, D Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontents to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance*, November 2001.

[17] H.G. Groß, M. Melideo, and F. Barbier. Component+ Methodology. Technical report.

[18] K.J. Fernandes, V.H. Raja, and M. Morley. A System Level Testing Modeling Mechanism in a Reengineering Environment. *Journal of Conceptual Modeling*, 2001.

[19] Y. Wang, G. King, D. Patel, S. Patel, and A. Dorling. On coping with real-time software dynamic inconsistency by built-in tests. *Annals of Software Engineering*, 7(1):283–296, 1999.

[20] Y. Li, J. Yin, and J. Dong. A Component Management System for Mass Customization. *Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences-Volume 2 (IMSCCS'06)-Volume 02*, pages 398–404, 2006.

[21] H. Mili, F. Mili, and A. Mili. Reusing software: issues and research directions. *Software Engineering, IEEE Transactions on*, 21(6):528–562, 1995.

[22] G. T. Heineman and W. T. Councill. *Component-based Software Engineering, Putting the Pieces Together*. Prentice-Hall, 2001. ISBN: 0-201-70485-4.

Chapter 10

Paper E: Introducing Component Based Software Engineering at an Embedded Systems Sub-Contractor

Mikael Åkerholm, Kristian Sandström, and Ivica Crnkovic
submitted for publication

Abstract

Attractive benefits with successful implementation of component-based principles include managing complexity, reduction of time-to-market, increased quality, and reusability. Deployment of component-based development is however not simple - it depends on many strategic, technical, and business decisions. In this paper we report experiences from our attempts with finding a correct implementation of component-based principles for the business situation of sub-contractors of embedded systems.

Findings related to suitable component models, component technologies, and component management is presented.

10.1 Introduction

In this paper we report our experiences from introducing Component-Based Software Engineering (CBSE) [1, 2] at the company, CC Systems[1], acting as sub-contractor and Commercial Off The Shelf (COTS) components supplier for embedded systems.

CBSE can be seen as analogous to engineering approaches in other engineering domains. For examples, mechanical engineers build systems using well-specified components such as nuts and bolts, and the building industry uses components as large as walls and roofs (in turn assembled from smaller components). CBSE has proven to be effective for desktop and web-applications, however, not yet for development of software for embedded systems. Implementation and deployment of CBSE for development of embedded systems is not trivial. As its success depends on many factors where some of them are, selection (or development) of a component technology that can be efficiently used in the development and maintenance process, and satisfy the run-time requirements of the particular domain.

We present experiences from four cases in this paper. One case, SaveCCT, is a study where a group of researches demonstrates a prototype component technology in a real industrial environment. The second case, CrossTalk, utilises CBSE principles for realizing a software platform supporting "any" system consisting of the company's hardware. The third case, CC Components, make use of a component repository when possibilities to create or reuse components arise in the development projects. Finally, the fourth study is an evaluation of a method supporting the sometimes necessary work with adaptation of components to fit usage in different development projects.

Our findings indicate that CBSE principles are suitable for embedded systems sub-contractors, but it might be harder to practice CBSE as sub-contractor than product owning company. The necessary technical needs of, e.g., expressiveness in the component models, resource efficiency of component based applications, and analysis possibilities can be considered met.

The following section (section 10.2) presents the motivation and goals with the research presented in this paper. Section 10.3 presents data for the different cases we have studied. The findings from our experiences when working with the different cases are reported in section 10.4. A discussion of the results is provided in section 10.5. Finally, section 10.6 concludes the paper.

[1]Cross Country Systems, http://www.cc-systems.com

10.2 Goals and Motivation

The primary goal of this experience paper is to contribute to the overall understanding of the needs of component technologies and processes for practicing CBSE in the domain of embedded control systems for vehicles and machines.

The studies have all been performed at CC Systems engineering sites in Finland and Sweden. CC Systems develops electronics targeting vehicles and machines in rough environments. From software and hardware controlling safety critical by-wire functions, to software and hardware for powerful on-board display based information systems with back-office connections.

The studies focus on the control related part of the systems. The focus is chosen because CBSE as approach have had a limited success for development of such systems. These systems are the most critical for the overall vehicle functionality, with maximum demands on qualities such as timeliness, safety, and reliability. It is also known that these qualities are not addressed by most existing commercial component technologies, and consequently these systems cannot be developed with such component technologies. Many component technologies that might be suitable exists within academia and some are to a limited extent used within industry, e.g., Koala [3] used internally at Philips [2], Rubus [4] used by some Swedish vehicle manufacturers, and different implementations of the IEC61131-3 standard [5]. However, as pointed out in [6], there is currently no de-facto standard component technology within the domain of vehicular systems; although CBSE seams to get a lot of attention from industry, e.g., East[3] and Autosar[4]. This leads us to the goal of assessing if the limited success of CBSE in the domain is depending on an inability of existing commercial technologies to support the requirements of embedded vehicular applications.

Another very interesting question is how an ideal component model for the domain should take the trade-off between supporting predictability, and ease to express common functionality in vehicular control systems? The core part of a component model is related to defining what a component is, and possibilities for component interaction. There are several important design decisions that have to be made when defining a component model and one is the trade-off between flexibility and predictability. It is the trade-off with respect to ease of implementing vehicular control systems (high degree of flexibility), and support for prediction of quality attributes considered important in the domain

[2]Philips, http://www.philips.com/

[3]East, http://www.east-eea.net/

[4]Autosar, http://www.autosar.org

(high degree of predictability). A design choice of our suggested component model, SaveCCM [7], is in contrast to many of the current component models to sacrifice some flexibility to facilitate analysis and predictability.

Finally resource-efficiency (the consumption of a minimum of resources in achieving an objective) is important in the domain since products are typically produced in high volumes. Poor resource efficiency in component frameworks might be an important reason for not choosing CBSE [6]. At the same time the basic ideas of CBSE has been driven from the needs of PC/Internet applications where resource efficiency typically is not an issue. A reusable component should according to CBSE theory be general. An obvious method to create a general component is to implement support for many methods that might suit different purposes. Using this approach imply that you might end up with using only a small part of the component, and the rest is "dead code" in the application. This might be a problem for resource constrained embedded systems, and highly recommended to avoid for systems with high Safety Integrity Level (SIL), i.e., SIL 2 and above in IEC61508 [8].

10.3 CBSE Activities

Four CBSE activities are summarized in the following sub-sections, denoted *Case 1-4*. *Case 1* is a study by a group of researchers in a real industrial environment; the study evaluates technical properties of a component technology. In *Case 2*, the company utilises CBSE principles for realizing a Product-Line Architecture (PLA) for platform software, here the suitability of CBSE and the component technology are evaluated. In *Case3* component-based reuse is practiced when opportunities arise, here the CBSE principles in this context is evaluated. Finally, *Case 4* is an evaluation by researchers in an industrial environment of a method related to component adaptation, here the method itself is evaluated. The experiences and lessons learned of the cases are summarised in section 10.4.

10.3.1 Case 1, SaveCCT

Case 1, is a demonstration of the component technology SaveCCT [9], by usage on a fictive but representative application in a real industrial environment at CC Systems. The main purpose with this case is evaluation of the technical properties of the component technology.

The research prototype of the SaveComp Component Technology (SaveCCT)

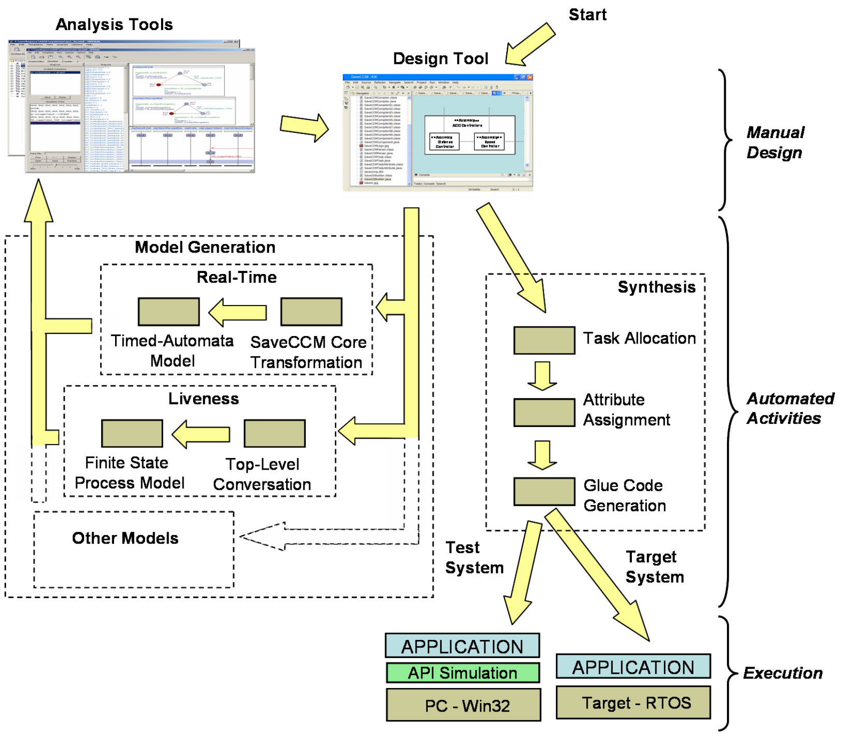


Figure 10.1: An overview of our research prototype SaveCCT

used during this study is visualized in Figure 10.1. SaveCCT is described on the top level, by distinguishing manual design, automated activities, and execution.

Manual design is the entry point for the development; here a component-based strategy is used, supported by a set of tools for design and analysis. The SaveCCT design tool provides support for graphical assembly of applications from existing components. The tool allows designers to specify the component interconnection logics, and express high level constraints on the resulting application. Assembling components is done with respect to the rules of the SaveComp Component Model (SaveCCM) [7]. 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. As shown in the figure, SaveCCT incorporates a number of analysis tools, which can be used for verifying specific attributes of the application, e.g., related to timeliness and safety. To efficiently incorporate an analysis tool, as much as possible of the translation from the model created with the design tool to the model required by the desired analysis tool should be automated. In this study we incorporated LTSA [10], and Times [11].

Automated activities produce necessary code for the run-time system (i.e., glue-code), and different specialized models of the application for analysis tools. The synthesis activity generates all low level code (i.e., hardware and operating system interaction), meaning that components are free from dependencies to the underlying platform. Furthermore, the code generation step statically resolves resource usage and timing, with the strategy to resolve as much as possible during compile-time instead of depending on costly run-time algorithms.

To achieve efficient and predictable run-time behaviour, and reliable support for pre-runtime analysis, SaveCCT assumes a real-time operating system (RTOS) as underlying platform. The prototype used the Quadros [5] RTXC operating system, which is a standard fixed-priority pre-emptive multitasking RTOS used in some applications by CC Systems. The supported target hardware in the current version is CrossFire MX1 from CC Systems, which is an electronic control unit intended for control systems running in rough environments. To facilitate testing and debugging we incorporate CCSimTech [12], which is a simulation framework that offers generic hardware emulation components for common hardware in embedded systems, e.g., I/O (digital and analogue), network technologies, and memories. This environment represents a typical platform used in development projects by CC Systems, and thus serves as an example of an industrial environment for the SaveCCT prototype.

10.3.2 Case 2, CrossTalk

Case 2, CrossTalk [13], is an initiative driven by CC Systems, which have been taken influences from the research demonstrated in *Case 1*. The main goal with this initiative is to take advantage of the support for product-line architectures that component-based approaches give. The goal is rapid and costs effective assembly of platform software, through enabling addition and/or replacement of components to a baseline platform depending on the needs from a certain application. The CrossTalk platform has been used in numerous real development projects by CC Systems.

A CrossTalk based system is built on an open-ended component-based CrossTalk platform, the concept is to have *one* platform to build *any* system

[5]Quadros, http://www.quadros.com/

consisting of the company's own hardware. Figure 10.2 illustrates the concept, which we describe here with the following list:

1. System architecture, in terms of computer nodes and their responsibility is established. The hardware for a CrossTalk system are selected among more than ten different nodes, e.g., control modules, communication gateways, and display units. The communication between the different nodes on the machine is based on CANopen[6] this means that it is also possible to integrate any third-party node in the system that uses the CANopen protocol, but treatment of such nodes is beyond the scope of this paper.

2. Based on the functionality designated each computer node, platform components from the CrossTalk repository is selected to constitute software platform for each node in the system. The repository is based on a standard version control system, and the components are IEC61131-3 [5] components. The components are assembled using the CoDeSys tool from 3S[7].

3. The open-ended platform software is deployed on each of the nodes in the system; the application is then built by the customer or by CC Systems in a separate project, by continuing the work in the CoDeSys tool.

10.3.3 Case 3, CCComponents

CC Components is another initiative with influences from *Case 1*. Here the intention is to package reusable parts of applications into software components, and to reuse components when suitable. Notice here that the intention is not to build entirely component-based systems; systems are built through a combination of components and non-component-based software.

To efficiently take advantage of CBSE, development processes for system development, component assessment, and component development are separated, as proposed in e.g., [14].

As demonstrated in Figure 10.3, CC Systems has a system development process based on Rational Unified Process (RUP)[8], thus this process needs no further description here. In the inception phase all development projects should

[6]CiA, CANopen, http://www.can-cia.org/
[7]3S CoDeSys, http://www.3s-software.com/
[8]IBM Rational, http://www-306.ibm.com/software/rational

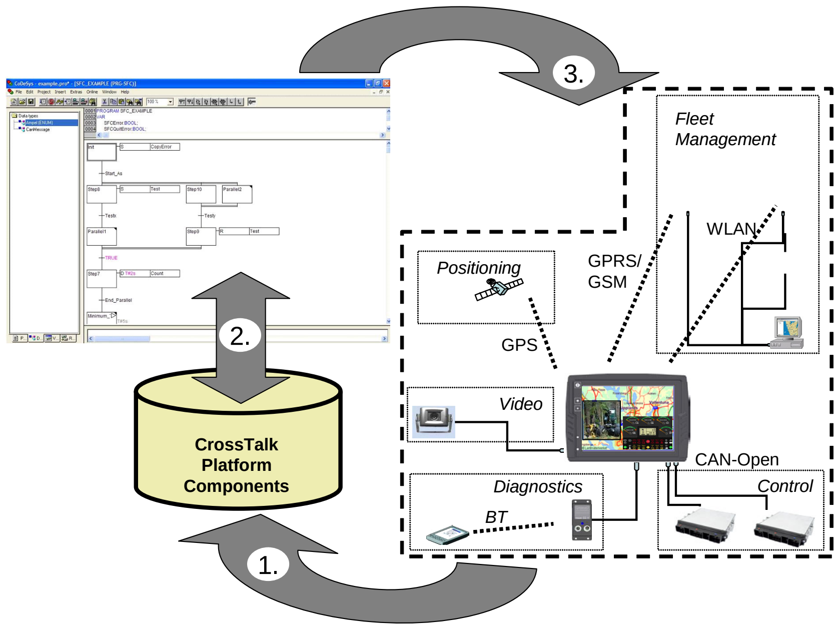


Figure 10.2: Workflow when assembling a software platform for a CrossTalk-based system

initiate a component assessment process with the intention to find suitable components to reuse, and suggestions for specifications of components to develop. The component assessment process has the following steps:

Find, component assessment starts with finding components that might provide the required functionality for some part in the project.

Select, if candidate components for reuse are found, the selection of which components to reuse in the project is documented and motivated.

Specify, if no candidate components are found, but the project finds a certain part of the system very suitable to be packaged as a reusable component a specification for such a component is created.

Evaluate, specifications are evaluated by the CC Components board (a group responsible for the repository). Generally, the board must be convinced that the suggested component will be target for reuse in other projects, before a separate component development project is initiated.

Verify, this step is required to test that the component really fits the intended purpose as soon as possible to avoid that the component is assumed to be fit and well-tested until the very late stages in the system development project.

Component development is guided by the same RUP-based process as system development, but the target to develop reusable components is made clear through lifting the development from the process of a particular project to a separate process ending with delivery to the common company-wide CC Components repository. There is no formal component definition. The only technical requirement is that the components in the repository must have all their dependencies specified in the interfaces, combined with requirements on standardized documentation.

10.3.4 Case 4, Component Metadata for Traceability

Case 4, is an evaluation of a prototype implementation [15] of a method supporting component assessment, and component development. The theory behind the method is described in [16], it is based on work by Orso et.al. [17] suggesting to (re)use component metadata to support software engineering tasks. The need is based on experiences from *Case 3*, where component assessments often results in needs for component adaptations, this will be further discussed

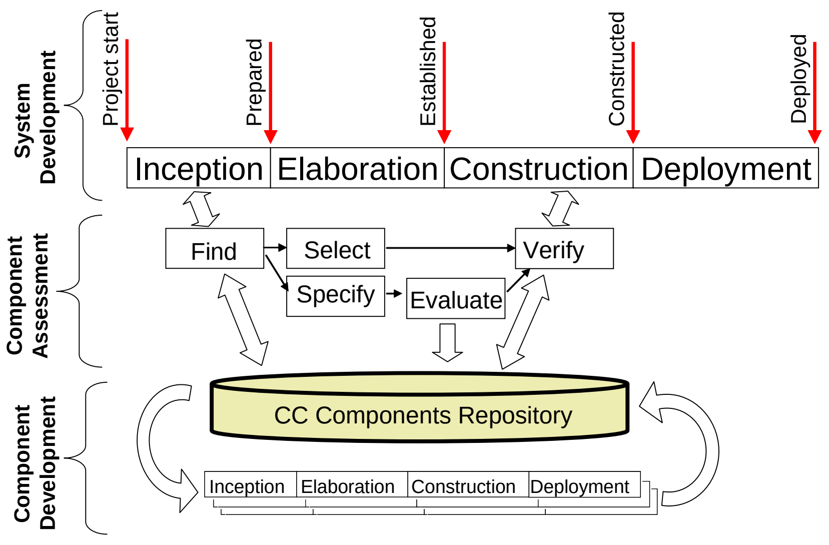


Figure 10.3: CC Components development processes

in section 10.4. This case is based on a prototype demonstration on a representative software component, for a group consisting of two project managers, four developers, and one sales manager.

Figure 10.4, gives an overview of the method. The method affects the processes of component assessment, and component development, and there is a metadata associated with all components which is central in the method. The purpose of the metadata is to maintain traceability of requirements through design and testing during component development.

The metadata is collected during component development, and the arrows in the metadata in the figure illustrates that the metadata contains references between requirements, design, and test cases associated with the component. This information is traceability information of how the requirements fulfilled by the component are related to the internal realization of the component, and how the different test cases relate to different parts of the realization and different requirements.

The component assessment process has, in comparison to *Case 3* Figure 10.3, a new activity after Find called Modify:

Modify, here modification requests of existing components are created, when modifications are necessary for the system development project. These requests are then evaluated by the board of CC Components in the following Evaluation step, as previosly desribed in *Case 3*.

During the component assessment the metadata with traceability is utilized for performing impact analysis of a desired modification of a component. The purpose of the impact analysis is to estimate the amount of work and consequences of performing the desired modification. A prototype tool has been developed that automatically produces analysis of which parts of a components design and test cases that are affected by a modification. Where the modifications are defined through giving the desired change of the requirements. The output is used as input to the evaluation step where the board decides whether the component modification should be performed or dismissed.

During the component development process where a new variant of a component is developed, the impact analysis gives guidance to the work. It gives information of which parts of the component that should be modified. It also specifies which test case that should be used for regression testing after the change, i.e., points out which test cases that must produce the same results.

The need to adapt software components have been known in the CBSE community, e.g., a survey on the topic in 1999 [18]. Common for many of the proposed techniques is the support for configuration of components, e.g., [19, 20]. However, the flip-side with these techniques is that future scenarios must be predicted, and that the configuration code increase complexity and thereby resource usage. The other main principle for existing techniques is to apply external adaptation through wrappers [21], adaptors [22], or connectors [23]. The main limitation here is that optimization of the component's internal realization is not possible, e.g., it is not possible to remove functionality. Thus, none of these techniques is perfect for the problem we have encoutered with resource constrained embedded systems.

10.4 Experiences

In this section we summarize the findings from the above reported cases. We do this case by case.

10.4.1 Case 1

The component model is based on data-flow (or pipes-and-filters) interaction, this has been chosen to give good support for expressing the key functionality

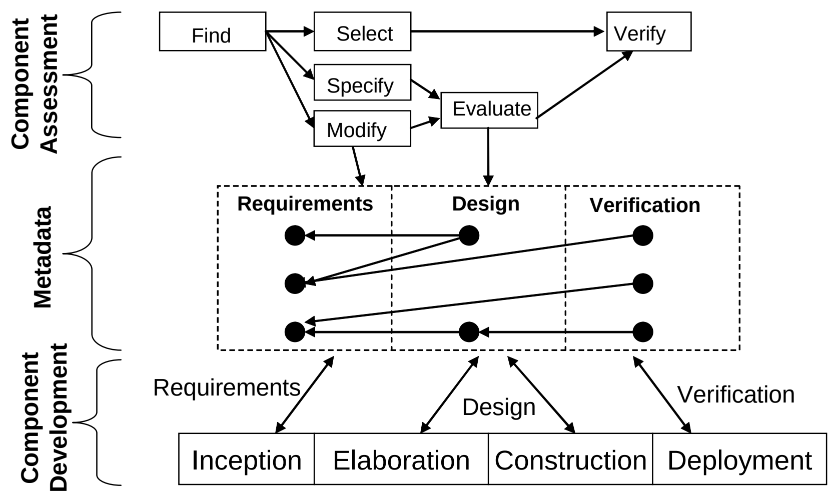


Figure 10.4: Using metadata in component assessment and component development

of control systems. Designing the fictive application according to component-based principles was relatively straight-forward, and SaveCCM proved sufficiently expressive for this type of system.

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

The resulting system was sufficiently resource efficient. It utilizes only a small part of the available capacity in the target hardware, which is approximately the utilization 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.

We should also make clear that the demonstrated component technology was considered unusable in real development projects, since the quality and usability of the included prototype tools was considered below tolerable levels. Mature tools is a basic need in practice.

10.4.2 Case 2

The component model, i.e., IEC61131-3 with its roots in the automation domain, is based on data-flow (or pipes-and-filters) interaction. Studying this case fortify that this is a suitable component interface also in the vehicular domain for control related systems. Numerous control systems have successfully been built for vehicles and machines based on the CrossTalk concept.

Reuse of low-level components (i.e., CrossTalk platform components) through component based principles is successfully practiced in this case. The experience is that product-line architectures for platform software can be efficiently created with component-based principles. The company also stresses that the component-based principles in this case results in short and predictable development projects, and higher software quality.

Regarding resource efficiency, it is known that the CoDeSys run-time framework requires additional processing compared to realizing the system with lower level programming of the hardware. Thus, it is a trade-off between resource efficiency and development efficiency.

10.4.3 Case 3

Overall the introduction of a company-wide component repository is a success, every time a well-tested component can be reused "as is" without modification there is a good return of the investments made in the component development. Reuse was practiced before, but more unstructured, depending of the knowledge about reusable software assets inside the project team. The evaluation step of component specifications is seen as promising for the future to ensure that only reusable components are developed.

However, problems with reusing components have also been identified, which might be similar for other sub suppliers to customers with high volumes or safety critical applications. The basic foundation of CBSE, to build general components that can be (re)used in many applications, is harder to practice for a sub-supplier, especially in the domain of embedded control systems. Components may require functional additions or adaptation associated with reuse. The adaptation needs seems to be higher for the sub-suppliers business case. At the same time, due to domain requirements, components can often not include any extra functionality. Instead of being based on general components, applications must be dedicated and specialized to its task for high volume products. Safety critical applications are even worse since no "dead code" is allowed in source files or on target for certification according to the higher SIL levels of, e.g., IEC61508 [8]. Practicing reuse under these circumstances often require adaptations.

10.4.4 Case 4

The conclusion from *Case 4* is that the method is promising to support the adaptation needs. The method causes no overhead in the internal realisation of software component itself, and the components can be highly specialized for every scenario. If this can be efficiently implemented in practice, it supports the adaptation needs identified at the company.

Another positive side is reuse of the documentation of the traceability information given through the graphs in the metadata. This is becoming more and more important for all companies in the domain, due to legislation of using system safety standards in development. The IEC61508 standard [8], which is the main standard for functional safety of electronic programmable systems, requires traceability of requirements through the different stages of development. CMMI [24], a process improvement approach, even requires bidirectional traceability. This is the ability to trace requirements both forward and backward, i.e., requirements through the development process into the product and from the product backwards to requirements. This becomes possible, reusable, and well-documented, through the metadata.

On the negative side were the developers' concerns about using one tool more, when you ideally would like as few tools as possible to work efficiently. Another concern was the effort to create the dependency graphs, which might be time consuming and complex for big components.

10.5 Discussion

Here we discuss the findings related to the goals stated in section 10.2.

One of the purposes with the studies was to evaluate CBSE as engineering approach for the domain. To start with, the continuous interest and investments from the company in these activities indicates that CBSE is an attractive engineering approach. The success with both increased efficiency and quality in *Case 2* and *Case 3*, generally demonstrates the potential of CBSE and especially when it comes to PLA. It also shows that component technologies with tool-suites that are mature enough for industrial needs exists, here manifested through CoDeSys. Whether the functional blocks of the IEC61131-3 standard qualifies as real software components according to some definition remains unsaid, but it is here proven that it is possible to treat them as components. The conclusion is that the use of CBSE in the domain should not be limited by lack of existing commercial technologies, even if we cannot dismiss this as a reason.

We could also observe that it can be problematic to reuse components for sub-contractors, primary from *Case 3*. Sub-contractors, as CC Systems in this case, might take contracts on realizing similar functions with slightly different requirements for different customers. If the target systems are safety-critical or produced in high volumes, general components (with the side-effect of being bigger) must often be discarded in favour of solutions tailored for the particular system.

It is interesting that it within the same company was possible to observe the usage of CBSE as product owner in *Case 2*, and as sub-contractor *Case 3*. The experiences from this is that it is definitively easier to take advantage of CBSE being a product-owner, in fact being a product owner you actually plan for reuse. The next generation of the system will most likely be an improvement of the existing system; it becomes natural to take reuse of existing components into consideration when planning for the next generation.

Next important concern from section 10.2 is the trade-off between flexibility and predictability in component models. Both in *Case 1* and *Case 2*, where more formal component models were used, the component model was based on data-flow (or pipes-and-filters) interaction, this has been chosen to give good support for expressing the key functionality of control systems. Designing the fictive application according to component-based principles was relatively straight-forward, and SaveCCM proved sufficiently expressive for this type of system. CrossTalk has been used for numerous real control systems and it has proven to be suitable in every case. The basic interaction mechanism is thus well proven in practice, but important to stress is that we cannot dismiss other component interaction approaches from our studies. The analysis of real-time and reliability properties demonstrated in *Case 1* shows that is possible to create a component model that is expressive enough for the applications and at the same restrictive enough to allow this type of predictions. However this has not been proven in real projects.

The experiences also justify apprehensions concerning risks of poor resource efficiency of component-based applications. However, it is also demonstrated in *Case 1* that it is possible to resolve resource usage and timing statically during compile-time without costly run-time mechanisms, but this is not yet common in commercial mature technologies. This might actually be one of the reasons for the limited usage of CBSE in the domain today. Furthermore, *Case 4* demonstrated a method supporting the adaptation needs, which got positive feedback to address the specialization problem, but it has not been used in real projects.

10.6 Conclusions and Future Work

In this paper we have reported experiences from four cases where we have introduced/demonstrated CBSE principles at CC Systems.

Overall our findings indicate that CBSE principles are suitable for embedded systems sub-contractors, but also that it might be harder to practice CBSE as sub-contractor than product owner. The most technical needs of expressiveness in the component models, resource efficiency of component based applications, and analysis possibilities can be considered possible to fulfil with a combination of the contents in the different cases. According to our studies the most important need is related to resource efficiency. Resource efficient component frameworks with mature tools together with support for adaptation of software components themselves are needed.

For future work it would be interesting to explore more about the impact from the business situation on CBSE. In the domain of control systems for vehicles and machines we can identify three major business situations *sub-suppliers on contract basis*, *COTS suppliers*, and *product owners*. Note that it might be possible to study all these within a single company, as e.g., CC Systems, hopefully with increased possibilities to limit influences from other differences. Different goals with practicing CBSE would also be interesting to explore in combination with the different business models.

10.7 Acknowledgements

This work was partially supported by the Swedish Foundation for Strategic Research (SSF) via PROGRESS, Save, and Save-IT.

Bibliography

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

[2] G. T. Heineman and W. T. Councill. *Component-based Software Engineering, Putting the Pieces Together*. Prentice-Hall, 2001. ISBN: 0-201-70485-4.

[3] Rob van Ommering, Frank van der Linden, Kramer Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, march 2000.

[4] Kurt-Lennart Lundbäck, John Lundbäck, and Mats Lindberg. Development of dependable real-time applications. Arcticus Systems, December 2004.

[5] International Electrotechnical Commission IEC. *International Standard IEC 61131, Programmable controllers*, 1992.

[6] Ivica Crnkovic. Component-based approach for embedded systems. In *9th International Workshop on Component-Oriented Programming*, Oslo, June 2004.

[7] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. SaveCCM – a component model for safety-critical real-time systems. In *Proc. 30th Euromicro Conference*, pages 627–635, 2004.

[8] International Electrotechnical Commission IEC. Standard: IEC61508, Functional Safety of Electrical/Electronic Programmable Safety Related Systems. Technical report.

[9] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The save approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[10] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[11] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *In Proceedings of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*. LNCS Springer, 2003.

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

[13] CC Systems. Crosstalk generic control system platform. Technical report, CC Systems, 2007.

[14] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances, ICSEA'06*. IEEE, October 2006.

[15] Q. Tien Le. Component design tool for embedded system components. Technical report, Masters Thesis, MRTC, Mälardalen Univ., 2008.

[16] Mikael Åkerholm, Joakim Fröberg, Kristian Sandström, and Ivica Crnkovic. A model for reuse and optimization of embedded software components. In *29th International Conference on Information technology Interfaces, (ITI 2007)*. IEEE, June 2007.

[17] A. Orso, M. J. Harrold, D Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontents to support the regression testing of component-based software. In *Proceedings of the International Conference on Software Maintenance*, November 2001.

[18] G. T. Heineman. An evaluation of component adaptation techniques. In *2nd ICSE Workshop on Component-Based Software Engineering*, 1999.

[19] K. Cooper, J. Zhou, H. Ma, I. L. Yen, and F. Bastani. Code parameterization for satisfaction of qos requirements in embedded software. In

Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, 2003.

[20] J. Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 5(41), 1999.

[21] J. Brant, B. Foote, R. e. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings of 12th European Confernece on Object-Oriented Programming (ECOOP98)*, July 1998.

[22] D. M. Yellin and R. E. Strom. Protocol specification and component adaptors. *ACM Trans. on Programming Languages and Systems*, 2(19):292–333, March 1997.

[23] K.-K. Lau, L. Ling, and Z. Wang. Composing components in design phase using exogenous connectors. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2006.

[24] SEI. CMMI for development, version 1.2. Technical report, Technical Report CMU/SEI-2006-TR-008, 2006, 2006.