Mälardalen University Licentiate Thesis
No.44

A Software Component Technology for Vehicle Control Systems

Mikael Åkerholm

February 2005

MÄLARDALEN UNIVERSITY

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 91-88834-92-1
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Software is fantastic! Numereous modern high-tech products incorporate software. This thesis focus on software for vehicles. As high-tech products, modern vehicles contain much software that provides advanced functionality, e.g., efficient engine control, anti-spin systems, and adaptive-cruise control.

However, software engineering is not without problems! Software can contain errors, is often delivered later than promised and the cost of its development constitutes a major part of the total development cost of the vehicle. These problems are consequences of the complexity of the systems we build with software, in combination with the immaturity of an evolving discipline.

We therefore need improved approaches to software engineering. Component-based software engineering is a promising approach. It is analogous with approaches used 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). It has proven to be effective in application domains such as desktop and web-applications. However, it has not yet been widely adopted for use in the development of vehicular software; one of the reasons being that the commercial component technologies are developed specifically for other domains and to support other types of applications.

This research aims at developing a component technology for embedded control systems in vehicles. Such a technology would enable software engineers in the vehicular domain to make use of component-based software engineering. We have addressed questions concerning quality attributes, and how component-based applications should be built and modelled, in the context of vehicular systems. Furthermore, based on our answers we have implemented, and evaluated a prototype component technology in cooperation with industry. The results confirm the suitability of our prototype, but also show that it must be further developed if it is to meet wider industrial requirements.

Till Jenny och Lucas

Preface

I have learned much during my two years as Ph.D. student, but most important during the journey has been all workmates that has made it such a good and fun time. Thank you all! That include all personnel at the Department of Computer Science and Engineering, in the SAVE project, and people at companies I have been in contact with.

I especially want to thank my supervisors Prof. Ivica Crnkovic and Dr. Kristian Sandström. I have really appreciated to work with you, you are the best! I also want thank Prof. Hans Hansson, Dr. Mikael Nolin, and Prof. Christer Norström for your advices and invaluable guidance during the time.

This work had not been possible without all fruitful cooperation and discussions with my fellow Ph.D. students. The closest cooperation has been with Johan Fredriksson and Anders Möller. Thank you both!

Thanks also to Jörgen Hansson and Ken Lindfors at CC Systems for inviting us to test our ideas. To Johan Strandberg and Fredrik Löwenhielm at CC Systems for all time and energy spent on all our technical questions. Among all helpful people I have met at different companies I want to thank Joakim Fröberg, Jakob Axelsson, Mattias Ekman, Ola Larses, Bo Neidenström, and Bertil Emmertz, for your time discussing my questions.

Finally, I want to thank my mother and father, relatives and friends, for all your love and support.

Mikael Åkerholm
Västerås, January, 2005

List of Publications

Reports included in the thesis

Conferences and Workshops

Paper A Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin, *Evaluation of Component Technologies with Respect to Industrial Requirements*, In Euromicro Conference, Component-Based Software Engineering Track Rennes, France, August 2004.

This paper is an evaluation of the suitability of exiting component technologies for vehicular systems. The evaluation is based on a literature survey of existing component technologies for embedded systems, and a study based on interviews capturing requirements on component technologies from the vehicle industry.

Mikael's part of the work has been to provide knowledge about the component technologies and participate in the evaluation process. He has not participated in collection of the industrial requirements, but he has adopted and used the requirements in this work.

Paper B Mikael Åkerholm, Johan Fredriksson, Kristian Sandström, and Ivica 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.

This paper is based on a survey were representatives from different vehicular companies have placed priorities on a list of quality attributes. The paper presents the results of the survey, and in addition, a discussion of how the quality attributes could be supported by a component technology.

Mikael has been involved in all parts of the work in this paper; he has initiated the study and led the process.

Paper C Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren, *SaveCCM - a component model for safety-critical real-time systems*, In Euromicro Conference, Special Session Component Models for Dependable Systems Rennes, France, September 2004.

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.

This paper is based on discussions within a group of researchers', even larger than the set of authors. Mikael has participated in the discussions, and in the writing process.

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

In this paper we show how to use component based software engineering for low footprint embedded systems. The key concept is to provide expressive design time models and yet resource effective run-time models by statically resolve resource usage and timing by powerful compile-time techniques.

The compile-time method is based on earlier research, in particular experiences from Kristian's (first author) previous work. Mikael has been involved in all parts of the work, with focus on the component model and compile-time step.

Paper E Mikael Åkerholm, Anders Möller, Hans Hansson, and Mikael Nolin, *Towards a Dependable Component Technology for Embedded System Applications*, In Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS2005), Sedona, Arizona, february, 2005

This paper describes a prototype component technology developed for control applications in vehicles. The component technology has been evaluated with an application in cooperation with industry.

Mikael's main contribution to this work is implementation of the component model, and the compile-time allocation to the operating systems.

There has been an equal amount of efforts between Anders and Mikael in implementation of the test application, and evaluation of the result.

Other reports, not included in the thesis

Conferences and Workshops

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

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

- Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin, *Software Component Technologies for Real-Time Systems - An Industrial Perspective*, In WiP Session of Real-Time Systems Symposium (RTSS) Cancun, Mexico, December 2003.

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

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

- Ivica Crnkovic, Igor Cavrak, Johan Fredriksson, Rikard Land, Mario Zagar, and Mikael Åkerholm, *On the Teaching of Distributed Software Development*, In 25th International Conference Information Technology Interfaces Dubrovnik, Croatia, June 2003.

Technical Reports

- Mikael Åkerholm, Anders Möller, Hans Hansson, and Mikael Nolin, *SaveComp - a Dependable Component Technology for Embedded Systems Software*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-165/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, December 2004.

- Mikael Åkerholm, Kristian Sandström, and Johan Fredriksson, *Interference Control for Integration of Vehicular Software Components*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-162/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May 2004.

- Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin, *An Industrial Evaluation of Component Technologies for Embedded-Systems*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February 2004.

- Mikael Nolin, Johan Fredriksson, Jerker Hammarberg, Joel G Huselius, John Håkansson, Annika Karlsson, Ola Larses, Markus Lindgren, Goran Mustapic, Anders Möller, Thomas Nolte, Jonas Norberg, Dag Nyström, Aleksandra Tesanovic, and Mikael Åkerholm, *Component Based Software Engineering for Embedded Systems - A literature survey*, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-102/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June 2003.

- Ivica Crnkovic, Goran Mustapic, and Mikael Åkerholm, *Modern technologies for modeling and development of process information systems*, MRTC Report ISSN 1404 - 3041 ISRN MDH - MRTC - 100/2003 - 1 - SE, Mälardalen Real-Time Research Centre, Mälardalen University, May 2003.

- Mikael Åkerholm, and Johan Fredriksson, *A Sample of Component Technologies for Embedded Systems*, Technical Report, Mälardalen University, November 2004.

Contents

I

Thesis

Chapter 1

Introduction

Building systems from components is an old engineering practice. Cars have been assembled from components since Henry Ford built the first T-Ford at the beginning of the 20th century. The component-based assembly of computers can be considered to have been begun in the 1950's when several transistors were first integrated in a single circuit. Software itself has also been built for many years using a component-based strategy. The development of complex programs has been made possible through the development of sub-programs, in a system decomposition process, e.g., client-server, and object-oriented design [1]. However, with a few exceptions e.g., mathematical and graphical function-libraries, the component-based strategy has, historically, not been as successful in the software industry, as in the examples from other industries, previously mentioned. This is, perhaps, mostly because attempts to reuse software components have resulted in serious problems due to architectural mismatches between components or components and the surrounding environment [2]. Before software engineering can be a mature, well founded discipline, these issues must be resolved. As stated by Szyperski [3]:

> *Building, using and reusing components is what all other engineering disciplines have done when they have become mature, the software discipline should also follow.*
>
> Szyperski

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 [4]. 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. 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 [5] and .NET [6] from Microsoft, and Enterprise JavaBeans [7] from Sun Microsystems.

CBSE has been successfully used in several software development projects, mainly in the domains of desktop and e-business applications, less frequently in embedded applications. This thesis addresses the problem of defining a component technology suitable for the development of software for embedded control-systems in vehicles. The underlying assumption is that one reason for the limited success of CBSE practice in the embedded systems domain has been the inability of commercially available component technologies to provide solutions that meet typical requirements of embedded applications e.g., resource-efficiency (the consumption of a minimum of resources in achieving an objective), predictability, and safety. In brief, we have addressed the problem through development of a prototype component technology for embedded vehicular systems, and the suitability of the result has been evaluated by means of an experiment in cooperation with industry.

The work has been carried out within the SAVE project [8]. The main goal of the project is to begin establishing an engineering discipline for systematic development of component-based software for safety-critical embedded systems. The focus of the project is on a single application area (vehicular systems), to reduce the overall project complexity to a manageable level. The component-technology presented in this thesis can be seen as one of the core parts of the project, other results from the project being integrated in future work.

The two sections following immediately provide introductions to basic technical concepts of component technologies, and vehicular systems, as a foundation for reading the rest of the thesis.

1.1 Component Technology Terminology

This section provides a brief introduction to central technical concepts of component technologies frequently used in the remainder of the thesis. For more information about CBSE and component technologies refer to, e.g., Heineman and Councill's book [9], Szypeki's book [3], or Crnkovic and Larsson's book [4].

Fundamental to CBSE is that software applications are built from components. The components are to be *composed* (or *assembled*) into applications, i.e., combining them to give the desired behaviour. A component technology provides support for the composition of component-based software. It 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 1.1 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. This playground table will be used as a metaphor for a component technology in the following description of technical concepts.

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 [7]. 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 a set of rules to be followed by users. It defines different component types that are supported by the technology, possible interaction schemes between components, and clarifies how different resources

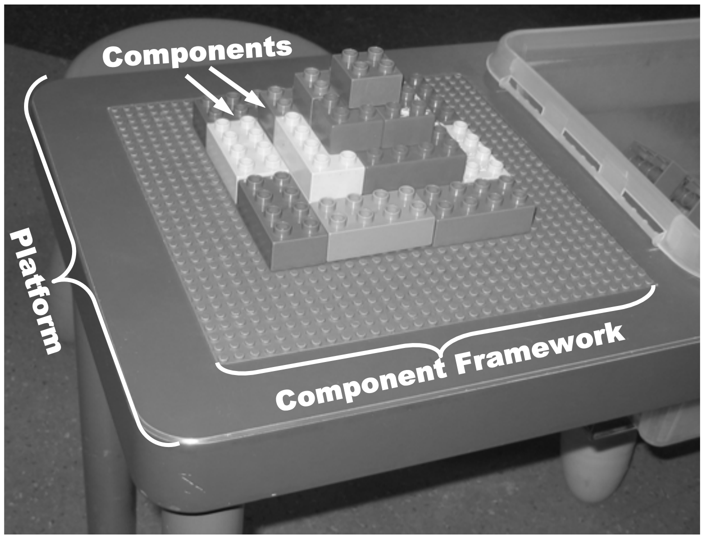


Figure 1.1: A Component Technology for Building Arbitrary Shapes

are bound to components. Compliance with a component model distinguishes a component from other forms of packaged software [10]. In our playground example, the component model is represented by the abstract rules that the children must follow when assembling blocks because the blocks can only be assembled in a certain pattern. The supplier of the blocks also follows 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.

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 [11]. 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 [3]. 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 tries to be consistent with the majority of these [9]:

> *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. Components should have well-specified interfaces, since CBSE relies heavily on interfaces. 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.

1.2 Vehicle Electronic Systems

The application domain in this thesis is embedded control systems in modern vehicles, e.g., in passenger cars, trucks, and heavy vehicles. Modern vehicles contain electronics that can be classified as follows [12]:

Power train and chassis systems. These include systems that are highly critical for the vehicles functionality, controlling, e.g., engine, brakes, and steering. They are characterised by high demands on safety, reliability, and hard real-time constraints.

Cabin systems. Less critical systems, that also are core parts of the functionality, e.g., dashboard instruments, electrically powered windows, and air-conditioning.

Infotainment systems. Systems dedicated to information processing, entertainment, and communication with others outside the vehicle, e.g., audio, video, and satellite-navigation systems. These systems are not closely integrated with the core functionality of the vehicle and are easier to replace, supplement and remove.

We concentrate on the power train and chassis, and use the common term *vehicular systems*, when referring to these systems in different classes of vehicles, e.g., cars, and heavy vehicles. The focus is chosen because 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 qualities that are not addressed by existing commercial component technologies (e.g., .NET [6], and Enterprise JavaBeans [7]), and consequently cannot be developed with the existing commercial component technologies. We observe however that the existing technologies might be well-suited in the development of infotainment applications, which are similar to desktop- and web-applications for personal computers.

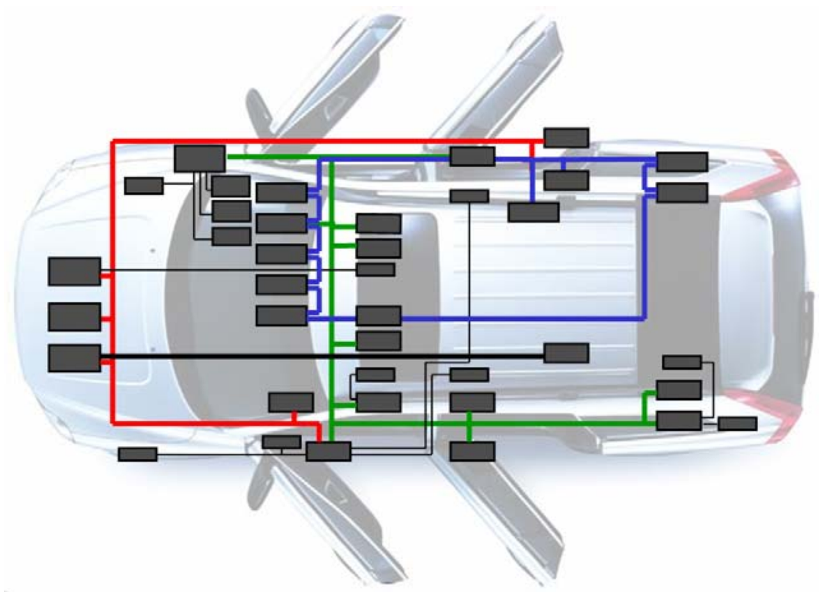


Figure 1.2: Overview of the electronic system architecture in Volvo XC90

The physical architecture of the electronic system in vehicles is a complex distributed computer system. The computer nodes are designated Electronic Control Units (ECUs), and are often developed by different vendors and use different hardware. 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 [13]. In figure 1.2 from [13], the electronic architecture of a Volvo XC90 is shown. The figure shows the approximate location of the forty ECUs in the vehicle, and to which networks each ECU is connected. The location of each ECU is primarily determined by where the controlled object is in the vehicle to minimize the length of wiring to sensors and actuators, e.g., the engine controller is placed close to the engine. Interconnection networks permit the implementation of functions through collaboration between several ECUs, e.g., the Electronic Stability Program (ESP) developed by Mercedes-Benz [14]. ESP utilises ECUs controlling the engine and brakes, to assist the driver in situations in which the vehicle skids.

1.3 Outline of Thesis

The remainder of Part I is outlined as follows. Chapter 2 is a summary of the research. It contains research questions, and summarizes the contributions of the published papers constituting this licentiate thesis. In Chapter 3, related work is summarized. Finally, Chapter 4 contains conclusions and discusses possibilities for future work. Part II, contains the papers included in Chapter 5 to Chapter 9.

Chapter 2

Research Summary

This chapter contains a summary of the research methods, questions, and contributions relevant to the thesis. The research methodology is described in Section 2.1. Section 2.2 presents the general picture, i.e., the real-world problem that the contribution of this thesis helps to solve. Section 2.3 presents the focus of the thesis, in terms of research questions that define the research setting. Finally, Section 2.4 presents the contributions of the thesis, how the results have been obtained, and how they have been evaluated.

2.1 Methodology

We have adopted the research methodology described by Shaw in [15]. The methodology is derived from experience of more than a decade of the actual performance of software architecture research. The main activities are:

1. 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 2.2.

2. Transfer the problem to a research setting, and define research questions. The research setting is a simplification 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. The details are discussed in Section 2.3.

3. Answering the research questions. In this phase, the work is aimed at 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 2.4 describes the methods and the answers obtained in this thesis.

4. The research results are validated 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 describing experiences, and by persuasion through argumentation. Validation is also addressed in Section 2.4.

2.2 Problem Definition

This thesis addresses the problem of defining a component technology for embedded control systems in vehicles. There are several challenges related to the problem, in an area relatively unexplored in comparison with component technologies for desktop- and web-applications.

The existing commercial component technologies have been developed within the PC domain, which is a domain with requirements fundamentally different from those of vehicular systems. In comparison with PC software, the safety requirement for vehicular software is much greater. Unsafe behaviour of a moving vehicle can result in injury, even death, to persons. The demand for software reliability is greater since failure could result in financial loss or loss of goodwill. A minimum of resource consumption is required to minimize vehicle production-cost. Vehicles are often produced in large numbers, in the case of cars, often of the order of millions per year. Finally, vehicular systems are real-time systems, due to temporal requirements based on the vehicle functions. Consider the software for an Anti-lock Brake System (ABS). When the brake on one wheel locks, the system must release the brake pressure within a certain time for the driver to remain in control of the steering. On the other hand, certain systems cannot respond too soon. When a collision occurs, the software controlling an air-bag must be activated within a certain time-interval; it is useless if it is activated too late or too early.

The overall industrial problem is to define a component technology that provides methods to accomplish the main application requirements as discussed above, and also supports typical engineering activities in the vehicular domain. Wolf [16] defines the primary objectives for engineers in the embedded systems domain, in which vehicular systems are a specialisation, with the subjects: architecture, analysis, modelling, verification, and application orientation. Some of these subjects are involved in all software engineering, but the approach to them and their purposes distinguishes the software in embedded systems from other classes of software. Each subject might in turn have a different importance and slightly different meaning in different domains of embedded systems.

System Architecture. Architectures for embedded systems are defined to serve the functions of a particular application using resources efficiently. This calls for specialised component technologies which use resources efficiently, to meet the demands of the vehicular domain, not for a broad range of (embedded) applications [17]. This argument is also supported by Larn and Vickers when they describe the risk of performance compromises in the architecture, due to support for reusability of general components [18]. Furthermore, Crnkovic identifies key priorities in research to establish a CBSE engineering discipline for embedded systems. One of these is the development of component run-time frameworks which require a minimum of resources [19].

Analysis. A suitable component technology should provide support for reasoning about such quality attributes as the performance and size of systems at an early stage in the design process. This is a goal the designers of embedded systems strive to achieve. The need for predictable component technologies in the automotive domain is stressed in, e.g., [20, 19].

Modelling. To simplify analysis and understanding of the system, developers need models of different aspects of applications at a higher abstraction level than the source code provides. For instance, architectural models for analysing and understanding basic functionality, timing models for real-time analysis, and behaviour models of the system in its intended environment for safety analysis. In a component-based approach, the description of the component assembly serves as an architectural model of the application. The architectural model can be supplemented with other modelling aspects, or with additional models. Möller et al., have obtained further requirements of the component modelling language from industry, observing that it should follow commercially supported standards [20].

Verification. Applications must be verified in accordance with functional and non-functional specifications. For a component technology, predictability to enable the use of formal methods, complemented by practically oriented simulation and test support are needed to meet these constraints. Much research performed by the CBSE community addresses the prediction of different quality attributes of component based applications, e.g., [21, 22, 23, 24, 25].

Application orientation. The control of the vehicle dynamics is not the only functionality to be considered. The control-related part of the application is its core but is in fact only a small part of the whole vehicle application. The component technology must provide support for more than the implementation of control systems. Typical functionality that coexists in the same environment includes fault tracing and handling, monitoring and logging for diagnostics during service, and human interaction through displays and controls. There is an increasing number of infotainment applications in modern vehicles, but this functionality often remains separate from the vehicle control system.

2.3 Research Questions

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 the research questions. The research is based on two fundamental assumptions that motivated the thesis:

1. The use of CBSE in the domain is limited by the inability of existing commercial technologies to support the requirements of embedded vehicular applications.

2. Despite the different demands on embedded vehicular software as compared with other software, CBSE is an attractive engineering approach.

As noted in the preceding section, commercially available technologies are not suitable for the vehicular domain, which is a logical reason for the companies to approach CBSE with caution.

The validity of the second concept is demonstrated however by the efforts the vehicular industry is actually making in CBSE related projects, e.g., EAST [26], and AUTOSAR [27]. The amount of functionality implemented with software in vehicles continues to increase leading to increased software complexity. Combined with requirements for lower costs and shorter development time (as in all industrial segments), there is a need for better systematic engineering approaches. CBSE is one candidate approach but, however, for it to be utilised efficiently, a mature component technology with good domain-specific support is needed.

With these assumptions in mind, the main research question (Q) is formulated as follows:

How can a component technology for vehicular software be implemented, to provide the functionality desired for vehicular applications, with support for the important quality attributes essential in the vehicular domain?

(Q)

The main question is broad, and admits the possibility of different answers. We intend to explore one of the possibilities, but to do this thoroughly by answering separately a number of sub-questions addressing component models, frameworks, and quality attributes. The sub-questions are formulated:

Which quality attributes are the most important in the vehicular domain, and how do they relate to a component technology?

$(Q1)$

Quality attributes define the quality of software. Quality attributes for the evaluation of software have been standardised in [28]. The top-level attributes in the standard are functionality, reliability, usability, efficiency, maintainability, and portability. This sub-question seeks to establish domain-specific priorities of different quality attributes, and it is reasonable to expect that attributes can have different priorities in different software domains. The second part of the question addresses the relations between a component technology and the quality attributes that are important in the domain. We expect that some quality attributes are more dependent on the choice of component technology than others. Furthermore, it is known that quality attributes are interdependent, and that some contradictoty attributes are difficult to support simultaneously [29, 30]. Therefore, the establishment of domain-specific attribute priorities can be used as guidance when resolving conflicts.

How can a component model support predictability of important quality at-

tributes and be suitable for expressing common functionality in vehicular control systems?

$$(Q2)$$

This question calls for an approach by defining components and possibilities for component interaction, with respect to ease of implementing vehicular control systems, and support for prediction of quality attributes considered important in the domain (identified in $(Q1)$). The general approach to increasing the predictability of software is to limit its flexibility. For example, pre-run-time scheduling limits the arrangement of activities to a pre-defined schema. The execution path for such systems is easier to predict than that of flexible run-time scheduled systems. However, flexibility increases the possibilities for engineers when implementing applications. This question seeks the trade-off between the predictability desirable and the flexibility necessary when defining applications in the context of vehicular control systems.

How can an efficient utilisation of resources be achieved in component based applications?

$$(Q3)$$

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 has received special attention in this work. This focus is based on the belief that poor resource efficiency is an important reason for vehicular companies not choosing to utilize commercial component technologies. Commercial 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 [31], while typical hardware used in vehicular systems can have 128 kB [32].

2.4 Contributions

To address the above questions, we have conducted literature surveys, developed new methods, implemented a prototype for evaluation, and also utilised experience from earlier work by senior researchers. The work has also included industrial involvement. Interviews with technical staff regarding current system engineering methods were carried out early in the project as orientation for the initial research steps. Representatives of different vehicular companies have been consulted regarding the priority given to software quality attributes. Finally, validation of the early results has been performed in cooperation with industry.

The contributions and validation of the research are presented with the questions as starting points. The results provide answers to the research questions, but are subject to certain limitations, which are also discussed and are suitable for further research. There are more detailed presentations of the contributions in the five papers referred to in the presentation.

Contributions answering Q

How can a component technology for vehicular software be implemented, to provide the functionality desired for vehicular applications, with support for the important quality attributes essential in the vehicular domain?

Paper A addresses the main question by evaluating existing component technologies intended for embedded systems. The evaluation is based on a literature survey of different component technologies [33]. The literature survey is compared with a study based on interviews to determine the requirements of the vehicle industry for component technologies [20]. One of our conclusions is that none of the technologies studied completely satisfies the industry requirements. Furthermore, no single technology stands out as being a significantly better choice than the others; each technology has its own pros and cons. However, worth notice is that two of the technologies investigated (i.e. Koala [34], and Rubus CM [35]) are actually used successfully in practice by industry without meeting all the requirements specified in the evaluation.

The prototype component technology described and evaluated in Paper E, shows how far we have reached in finding an answer to the main question. The prototype is based on the component model described in Paper C, and compile-time techniques from Paper D. The component technology was developed for control applications in vehicles, providing both support for predictability of quality attributes and typical functionality in vehicular control applications. The component technology has been evaluated with an application in cooperation with industry. The conclusion from the evaluation is that our initial study provides positive evidence, but that the technology needs to be further developed to be applicable in an industrial context.

There are several assumptions and limitations in the answer that need to be addressed in future work. The suggested component technology is a promising prototype including only the core functionality. It needs to be extended with supporting tools for modelling and configuration management. Furthermore, a vehicular function is often distributed across several interconnected physical nodes. Support for distribution of components would therefore be a valuable extension of the prototype. The question also addresses support for important quality attributes. This support has not been fully implemented and evaluated, but an initial suggestion is described in Paper B.

Contributions answering $Q1$

Which quality attributes are the most important in the vehicular domain, and how do they relate to a component technology?

This question is the main topic of Paper B. The paper is based on a survey in which representatives of different vehicle companies have allocated priorities in a list of quality attributes considered to be of particular interest to them. In order to reduce the number of attributes, those with obvious similarities in their definitions have been grouped in more generic types of properties; e.g., portability and scalability are considered to be covered by maintainability. The result shows that the companies involved give approximately the same priorities to quality attributes. Reasonably, the most important concerns are related to dependability characteristics, e.g., safety, and reliability. Usability is a property important to the customers but also crucial in competition on the market. Slightly less important attributes are related to the product life cycle (extendibility, maintainability).

The second part of the paper continues with a discussion relating quality attributes to a component technology, and a suggestion of quality attribute support in a component technology suitable for the vehicle domain. In addition, it discusses where in the technology the support should be implemented: inside or outside the components, in the component framework, on the system architecture level, or if the quality attributes are usage-dependent.

There are several questions respecting the validity of the study. Is the list of quality attributes that we have given to the companies appropriate? Have we captured the companies view or the representatives view? Is it a coincidence that we obtained similar priorities from the different companies in the study, or is it a common opinion? Even if such questions are motivated, the research results put focus on the main concerns. They can be used as initial guidance for further development of component technologies suitable for vehicular systems. To continue the investigation and resolve certain questions, a future workshop could be organized at which the representatives interviewed and possibly other stakeholders from industry could discuss and expand upon the results.

Contributions answering $Q2$

How can a component model support predictability of important quality attributes in the domain, and be suitable for expressing common functionality in vehicular control systems?

Paper C 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 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 a set of ports, which are points of interaction between the elements. We distinguish between input- and output-ports, and the complementary aspects data transfer (data-flow) and execution triggering (control-flow).

The component model has been designed to easily express common functionality in vehicular systems. Some specific examples of key functionality are: feedback control, system mode changes, and static configuration for product-line architectures.

The component model is limited to simplify future analysis of different properties, e.g., real-time, reliability, and safety properties. It may, however, be too restrictive, and should be expanded. Paper E contains an evaluation of our prototype implementation of a component technology using the SaveCCM component model, which we found to be sufficiently expressive. We studied only a single application, and are not able to generalise and claim that it is a representative case. If future studies indicate that we must extend SaveCCM, this will be done, maintaining predictability as far as possible.

Contributions answering $Q3$

How can resource efficiency of component-based applications be achieved?

This question is mainly treated in Paper D. In commercial component technologies the component-based approach is made possible by powerful run-time mechanisms, which for resource-limited systems have the disadvantage of increased resource utilisation. In this paper, we show how a component-based approach is permitted by the use of powerful compile-time techniques. The key concept is the clear distinctions between design-time, compile-time, and run-time. The method allows both expressive design-time models and resource-effective run-time models by statically resolving resource usage and timing during compile-time. This results in a component technology with resource-effective mapping of a component model to a commercial real-time operating system. The evaluation of the prototype technology in Paper E proves that the compile-time methods are able to generate resource-efficient systems from component-based designs.

The strategy is to resolve as much as possible during compile-time and to utilise resource-efficient platforms during execution. To further improve resource-efficiency, the compile-time mapping can be further optimised; 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.

Chapter 3

Related work

Recently component technologies for different classes of embedded systems have been developed both in academia and in industry. Some of them are discussed in Section 3.1, together with related projects within the different domains. Section 3.2 addresses CBSE research not targeting particular domains, but that are important ingredients to enable CBSE for embedded systems. We review basic research regarding predictability of quality attributes of component-based systems, and some efforts to enable use of existing commercial component technologies in embedded systems.

3.1 Component Technologies in Different Domains

In this section some examples of component technologies from different domains of embedded systems are discussed. We discuss vehicular systems, consumer electronics, and automation systems, in separate subsections.

3.1.1 Vehicular Systems

A component technology for vehicular software must focus on safety and dependability. Software engineers in the vehicular domain create software for, e.g., cars, trains, trucks, and heavy vehicles. These software systems have much in common, but also some differences. They have high demands on safety and dependability, since moving vehicles can be dangerous. The embedded computer systems often consist of several tens of interconnected Electronic Control Units (ECUs). The ECUs are either developed in-house or by sub-contractors, and often has to cooperate in order to solve a task, e.g., an anti-spin system in a car often consists of collaborating ECUs at each wheel. Therefore coping with interconnection networks is also a basic requirement for the software. One of the biggest differences with the software for different vehicles comes from the production volume of the intended vehicle. Cars are typically produced in the order of millions per year, and thus it is beneficial to spend large engineering efforts that lower the production cost, e.g., optimising the code for smaller memory consumption can enable the use of a smaller and cheaper memory. While heavy vehicles as wheel-loaders are produced in smaller quantities, consequently it is not cost-efficient to spend the same efforts on optimisation.

The Rubus Component technology is a commercial technology that is used for heavy vehicles. It is shipped, and tightly integrated, with the Rubus operating system [35, 36]. Rubus is not only tailored for automotive systems, it addresses resource limited systems with real-time requirements. The purpose and main objective with the component technology is to support reuse, and product line management. A component consists of one or more tasks, which are the run-time entities executed, by the operating system. Aggregate components can also be specified, called composite which is a logical design-time composition of one or more components. Interaction between components is achieved by connecting data ports, forming a control-flow (pipes and filters) pattern. Rubus components are statically scheduled, and sophisticated timing requirements can be specified, i.e, release-time, deadline, worst-case execution-time and period-time. The main limitation is that only periodic activation is possible with the static scheduling approach.

Within the vehicle domain there is one major project focusing on software components. AUTOSAR [27] is a global project with some of the major actors; it can be seen as an extension of EAST-EAA [26]. It is a standardization project with the goal to provide an open standard component technology for automotive electronic equipment.

3.1.2 Consumer Electronics

Product line management is important in a component technology for consumer electronics. Consumer electronics software is software embedded in, e.g., television equipment, audio equipment, and kitchen appliances. These products are usually produced in very high volumes, and the customers are price- and design-sensitive. Therefore production cost, and time to develop new products that meet new trends are important. To cope with this, product families with small differences in functionality are common. The software is often organised in a base-line variant, which is configured or extended to rapidly provide new products from the same product family. The requirements on safety and reliability are not as high as for vehicular systems.

Koala [37] is a component technology developed and used by Philips. The Koala components are encapsulated units, with all interaction between its interfaces. There are separate interfaces for which services a component provide and which services it requires from its surrounding. Furthermore, components can have multiple interfaces, which is a way for handling evolution. Components interact through their interfaces, using a client-server pattern. Aggregation of components is also possible. Component binding flexibility can be achieved with switches. A switch chooses between interfaces offered by different components at run time, with possible static reduction at compile-time.

Space4U [38] is a continuation of the ROBOCOP project [39], which in turn was a continuation to enhance the Koala model. The major actors are Philips Electronics, Technical University of Eindhoven, and Nokia. The goal of the ROBOCOP project was to develop a middleware-architecture for high volume consumer electronics, supporting robust, reliable and manageable use of software components. A main extension to Koala was the introduction of models associated with components. Each component has a set of models, where each model provides different information about the component. The models may be in different forms, e.g., textual, or binary form. They may model functional and non-functional properties of a component, e.g., functionality of the component, timing, reliability, and memory usage. The following active project Space4U has the goal to complete and extend the ROBOCOP prototype with, e.g., fault prevention, power management and terminal management aspects.

3.1.3 Automation Systems

The diversity of systems within the automation industry will most likely require several different component technologies. The automation industry includes a broad range of software, from software in small field devices to operator stations. The diversity of the requirements for the software in different applications is wide. The automation industry deal with software that is very safety critical and has high demands on temporal correctness and reliability, e.g., software for controlling temperature and heat in sensitive chemical processes. It also includes software with focus on usability rather than safety or dependability, e.g., production planning applications. The production volume of automation equipment is also varying, from customized site specific software, to more general purpose.

IEC 61131-3 [40], is a set of standard programming languages for Programmable Logic Controllers (PLCs). The standard defines three graphical and two text-based languages. IEC 61131-3 concentrates on the syntax, but leaves the semantics less definitive. The graphical Function Block Diagram (FBD) language can be treated as a component composition language. Components (control blocks) have a set of in and out data-ports as external interface, and a hidden internal implementation. System implementation is done by connecting in and output ports, forming a control flow (pipes and filters) pattern. Components can also be aggregated from sub-components. The component framework is not standardized; it is assumed that components are allocated to threads scheduled by an assumed operating system. There are some restrictions of the underlying operating system like no pre-emption of tasks.

Port Based Objects (PBO) [41] is a component technology specialised on reconfigurable robotics applications, from the Advanced Manipulators Laboratory at Carnegie Mellon University. A component is defined as an object, with an arbitrary number of input, output, and resource ports. Input and output ports are used for communication between objects, while resource ports are aimed for communication with sensors, actuators or other external devices or subsystems. The component model uses the control flow (pipe and filter) pattern. It is aimed for multiprocessor systems, to support distribution the connections are implemented as global variables. The technology is focused on control applications, and has support for modelling output response from given inputs of closed or open loop systems by applying transfer functions. Timing analysis and analysis of the state variable communication are also supported. The PBO model uses the Chimera multiprocessor operating system [42] during run-time.

PECOS [43] was a collaborative project between ABB and academia. The goal was to develop a component technology for field devices. The project tried to consider non-functional properties very thoroughly in order to enable assessment of the properties during construction time. Components have data ports for interactions, and connecting them forms a control-flow (pipes and filters) pattern. Besides data ports, the components also have interfaces to express arbitrary non-functional properties and constraints. Components can be of passive-, active-, or event-type. Passive components does not have an own execution thread, they are only utilised by other types of components. Active components have their own thread that is periodically scheduled. Event components are components that are triggered by an external event and have a thread of control. Components can also be aggregated, these components are called composite while non-aggregated are called leaf-components. There is no special component framework in the PECOS project, too meet industrial requirements on platform independency, or at least portability.

3.2 Domain Independent CBSE Research

There are a lot of basic and domain independent research within the CBSE community. Here we focus on efforts especially interesting for embedded systems. Analysis and support for different quality attributes are discussed in section 3.2.1, and some efforts to enable the use of existing commercial component technologies in embedded systems are reviewed in Section 3.2.2.

3.2.1 Quality Attribute Prediction

In the CBSE community, a lot of research efforts address prediction of quality attributes of component-based applications. The efforts are motivated by the assumption that component-based systems are a suitable base for quality attribute prediction. The key idea is that quality attributes is easier to assign to components than to whole systems, and that components can provide sufficient information to reason about quality attributes of the system. In the ISO 9126 standard [28], six quality attributes (functionality, reliability, usability, efficiency, maintainability, and portability) are defined for evaluation of software quality, where each of these attributes can be divided into a number of sub-attributes.

Predictable Assembly from Certifiable Components (PACC) [44], is a project at the Software Engineering Institute (SEI). The project focuses on how a component technology can be extended or restricted to achieve predictable assembly from certifiable components, and one of the results is the Prediction-Enabled Component Technology (PECT) [24]. It is a development infrastructure that incorporates a component technology, development tools and analysis techniques. The idea is that any component technology can be used in the bottom but composition rules enforced by the development tools guarantee critical runtime properties, e.g., PECT enforces that predictable construction patterns are used. What is allowed for a user and what is required by the underlying component technology are determined by the available analysis methods and prediction goals.

In the Ph.D. thesis by Larsson [45], prediction of quality attributes is the main topic. In his work there is a division of quality attributes based on how they can be predicted on the system level. Directly composable attributes, are possible to analyze given the same quality attributes from the components. Architecture related attributes, are possible to analyze given this attribute for the components and the assembly architecture. Derived attributes, are possible to analyze from several attributes from the involved components. Usage dependent attributes, need a usage profile to enable analysis. System environment context dependent attributes, are only possible to analyze given environment attributes.

Reussner et al [22] shows how reliability can be calculated using Markov chains. One known problem in the use of Markov chains is the rapid growth of the chain and complexity. The component-based approach gives a solution, it permits a hierarchical approach. 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 [46], and memory consumption attributes in [47].

3.2.2 Adopting General Purpose Technologies

The most widely used component technologies are either intended for desktop applications (e.g., COM [5], and JavaBeans [48]), or for development of distributed large scale applications (e.g., CCM [49], EJB [7], and .NET [6]). However, certain efforts have been made to adopt and tune these types of technologies for embedded systems.

In Lüders licentiate thesis [50] COM has been adopted for parts of an industrial control system at ABB. However, it is mainly the component interface specification parts of COM that has been adopted. Furthermore, it is not a commercial implementation of COM that has been used, they have implemented an own custom run-time framework. The main conclusion from the study is that the efforts to implement new parts of the system have decreased from one third up to one half of the former efforts, after the introduction of a component-based approach. It has also resulted in a better documented and more modularised system. Concerning COM as a technique for industrial control systems, the study shows that it is possible. However, they have only used a sub-set of COM, and still emphasise the importance of careful application design to avoid overheads that cause variability. Furthermore, due to the run-time binding of interfaces to components, the study observes that COM introduces a potential source of run-time errors.

Sun Microsystems provide three different versions of the Java 2 platform, enterprise edition [31] for distributed large scale applications, standard edition [51] for desktop applications, and micro edition [52] for embedded devices. Enterprise JavaBeans is their component technology for distributed web-based applications provided for the J2EE platform. JavaBeans is their component technology for desktop applications, provided under the J2SE release. The most suitable Java version for embedded systems (J2ME) does on the other hand not contain any component technology. However, it might be possible to add support for their component technologies by installing extra packages and extensions, resulting in a customised version reminding of the services provided by J2SE or J2EE. The suitability of such a configuration is an interesting target for future evaluation.

Microsoft has two main platforms for embedded devices. Windows XP embedded [53] that is a scalable version of Windows XP, that allows users to omit certain parts of the operating system to obtain smaller applications. However, Windows CE [54] is Microsoft's operating system targeting real-time requirements, and in combination with the .NET component technology, it might be a good candidate for the types of applications considered in this thesis. Studies of earlier versions of Windows CE has shown that Windows CE cause unpredictable timing behaviour [55], even if real-time performance has been promised. Worth to notice is that the real-time performance has improved with new releases of the operating system, and it becomes more and more suitable for real-time applications.

CORBA Component Model (CCM) is standard from OMG, for a component model utilising the CORBA platform [49]. CCM as CORBA is intended for transparent interaction over networks in an object-oriented style. The goal is that objects (components in case of CCM) can be physically located at any node in the network, and still be accessed like if they were local objects at the caller's node. CCM and CORBA are not originally designed for real-time or embedded systems. However, OMG has two additional standards for these purposes. Minimum CORBA [56] is a standard that omits computation intensive dynamic features and focus on less resource consumption. Real-Time CORBA [57] include additions to standard CORBA for real-time performance. The problem is that they are separate standards, so network applications that are both resource limited and have real-time constraints would require a combination of both standards.

Chapter 4

Conclusions and Future Work

The overall contribution of this thesis is a prototype containing the core parts of a component technology for vehicular software. The technology incorporates a component model, which we propose to use together with compile-time techniques to achieve run-time efficient, and platform independent component-based applications.

The component model has been designed to easily express common functionality in vehicular systems. The component model is based on the 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 for product-line architectures. The component model is restrictive compared to most commercial component models intended for other types of software (e.g., desktop and distributed enterprise applications). The limited flexibility is motivated by the high requirements of analysability of safety, timing, and reliability properties for vehicular software systems.

The compile-time techniques enable a component-based approach during design-time, combined with resource effective run-time models of real-time operating systems, by statically resolve resource usage and timing during compilation. In contrast to commercial component technologies where the component-based approach is facilitated by powerful run-time mechanisms, which for resource limited systems has the disadvantage of increased resource utilisation. The key concept is clear distinctions between design-time, compile-time, and run-time.

The conducted evaluation of our prototype component technology in co-operation with industry indicates that the component technology is promising and efficient for vehicular applications. The expressiveness of the component model seemed to be sufficient for efficient practice of component-based principles. The evaluation also proved that the compile-time methods are able to generate resource efficient systems from component-based designs.

This work covers the core parts of our component technology. In future work, we will include support for important quality attributes in the domain, which also has been treated within this thesis. We have presented a survey, where vehicular companies have placed priorities on a list of quality attributes. The result shows that the most important concerns are related to dependability characteristics, e.g., safety, and reliability. Usability is a property important for the customers but also crucial in competition on the market. Slightly less important attributes are related to the product life cycle (e.g., extendibility, maintainability). Based on the results from the survey, we have discussed how quality attribute support in a component technology for the vehicular domain could be implemented in our future work.

In future work we will also integrate the technology with supporting tools (new or existing) for, e.g., graphical modelling and configuration management. New mechanisms like databases for structured handling of shared data, and run-time monitoring and test support are also targets for future work. Furthermore, we will also address the communication network in the vehicles, since much functionality in the vehicles are utilising the network. However, in all such extensions we will still prioritise predictability in the trade-off between more expressiveness, and keeping the predictability high.

Bibliography

[1] S. L. Pfleger. *Software Enginnering Theory and Practice*. Prentice Hall, 2001. ISBN 0-13-029049-1.

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

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

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

[6] 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.

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

[8] Save project. http://www.mrtc.mdh.se/SAVE/ (Last Accessed: 2005-01-18).

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

[10] 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.

[11] I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, editors. *Component-based Software engineering - CBSE 2004 Sympoisum*. Springer Verlag, 2004. LNCS 3054 2004-05-17 ISBN: 3-540-21998-6.

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

[13] 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.

[14] Mercedes-Benz. Mercedes-benz recent developments, safety. http://www.mercedes-benz.com/mbcom (Last Accessed: 2005-01-17).

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

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

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

[18] 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.

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

[20] 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.

[21] I. Crnkovic and M. Larsson. Classification of quality attributes for predictability in component-based systems. In *In DSN 2004 Workshop on Architecting Dependable Systems*, June 2004.

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

[23] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *Proceedings of the 1st international conference on Aspect-oriented software development*, 2002.

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

[25] G.A. Moreno, S.A. Hissam, and K.C. Wallnau. Statistical models for empirical component properties and assembly-level property predictions: Towards standard labeling. In *Proceedings of 5th Workshop on component based software engineering*, 2002.

[26] East, embedded electronic architecture project. http://www.east-eea.net/ (Last Accessed: 2005-01-18).

[27] Autosar project. http://www.autosar.org/ (Last Accessed: 2005-01-18).

[28] ISO/IEC. *Software engineering – Product quality – Part 1: Quality model, ISO/IEC 9126-1*, 2001.

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

[30] 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.

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

[32] Dag Nyström, Aleksandra Tesanovic, Christer Norström, Jörgen Hansson, and Nils-Erik Bånkestad. Data management issues in vehicle control systems: a case study. In *Euromicro Real-Time Conference 2002*, June 2002.

[33] M. Åkerholm and J. Fredriksson. A sample of component technologies for embedded systems. Technical report, MRTC, Mälardalen University, 2004.

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

[35] 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).

[36] K.L. Lundbäck. Rubus OS Reference Manual – General Concepts. Arcticus Systems: http://www.arcticus.se (Last Accessed: 2005-01-18).

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

[38] Space4u, software platform and component environment 4 you. http://www.extra.research.philips.com/euprojects/space4u/index.htm (Last Accessed: 2005-01-18).

[39] Robocop, robust open component based software architecture for configurable devices project. http://www.extra.research.philips.com/euprojects/robocop/index.htm (Last Accessed: 2005-01-18).

[40] IEC. *International Standard IEC 1131, Programmable controllers*, 1992.

[41] D.B. Stewart, R.A. Volpe, and P.K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, pages pages 759 – 776, December 1997.

[42] P.K. Khosla et al. The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications. *IEEE Transactions on Systems*, 1992. Man and Cybernetics.

[43] 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.

[44] Pacc project, predictable assembly from certified components. http://www.sei.cmu.edu/pacc (Last Accessed: 2005-01-18).

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

[46] 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.

[47] 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.

[48] P. Coffee, M. Morrison, R. Weems, and J. Leong. *How to Program Javabeans*. Ziff Davis, 1997. ISBN: 1562765213.

[49] J. Siegel. *CORBA 3 Fundamentals and Progamming, Second Edition*. John Wiley & Sons, Inc., 2000. ISBN: 0-471-29518-3.

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

[51] Sun Microsystems. Java 2 platform, standard edition (j2se). URL: http://java.sun.com/j2se/index.jsp (Last Accessed: 2005-01-17).

[52] Sun Microystems. Java 2 platform, micro edition (j2me). URL: http://java.sun.com/j2me/index.jsp (Last Accessed: 2005-01-17).

[53] Microsoft Corporation. Windows xp embedded home page. URL: http://-msdn.microsoft.com/embedded/windowsxpembedded/default.aspx (Last Accessed: 2005-01-17).

[54] Microsoft Corporation. Windows ce homepage. URL: http://msdn.-microsoft.com/embedded/windowsce/default.aspx (Last Accessed: 2005-01-17).

[55] C. M. Netter and L. F. Baceller. Assessing the real-time properties of windows ce 3.0. In *Proceedings of Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, May 2001.

[56] Object Management Group. MinimumCORBA 1.0. http://-www.omg.org/technology/documents/formal/minimum_CORBA.htm (Last Accessed: 2005-01-17).

[57] Object Management Group. Real-Time Corba. URL: http://www.omg.org/technology/documents/formal/real-time_CORBA.htm (Last Accessed: 2005-01-17).

II

Included Papers

Chapter 5

Paper A: Evaluation of Component Technologies with Respect to Industrial Requirements

Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin
In Euromicro Conference, Component-Based Software Engineering Track Rennes, France, August 2004

Abstract

We compare existing component technologies for embedded systems with respect to industrial requirements. The requirements are collected from the vehicular industry, but our findings are applicable to similar industries developing resource constrained safety critical embedded distributed real-time computer systems.

One of our conclusions is that none of the studied technologies is a perfect match for the industrial requirements. Furthermore, no single technology stands out as being a significantly better choice than the others; each technology has its own pros and cons.

The results of our evaluation can be used to guide modifications or extensions to existing technologies, making them better suited for industrial deployment. Companies that want to make use of component-based software engineering as available today can use this evaluation to select a suitable technology.

5.1 Introduction

Component-Based Software Engineering (CBSE) has received much attention during the last couple of years. However, in the embedded-system domain, use of component technologies has had a hard time gaining acceptance; software-developers are still, to a large extent, using monolithic and platform-dependent software technologies.

We try to find out why embedded-software developers have not embraced CBSE as an attractive tool for software development. We do this by evaluating a set of component technologies with respect to industrial requirements. The requirements have been collected from industrial actors within the business segment of heavy vehicles, and have been presented in our previous work [1]. Examples of heavy vehicles include wheel loaders, excavators, forest harvesters, and combat vehicles. The software systems developed within this market segment can be characterised as resource constrained, safety critical, embedded, distributed, real-time, control systems. Our findings in this study should be applicable to other domains with similar characteristics.

Our evaluation, between requirements and existing technologies, does not only help to answer why component-based development has not yet been embraced by the embedded-systems community. It also helps us to identify what parts of existing technologies could be enhanced, to make them more appropriate for embedded-system developers. Specifically, it will allow us to select a component technology that is a close match to the requirements, and if needed, guide modifications to that technology.

The reason for studying component-based development in the first place, is that software developers can achieve considerable business benefits in terms of reduced costs, shortened time-to-market and increased software quality by applying a suitable component technology. The component technology should rely on powerful design and compile-time mechanisms and simple and predictable run-time behaviour.

There is however significant risks and costs associated with the adoption of a new development technique (such as component-based development). These must be carefully evaluated before introduced in the development process. One of the apparent risks is that the selected component technology turns out to be inappropriate for its purpose; hence, the need to evaluate component technologies with respect to requirements expressed by software developers.

5.2 Requirements

The requirements discussed and described in this section are based on a previously conducted investigation [1]. The requirements found in that investigation are divided into two main groups, the technical requirements (Section 5.2.1) and the development process related requirements (Section 5.2.2). In addition, Section 5.2.3 contains derived requirements, i.e. requirements that we have synthesised from the requirements in Sections 5.2.1 and 5.2.2 but that are not explicitly stated requirements from the vehicular industry [1].

5.2.1 Technical Requirements

The technical requirements describe industrial needs and desires regarding technical aspects and properties of a component technology.

Analysable

System analysis, with respect to non-functional properties, such as timing behaviour and memory consumption is considered highly attractive. In fact, it is one of the single most distinguished requirements found in our investigation.

When analysing a system built from well-tested, functionally correct, components, the main issue is associated with composability. The composition process must ensure that non-functional properties, such as the communication, synchronisation, memory, and timing characteristics of the system, are predictabe [2].

Testable and debugable

It is required that tools exist that support debugging, both at component level (e.g., a graphical debugging tool), as well as on source code level.

Testing and debugging is one of the most commonly used techniques to verify software systems functionality. Testing is a very important complement to analysis, and testability should not be compromised when introducing a component technology. Ideally, the ability to test embedded-system software should be improved when using CBSE, since it adds the ability to test components in isolation.

Portable

The components, and the infrastructure surrounding them, should be platform independent to the highest degree possible. Here, platform independency means (1) hardware independent, (2) real-time operating system (RTOS) independent and (3) communications protocol independent. The components are kept portable by minimising the number of dependencies to the software platform. Eventually such dependencies are off course necessary to construct an executable system, however the dependencies should be kept to a minimum, and whenever possible dependencies should be generated automatically by configuration tools.

Resource Constrained

The components should be small and light-weighted and the components infrastructure and framework should be minimised. Ideally there should be no run-time overhead compared to not using a CBSE approach. Hardware used in embedded real-time systems is usually resource constrained, to lower production cost and thereby increase profit.

One possibility, that significantly can reduce resource consumption of components and the component framework, is to limit run-time dynamics. This means that it is desirable only to allow static, off-line, configured systems. Many existing component technologies have been design to support high run-time dynamics, where components are added, removed and reconfigured during run-time.

Component Modelling

The component modelling should be based on a standard modelling language like UML [3] or UML 2.0 [4]. The main reason to choose a standard like UML is that it is well known and thoroughly tested, with tools and formats supported by many third-party developers. The reason for the vehicular industry to have specific demands in this detail, is that this business segment does not have the knowledge, resources or desire to develop their own standards and practices.

Computational Model

Components should preferably be passive, i.e., they should not contain their own threads of execution. A view where components are allocated to threads during component assembly is preferred, since this is conceptually simple, and also believed to enhance reusability.

The computational model should be focused on a pipes-and-filters model [5]. This is partly due to the well known ability to schedule and analyse this model off-line. Also, the pipes-and-filters model is a good conceptual model for control applications.

5.2.2 Development Requirements

When discussing component-based development with industry, development process requirements are at least as important as the technical requirements. To obtain industrial reliance, the development requirements need to be addressed by the component technology and its associated tools.

Introducible

Appropriate support to gradually migrate to a new technology should be provided by the component technology. It is important to make the change in development process and techniques as safe and inexpensive as possible. Revolutionary changes in development techniques are associated with high risks and costs. Therefore a new technology should be possible to divide into smaller parts, which can be introduced incrementally. Another aspect, to make a technology introducible, is to allow legacy code within systems designed with the new technology.

Reusable

Components should be reusable, e.g., for use in new applications or environments than those for which they where originally designed [6]. Reusability can more easily be achieved if a loosely coupled component technology is used, i.e., the components are focusing on functionality and do not contain any direct operating system or hardware dependencies. Reusability is further enhanced by the possibility to use configuration parameters to components.

A clear, explicit, and well-defined component interface is crucial to enhance the software reusability. Also, specification of non-functional properties and requirements (such as execution time, memory usage, deadlines, etc.) simplify reuse of components since it makes (otherwise) implicit assumptions explicit. Behavioural descriptions (such as state diagrams or interaction diagrams) of components can be used to further enhance reusability.

Maintainable

The components should be easy to change and maintain, developers that are about to change a component need to understand the full impact of the proposed change. Thus, not only knowledge about component interfaces and their expected behaviour is needed. Also, information about current deployment contexts may be needed in order not to break existing systems. The components can be stored in a repository where different versions and variants need to be managed in a sufficient way. The maintainability requirement also includes sufficient tools supporting the service of deployed and delivered products. These tools need to be component aware and handle error diagnostics from components and support for updating software components.

Understandable

The component technology and the systems constructed using it should be easy to understand. This should also include making the technology easy and intuitive to use in a development project.

The reason for this requirement is to simplify evaluation and verification both on the system level and on the component level. Focusing on an understandable model makes the development process faster and it is likely that there will be fewer bugs. This requirement is also related to the introducible requirement (Section 5.2.2) since an understandable technique is more introducible.

It is desirable to hide as much complexity as possible from system developers. Ideally, complex tasks (such as mapping signals to memory areas or bus messages, or producing schedules or timing analysis) should be performed by tools.

5.2.3 Derived Requirements

Here, we present requirements that we have synthesised from the requirements in sections 5.2.1 and 5.2.2, but that are not explicit requirements from industry.

Source Code Components

A component should be source code, i.e., no binaries. Companies are used to have access to the source code, to find functional errors, and enable support for white box testing (Section 5.2.1). Since source code debugging is demanded, even if a component technology is used, black box components is undesirable.

However, the desire to look into the components does not necessary imply a desire to be allowed to modify them.[1]

Using black-box components would lead to a fear of loosing control over the system behaviour (Section 5.2.2). Provided that all components in the systems are well tested, and that the source code are checked, verified, and qualified for use in the specific surrounding, the companies might alleviate their source code availability.

Also with respect to the resource constrained requirement (Section 5.2.1), source code components allow for unused parts of the component to be removed at compile time.

Static Configurations

Better support for the technical requirements of analysability (Section 5.2.1), testability (Section 5.2.1), and resource consumption (Section 5.2.1), are achieved by using pre-runtime configuration. Here, configuration means both configuration of component behaviour and interconnections between components. Component technologies for use in the Office/Internet domain usually focus on dynamic configurations [7, 8]. This is of course appropriate in these specific domains, where one usually has access to ample resources. Embedded systems, however, face another reality; with resource constrained nodes running complex, dependable, control applications.

However, most vehicles can operate in different modes, hence the technology must support switches between a set of statically configured modes. Static configuration also improves the development process related requirement of understandability (Section 5.2.2), since each possible configuration is known before run-time.

5.3 Component Technologies

In this section, existing component technologies for embedded systems are described and evaluated. The technologies originate both from academia and industry. The selection criterion for a component technology has firstly been that there is enough information available, secondly that the authors claim that

[1]This can be viewed as a "glass box" component model, where it possible to acquire a "use-only" license from a third party. This license model is today quite common in the embedded systems market.

the technology is suitable for embedded systems, and finally we have tried to achieve a combination of both academic and industrial technologies.

The technologies described and evaluated are PECT, Koala, Rubus Component Model, PBO, PECOS and CORBA-CCM. We have chosen CORBA-CCM to represent the set of technologies existing in the PC/Internet domain (other examples are COM, .NET [7] and Java Enterprise Beans [8]) since it is the only technology that explicitly address embedded and real-time issues. Also, the Windows CE version of .NET [7] is omitted, since it is targeted towards embedded display-devices, which only constitute a small subset of the devices in vehicular systems. The evaluation is based on publicly available, documentation.

5.3.1 PECT

A Prediction-Enabled Component Technology (PECT) [9] is a development infrastructure that incorporates development tools and analysis techniques. PECT is an ongoing research project at the Software Engineering Institute (SEI) at the Carnegie Mellon University.[2] The project focuses on analysis; however, the framework does not include any concrete theories - rather definitions of how analysis should be applied. To be able to analyse systems using PECT, proper analysis theories must be found and implemented and a suitable underlying component technology must be chosen.

A PECT include an abstract model of a component technology, consisting of a construction framework and a reasoning framework. To concretise a PECT, it is necessary to choose an underlying component technology, define restrictions on that technology (to allow predictions), and find and implement proper analysis theories. The PECT concept is highly portable, since it does not include any parts that are bound to a specific platform, but in practise the underlying technology may hinder portability. For modelling or describing a component-based system, the Construction and Composition Language (CCL) [9] is used. The CCL is not compliant to any standards. PECT is highly introducible, in principle it should be possible to analyse a part of an existing system using PECT. It should be possible to gradually model larger parts of a system using PECT. A system constructed using PECT can be difficult to understand; mainly because of the mapping from the abstract component model to the concrete component technology. It is likely that systems looking identical at the PECT-level behave differently when realised on different component technologies.

[2]Software Engineering Institute, CMU; http://www.sei.cmu.edu

PECT is an abstract technology that requires an underlying component technology. For instance, how testable and debugable a system is depends on the technical solutions in the underlying run-time system. Resource consumption, computational model, reusability, maintainability, black- or white-box components, static- or dynamic-configuration are also not possible to determine without knowledge of the underlying component technology.

5.3.2 Koala

The Koala component technology [10] is designed and used by Philips[3] for development of software in consumer electronics. Typically, consumer electronics are resource constrained since they use cheap hardware to keep development costs low. Koala is a light weight component technology, tailored for Product Line Architectures [11]. The Koala components can interact with the environment, or other components, through explicit interfaces. The components source code is fully visible for the developers, i.e., there are no binaries or other intermediate formats. There are two types of interfaces in the Koala model, the provides- and the requires- interfaces, with the same meaning as in UML 2.0 [4]. The provides interface specify methods to access the component from the outside, while the required interface defines what is required by the component from its environment. The interfaces are statically connected at design time.

One of the primary advantages with Koala is that it is resource constrained. In fact, low resource consumption was one of the requirements considered when Koala was created. Koala use passive components allocated to active threads during compile-time; they interact through a pipes-and-filters model. Koala uses a construction called thread pumps to decrease the number of processes in the system. Components are stored in libraries, with support for version numbers and compatibility descriptions. Furthermore components can be parameterised to fit different environments.

Koala does not support analysis of run-time properties. Research has presented how properties like memory usage and timing can be predicted in general component-based systems, but the thread pumps used in Koala might cause some problems to apply existing timing analysis theories. Koala has no explicit support for testing and debugging, but they use source code components, and a simple interaction model. Furthermore, Koala is implemented for a specific operating system. A specific compiler is used, which routes all inter-component

[3]Phillips International, Inc; Home Page http://www.phillips.com

and component to operating system interaction through Koala connectors. The modelling language is defined and developed in-house, and it is difficult to see an easy way to gradually introduce the Koala concept.

5.3.3 Rubus Component Model

The Rubus Component Model (Rubus CM) [12] is developed by Arcticus systems.[4] 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. The Rubus Operating System (Rubus OS) [13] has one time-triggered part (used for time-critical hard real-time activities) and one event-triggered part (used for less time-critical soft real-time activities). However, the Rubus CM is only supported by the time-triggered part.

The Rubus CM runs on top of the Rubus OS, and the component model requires the Rubus configuration compiler. There is support for different hardware platforms, but regarding to the requirement of portability (Section 5.2.1), this is not enough since the Rubus CM is too tightly coupled to the Rubus OS. The Rubus OS is very small, and all component and port configuration is resolved off-line by the Rubus configuration compiler.

Non-functional properties can be analysed during desing-time since the component technology is statically configured, but timing analysis on component and node level (i.e., schedulability analysis) is the only analysable property implemented in the Rubus tools. Testability is facilitated by static scheduling (which gives predictable execution patterns). Testing the functional behaviour is simplified by the Rubus Windows simulator, enabling execution on a regular PC.

Applications are described in the Rubus Design Language, which is a non-standard modelling language. The fundamental building blocks are passive. The interaction model is the desired pipes-and-filters (Section 5.2.1). The graphical representation of a system is quite intuitive, and the Rubus CM itself is also easy to understand. Complexities such as schedule generation and synchronisation are hidden in tools.

The components are source code and open for inspection. However, there is no support for debugging the application on the component level. The components are very simple, and they can be parameterised to improve the possibility to change the component behaviour without changing the component source code. This enhances the possibilities to reuse the components.

[4]Arcticus Systems; Home Page http://www.arcticus.se

Smaller pieces of legacy code can, after minor modifications, be encapsulated in Rubus components. Larger systems of legacy code can be executed as background service (without using the component concept or timing guarantees).

5.3.4 PBO

Port Based Objects (PBO) [14] combines object oriented design, with port automaton theory. PBO was developed as a part of the Chimera Operating System (Chimera OS) project [15], at the Advanced Manipulators Laboratory at Carnegie Mellon University.[5] Together with Chimera, PBO forms a framework aimed for development of sensor-based control systems, with specialisation in reconfigurable robotics applications. One important goal of the work was to hide real-time programming and analysis details. Another explicit design goal for a system based on PBO was to minimise communication and synchronisation, thus facilitating reuse.

PBO implements analysis for timeliness and facilitates behavioural models to ensure predictable communication and behaviour. However, there are few additional analysis properties in the model. The communication and computation model is based on the pipes-and-filters model, to support distribution in multiprocessor systems the connections are implemented as global variables. Easy testing and debugging is not explicitly addressed. However, the technology relies on source code components and therefore testing on a source code level is achievable. The PBOs are modular and loosely coupled to each other, which admits easy unit testing. A single PBO-component is tightly coupled to the Chimera OS, and is an independent concurrent process.

Since the components are coupled to the Chimera OS, it can not be easily introduced in any legacy system. The Chimera OS is a large and dynamically configurable operating system supporting dynamic binding, it is not resource constrained.

PBO is a simple and intuitive model that is highly understandable, both at system level and within the components themselves. The low coupling between the components makes it easy to modify or replace a single object. PBO is built with active and independent objects that are connected with the pipes-and-filters model. Due to the low coupling between components through simple communication and synchronisation the objects can be considered to be highly reusable. The maintainability is also affected in a good way due to the

[5]Carnegie Mellon University; Home Page http://www.cmu.edu

loose coupling between the components; it is easy to modify or replace a single component.

5.3.5 PECOS

PECOS[6] (PErvasive COmponent Systems) [16] is a collaborative project between ABB Corporate Research Centre[7] and academia. The goal for the PECOS project was to enable component-based technology with appropriate tools to specify, compose, validate and compile software for embedded systems. The component technology is designed especially for field devices, i.e., reactive embedded systems that gathers and analyse data via sensors and react by controlling actuators, valves, motors etc. Furthermore, PECOS is analysable, since much focus has been put on non-functional properties such as memory consumption and timeliness.

Non-functional properties like memory consumption and worst-case execution-times are associated with the components. These are used by different PECOS tools, such as the composition rule checker and the schedule generating and verification tool. The schedule is generated using the information from the components and information from the composition. The schedule can be constructed off-line, i.e., a static pre-calculated schedule, or dynamically during run-time.

PECOS has an execution model that describes the behaviour of a field device. The execution model deals with synchronisation and timing related issues, and it uses Petri-Nets [17] to model concurrent activities like component compositions, scheduling of components, and synchronisation of shared ports [18]. Debugging can be performed using COTS debugging and monitoring tools. However, the component technology does not support debugging on component level as described in Section 5.2.1.

The PECOS component technology uses a layered software architecture, which enhance portability (Section 5.2.1). There is a Run-Time Environment (RTE) that takes care of the communication between the application specific parts and the real-time operating system. The PECOS component technology uses a modelling language that is easy to understand, however no standard language is used. The components communicate using a data-flow-oriented interaction, it is a pipes-and-filters concept, but the component technology uses a shared memory, contained in a blackboard-like structure.

[6]PECOS Project, Home Page: http://www.pecos-project.org/

[7]ABB Corporate Research Centre in Ladenburg, Home Page: http://www.abb.com/

Since the software infrastructure does not depend on any specific hardware or operating system, the requirement of introducability (Section 5.2.2) is to some extent fulfilled. There are two types of components, leaf components (black-box components) and composite components. These components can be passive, active, and event triggered. The requirement of openness is not considered fulfilled, due to the fact that PECOS uses black-box components. In later releases, the PECOS project is considering to use a more open component model [19]. The devices are statically configured.

5.3.6 CORBA Based Technologies

The Common Object Request Broker Architecture (CORBA) is a middleware architecture that defines communication between nodes. CORBA provides a communication standard that can be used to write platform independent applications. The standard is developed by the Object Management Group [8] (OMG). There are different versions of CORBA available, e.g., MinimumCORBA [20] for resource constrains systems, and RT-CORBA [21] for time-critical systems.

RT-CORBA is a set of extensions tailored to equip Object Request Brokers (ORBs) to be used for real-time systems. RT-CORBA supports explicit thread pools and queuing control, and controls the use of processor, memory and network resources. Since RT-CORBA adds complexity to the standard CORBA, it is not considered very useful for resource-constrained systems. MinimumCORBA defines a subset of the CORBA functionality that is more suitable for resource-constrained systems, where some of the dynamics is reduced.

OMG has defined a CORBA Component Model (CCM) [22], which extends the CORBA object model by defining features and services that enables application developers to implement, mange, configure and deploy components. In addition the CCM allows better software reuse for server-applications and provides a greater flexibility for dynamic configuration of CORBA applications.

CORBA is a middleware architecture that defines communication between nodes, independent of computer architecture, operating system or programming language. Because of the platform and language independence CORBA becomes highly portable. To support the platform and language independence, CORBA implements an Object Request Broker (ORB) that during run-time acts as a virtual bus over which objects transparently interact with other objects located locally or remote. The ORB is responsible for finding a requested

[8] Object Management Group. CORBA Home Page. http://www.omg.org/corba/

objects implementation, make the method calls and carry the response back to the requester, all in a transparent way. Since CORBA run on virtually any platform, legacy code can exist together with the CORBA technology. This makes CORBA highly introducible.

While CORBA is portable, and powerful, it is very run-time demanding, since bindings are performed during run-time. Because of the run-time decisions, CORBA is not very deterministic and not analysable with respect to timing and memory consumption. There is no explicit modelling language for CORBA. CORBA uses a client server model for communication, where each object is active. There are no non-functional properties or any specification of interface behaviour. All these things together make reuse harder. The maintainability is also suffering from the lack of clearly specified interfaces.

5.4 Summary of Evaluation

In this section we assign numerical grades to each of the component technologies described in Section 5.3, grading how well they fulfil each of the requirements of Section 5.2. The grades are based on the discussion summarised in Section 5.3. We use a simple 3 level grade, where 0 means that the requirement is not addressed by the technology and is hence not fulfilled, 1 means that the requirement is addressed by the technology and/or that is partially fulfilled, and 2 means that the requirement is addressed and is satisfactory fulfilled. For PECT, which is not a complete technology, several requirements depended on the underlying technology. For these requirements we do not assign a grade (indicated with NA, Not Applicable, in Figure 5.1). For the CORBA-based technologies we have listed the best grade applicable to any of the CORBA flavours mentioned in Section 5.3.6.

For each requirement we have also calculated an average grade. This grade should be taken with a grain of salt, and is only interesting if it is extremely high or extremely low. In the case that the average grade for a requirement is extremely low, it could either indicate that the requirement is very difficult to satisfy, or that component-technology designers have paid it very little attention.

In the table we see that only two requirements have average grades below 1.0. The requirement "Component Modelling" has the grade 0 (!), and "Testing and debugging" has 1.0. We also note that no requirements have a very high grade (above 1.5). This indicate that none of the requirement we have listed are general (or important) enough to have been considered by all component-technology designers. However, if ignoring CORBA (which is not designed for embedded systems) and PECT (which is not a complete component technology) we see that there are a handful of our requirements that are addressed and at least partially fulfilled by all technologies.

We have also calculated an average grade for each component technology. Again, the average cannot be directly used to rank technologies amongst each other. However, the two technologies PBO and CORBA stand out as having significantly lower average values than the other technologies. They are also distinguished by having many 0's and few 2's in their grades, indicating that they are not very attractive choices. Among the complete technologies with an average grade above 1.0 we notice Rubus and PECOS as being the most complete technologies (with respect to this set of requirements) since they have the fewest 0's. Also, Koala and PECOS can be recognised as the technologies with the broadest range of good support for our requirements, since they have the most number of 2's.

However, we also notice that there is no technology that fulfils (not even partially) all requirements, and that no single technology stands out as being the preferred choice.

	Analysable	**Testable and debugable**	**Portable**	**Resource Constrained**	**Component Modelling**	**Computational Model**	**Introducible**	**Reusable**	**Maintainable**	**Understandable**	**Source Code Components**	**Static Configuration**	**Average**	**Number of 2's**	**Number of 0's**
PECT	2	NA	2	NA	0	NA	2	NA	NA	0	NA	NA	1.2	3	2
Koala	0	1	1	2	0	2	0	2	2	2	2	2	1.3	7	3
Rubus Component Model	1	1	0	2	0	2	1	1	1	2	2	2	1.3	5	2
PBO	2	1	0	0	0	1	1	1	1	2	2	0	0.9	3	4
PECOS	2	1	2	2	0	2	1	2	1	2	0	2	1.4	7	2
CORBA Based Technologies	0	1	2	0	0	0	2	0	0	1	0	0	0.5	2	8
Average	1.2	1.0	1.2	1.2	0.0	1.4	1.4	1.2	1.0	1.5	1.2	1.2	1.1	4.3	3.5

Figure 5.1: Grading of component technologies with respect to the requirements

5.5 Conclusion

In this paper we have compared some existing component technologies for embedded systems with respect to industrial requirements. The requirements have been collected from industrial actors within the business segment of heavy vehicles. The software systems developed in this segment can be characterised as resource constrained, safety critical, embedded, distributed, real-time, control systems. Our findings should be applicable to software developers whose systems have similar characteristics.

We have noticed that, for a component technology to be fully accepted by industry, the whole systems development context needs to be considered. It is not only the technical properties, such as modelling, computation model, and openness, that needs to be addressed, but also development requirements like maintainability, reusability, and to which extent it is possible to gradually introduce the technology. It is important to keep in mind that a component technology alone cannot be expected to solve all these issues; however a technology can have more or less support for handing the issues.

The result of the investigation is that there is no component technology available that fulfil all the requirements. Further, no single component technology stands out as being the obvious best match for the requirements. Each technology has its own pros and cons. It is interesting to see that most requirements are fulfilled by one or more techniques, which implies that good solutions to these requirements exist.

The question, however, is whether it is possible to combine solutions from different technologies in order to achieve a technology that fulfils all listed requirements? Our next step is to assess to what extent existing technologies can be adapted in order to fulfil the requirements, or whether selected parts of existing technologies can be reused if a new component technology needs to be developed. Examples of parts that could be reused are file and message formats, interface description languages, or middleware specifications/implementations. Further, for a new/modified technology to be accepted it is likely that it have to be compliant to one (or even more than one) existing technology. Hence, we will select one of the technologies and try to make as small changes as possible to that technology.

Bibliography

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

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

[3] B. Selic and J. Rumbaugh. Using UML for modelling complex real-time systems, 1998. Rational Software Corporation.

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

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

[6] 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.

[7] Microsoft Component Technologies. COM/DCOM/.NET. http://www.-microsoft.com.

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

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

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

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

[12] 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).

[13] K.L. Lundbäck. Rubus OS Reference Manual – General Concepts. Arcticus Systems: http://www.arcticus.se (Last Accessed: 2005-01-18).

[14] D.B. Stewart, R.A. Volpe, and P.K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, pages pages 759 – 776, December 1997.

[15] P.K. Khosla et al. The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications. *IEEE Transactions on Systems*, 1992. Man and Cybernetics.

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

[17] M. Sgroi. Quasi-Static Scheduling of Embedded Software Using Free-Choice Petri Nets. Technical report, University of California at Berkely, May 1998.

[18] 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.

[19] R. Wuyts and S. Ducasse. Non-functional requirements in a component model for embedded systems. In *International Workshop on Specification and Verification of Component-Based Systems*, 2001. OPPSLA.

[20] Object Management Group. MinimumCORBA 1.0. http://-www.omg.org/technology/documents/formal/minimum_CORBA.htm (Last Accessed: 2005-01-17).

[21] D.C. Schmidt, D.L. Levine, and S. Mungee. The Design of the tao real-time object request broker. *Computer Communications Journal*, Summer 1997.

[22] Object Management Group. CORBA Component Model. http://-www.omg.org/technology/documents/formal/components.htm (Last Accessed: 2005-01-17).

Chapter 6

Paper B: Quality Attribute Support in a Component Technology for Vehicular Software

Mikael Åkerholm, Johan Fredriksson, Kristian Sandström, and Ivica Crnkovic
In Fourth Conference on Software Engineering Research and Practice in Sweden, Linköping, Sweden, October 2004

Abstract

The electronics in vehicles represents a class of systems where quality attributes, such as safety, reliability, and resource usage, leaven all through development. Vehicular manufacturers are interested in developing their software using a component based approach, supported by a component technology, but commercial component technologies are too resource demanding, complex and unpredictable. In this paper we provide a vehicular domain specific classification of the importance of different quality attributes for software, and a discussion of how they could be facilitated by a component technology. The results can be used as guidance and evaluation for research aiming at developing component technologies suitable for vehicular systems.

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. CBD has been extensively used for several years in desktop environments, office applications, e-business and in general Internet- and web-based distributed applications. In many other domains, for example dependable systems, CBD is utilized to a lesser degree for a number of different reasons. An important reason is the inability of component-based technologies to deal with quality attributes as required in these domains. To identify the feasibility of the CBD approach, the main concerns of the particular domain must be identified along with how the CBD approach addresses these concerns and what is its ability to provide support for solutions related to these concerns are.

There is currently a lot of research on predicting and maintaining different quality attributes within the Component Based Software Engineering (CBSE) community, (also called non-functional properties, extra-functional properties, and illities), [1, 2, 3, 4, 5]. Many of the quality attributes are conflicting and cannot be fully supported at the same time [6, 7]. Thus, it is important for application and system developers to be able to prioritize among different quality attributes when resolving conflicts.

We provide a domain specific classification of the importance of quality attributes for software in vehicles, and discuss how the attributes could be facilitated by a component technology. The discussion contribute with a general description of the desired quality attribute support in a component technology suitable for the vehicle domain and it indicates which quality attributes require explicit support. In addition, it discusses were in the technology the support should be implemented: inside or outside the components, in the component framework, on the system architecture level, or if the quality attributes are usage dependent. Quality attributes might be conflicting; e.g., it is commonly understood that flexibility and predictability are conflicting. The ranking provided by industrial partners gives domain specific guidance for how conflicts between quality attributes should be resolved. The results also enable validation and guidance for future work regarding quality attribute support in component technologies for software in vehicular systems. This guideline can be used to verify that the right qualities are addressed in the development process and that conflicting interdependent quality attributes are resolved according to the domain specific priorities.

The starting point of this work is a list of quality attributes ranked according to their importance for vehicular systems. The list is provided through a set of interviews and discussions with experts from different companies in the vehicular domain. The results of the ranking from the vehicular companies are combined with the classification of how to support different quality attributes provided in [8]. The result is an abstract description of where, which, and how different quality attributes should be supported by a component technology tailored for the vehicular industry.

A component technology as defined in [9] is a technology that can be used for building component based software applications. It implements a component model defining the set of component types, their interfaces, and, additionally, a specification of the allowable patterns of interaction among component types. A component framework is also part of the component technology, its role can be compared to the role of an operating system, and it provides a variety of deployment and run-time services to support the component model. Specialized component technologies used in different domains of embedded systems have recently been developed, e.g., [10, 11]. There are also a number of such component technologies under development in the research community, e.g., [12, 13, 14]. The existence of different component technologies can be motivated by their support for different quality attributes, although they follow the same CBSE basic principles. It has been shown that companies developing embedded systems in general consider different non functional quality attributes far more important than efficiency in software development, which explains the specialization of component technologies [12].

The outline of the remaining part of the paper is as follows. Section 2 describes the conducted research method, and section 3 the results. Section 4 is a discussion of the implications of the results, regarding the support for quality attributes in a domain specific component technology. Section 5 discusses future work, and finally the section 6 concludes the paper.

6.2 Method

The research method is divided into three ordered steps:

1. During the first step a list of relevant quality attributes were gathered;

2. In the next step technical representatives from a number of vehicular companies placed priorities on each of the attributes in the list reflecting their companies view respectively;

3. Finally a synthesis step was performed, resulting in a description of the

desired quality attribute support in a component technology for vehicular systems.

The list of quality attributes have been collected from different literature trying to cover qualities of software that interest vehicular manufactures. In order to reduce a rather long list, attributes with clear similarities in their definitions have been grouped in more generic types of properties, e.g., portability and scalability are considered covered by maintainability. Although such grouping could fade the specific characteristics of a particular attribute, it put focus on the main concerns. In the ISO 9126 standard [15], 6 quality attributes (functionality, reliability, usability, efficiency, maintainability, and portability) are defined for evaluation of software quality. However, the standard has not been adopted fully in this work; it is considered too brief and does not cover attributes important for embedded systems (e.g., safety, and predictability). Furthermore, concepts that sometimes are mixed with quality attributes (for example fault tolerance) are not classified as quality attributes, rather as methods to achieve qualities (as for example safety). Finally, functionality is of course one of the most important quality attributes of a product, indicating how well it satisfies stated or implied needs. However, we focus on quality attributes beyond functionality often called extra-functional or non-functional properties. The resulting list of quality attributes is presented below.

Extendibility the ease with which a system or component can be modified to increase its storage or functional capacity.

Maintainability the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment.

Usability the ease with which a user can learn to operate, prepare inputs for, and interpret outputs from a system or component.

Predictability to which extent different run-time attributes can be predicted during design time.

Security the ability of a system to manage, protect, and distribute sensitive information.

Safety a measure of the absence of unsafe software conditions. The absence of catastrophic consequences to the environment.

Reliability the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

Testability the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. Note: testability is not only a measurement for software, but it can also apply to the testing scheme.

Flexibility the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.

Efficiency the degree to which a system or component performs its designated functions with minimum consumption of resources (CPU, Memory, I/O, Peripherals, Networks).

Representatives from the technical staff of several companies have been requested to prioritize a list of quality attributes, reflecting each of the respective companiesŠ view. The attributes have been grouped by the company representatives in four priority classes as shown in Table 1. The nature of the quality attributes imply that no quality attribute can be neglected. It is essential to notice that placing an attribute in the lowest priority class (4) does not mean that the company could avoid that quality in their software, rather that the company does not spend extra efforts in reaching it. The following companies have been involved in the classification process:

- Volvo Construction Equipment [16] develops and manufactures a wide variety of construction equipment vehicles, such as articulated haulers, excavators, graders, backhoe loaders, and wheel loaders.
- Volvo Cars [17] develops passenger cars in the premium segment. Cars are typically manufactured in volumes in the order of several hundred thousands per year.
- Bombardier Transportation [18] is a train manufacturer, with a wide range of related products. Some samples from their product line are passenger rail vehicles, total transit systems, locomotives, freight cars, propulsion and controls, and signaling equipment.
- Scania [19] is a manufacturer of heavy trucks and buses as well as industrial and marine engines.
- ABB Robotics [20] is included in the work as a reference company, not acting in the vehicular domain. They are building industrial robots, and it is the department developing the control systems that is represented.

Priority	Description
1	very important, must be considered
2	important, something that one should try to consider
3	less important, considered if it can be achieved with a small effort
4	Unimportant, do not spend extra effort on this

Table 6.1: Priority classes used to classify the importance of the different quality attributes

As the last step we provide a discussion where we have combined the collected data from the companies with the classification of how to support different quality attributes in [8]. The combination gives an abstract description of where, which, and how different quality attributes should be supported by a component technology tailored for usage in the vehicular industry.

6.3 Results

Figure 6.1 is a diagram that summarizes the results. The attributes are prioritized by the different companies, in a scale from priority 1 (highest), to 4 (lowest) indicated on the Y-axis. On the X-axis the attributes are presented with the highest prioritized attribute as the leftmost, and lowest as rightmost. Each of the companies has one bar for each attribute, textured as indicated below the X-axis. In some cases the representatives placed an interval for the priority of certain attributes, e.g., 1-3 dependent on application; in those cases the highest priority has been chosen in the diagram.

The result shows that the involved companies have approximately similar prioritization, except on the security quality attribute where we have both highest and lowest priority. Reasonably, the most important concerns are related to dependability characteristics (i.e. to the expectation of the performance of the systems): safety, reliability and predictability. Usability is a property important for the customers but also crucial in competition on the market. Slightly less important attributes are related to the life cycle (extendibility, maintainability). This indicates that the companies are ready to pay more attention to the product performance than to the development and production costs (in that sense a component-based approach which primary concerns are of business nature, might not necessary be the most desirable approach).

The results also shows that ABB Robotics, included as a reference com-

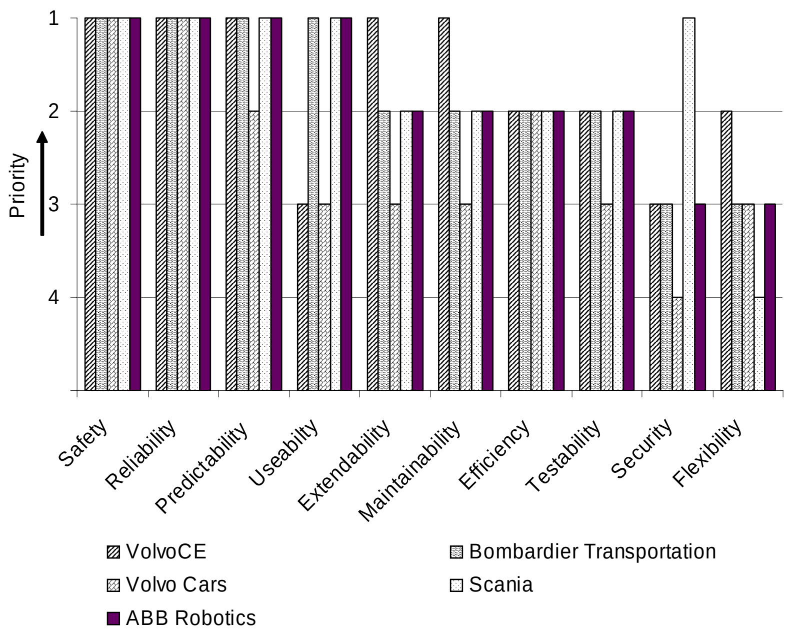


Figure 6.1: the results. Y-axis: priority of quality attributes in a scale 1 (highest), to 4 (lowest). X-axis: the attributes, with the highest prioritized attribute as the leftmost, and lowest as rightmost. Each of the companies has one bar for each attribute, textured as indicated below the X-axis.

pany outside the vehicular domain has also approximately the same opinion. It is not possible to distinguish ABB Robotics from any of the vehicular companies from a quality attribute perspective. These companies might use the same component technology with respect to quality attribute support; thus the results in the investigation indicate that the priority among quality attributes scale to a broader scope of embedded computer control systems.

6.4 Discussion of the results

A component technology may have built in support for maintaining quality attributes. However, tradeoffs between quality attributes must be made since they are interdependent [7, 6]. We will discuss how the different quality attributes can be supported by a component technology, and suggest how necessary tradeoffs can be made according to priority placed by industry. The discussion starts by treating the attribute that has received the highest priority (safety), and continues in priority order, in this way the conflicts (and tradeoffs) will be discussed in priority order. As basis for where support for a specific quality attribute should be implemented we use a classification from [8], listed below:

- Directly composable, possible to analyze given the same quality attributes from the components.
- Architecture related, possible to analyze given this attribute for the components and the assembly architecture.
- Derived attributes, possible to analyze from several attributes from the involved components.
- Usage dependent, need a usage profile to analyze this.
- System environment context, possible to analyze given environment attributes.

6.4.1 Safety

Safety is classified as dependent on the usage profile, and the system environment context. Similarly to the fact that we cannot reason about system safety without taking into consideration the surrounding context, we cannot reason about safety of a component: simply safety is not a property that can be identified on the component level. But a component technology can include numerous mechanisms that enhance safety, or simplify safety analysis. However, to perform safety analysis, usage and environment information is needed. A component technology can have support for safety kernels [21], surrounding components and supervise that unsafe conditions do not occur. Pre- and post conditions can be checked in conjunction with execution of components to detect hazardous states and check the range of input and output, used in specification of components in e.g., [22, 23]. Tools supporting safety analysis as fault tree analysis (FTA) or failure modes and effect analysis (FMEA) can also be provided with the component technology.

6.4.2 Reliability

Reliability is architecture related and usage dependent. The dominant type of impact on reliability is the usage profile but reliability is also dependent on the software architecture and how components are assembled; a fault-tolerant redundant architecture improves the reliability of the assembly of components. One possible approach to calculation of the reliability of an assembly is to use the following elements:

- Reliability of the components - Information that has been obtained by testing and analysis of the component given a context and usage profile.

- Path information (usage paths) - Information that includes usage profile and the assembly structure.

Combined, it can give a probability of execution of each component, for example by using Markov chains.

Also common for many simple systems, the reliability for a function of two components is calculated using the reliability of the components, and their relationship when performing the function. An AND relationship is when the output is dependent on correct operation of both components, and an OR occurs when the output is created when one of the two components operates correctly.

A component technology could have support for reliability, through reliability attributes associated with components, and tools that automatically determines reliability of given usage profiles, path information, and structural relationships.

It is noteworthy that even if the reliability of the components are known it is very hard to know if side effects take place that will affect an assembly of the components. E.g. a failure caused by a component writing in a memory space used by another component. A model based on these assumptions needs the means for calculating or measuring component reliability and an architecture that permits analysis of the execution path. Component models that specify provided and required interface, or implement a port-based interface make it possible to develop a model for specifying the usage paths. This is an example in which the definition of the component model facilitates the procedure of dealing with the quality attribute. One known problem in the use of Markov chains in modeling usage is the rapid growth of the chain and complexity [24]. The problem can be solved because the reliability permits a hierarchical approach. The system reliability can be analyzed by (re)using the reliability information of the assemblies and components (which can be derived or measured).

Reliability and Safety are not conflicting attributes. Reliability enhances safety, high reliability increases confidence that the system does what it is intended to and nothing else that might lead to unsafe conditions.

6.4.3 Predictability

We focus on predictability of the particular run-time attributes temporal behaviour, memory consumption, and functional behaviour. Predictability is directly composable and architecture dependent. Prediction of temporal behaviour is well explored in research within the real-time community. Depending on the run-time systems scheduling strategy, the shared resource access and execution demands of the scheduled entities, suitable prediction theories can be chosen, e.g., for fixed priority systems that are most common within industry [25, 26]. The choice of scheduling strategy is also a problem that has been addressed [27]. Static scheduled systems are more straightforward to predict than event driven systems that on the other hand are more flexible. Memory consumption can be predicted, given the memory consumption for the different components in the system [28]. However, two different types of memory consumption can be identified: static and dynamic. Static memory consumption is the most straightforward to predict, since it is a simple summation of the memory requirements of the included components. Dynamic memory consumption can be more complex, since it might be dependent on usage input, and thereby be usage dependent.

Predictability is not in conflict with the higher prioritized attributes reliability and safety. Predictable behaviour enhances safety and reliability, e.g., unpredictable behaviour cannot be safe because it is impossible to be sure that certain actions will not take place.

6.4.4 Usability

Usability is a rather complex quality attribute, which is derived from several other attributes; it is architecture related and usage dependent. Usability is not directly related to selection of component technology. Software in embedded systems (the most common and important type of software in vehicular systems) is usually not visible and does not directly interact with the user. However, more and more human-machine interaction is implemented in underlying software. In many cases we can see how the flexibility of software is abused - there are many devices (for example in infotainment) with numerous buttons and flashing screens that significantly decrease the level of usability. Use of a component technology may however indirectly contribute to usability - by building standard (user-interface) components, and by their use in different applications and products, the same style, type of interaction, functionality and similar are repeated. In this way they become recognisable and consequently easier to use.

Usability as discussed above is not in obvious conflict with any of the higher prioritized quality attributes.

6.4.5 Extendibility

Extendibility is directly composable and architecture related. It can be supported by the component technology through absence of restrictions in size related parameters, e.g., memory size, code size, and interface size. Extendibility is one of the main concerns of a component technology and it is explicitly supported Ű either by ability of adding or extending interfaces or by providing a framework that supports extendibility by easy updating of the system with new or modified components.

Extendibility is not in direct conflict with any of the higher prioritized attributes. However, conflicts may arise due to current methods used for analysis and design of safety critical systems real-time systems, the methods often results in systems that are hard to extend [29]. Predictability in turn enhances extendibility, since it makes predications of the impact of an extension possible.

6.4.6 Maintainability

Maintainability is directly composable and architecture related. A component technology supports maintainability through configuration management tools, clear architectures, and possibilities to predict impacts of applied changes.

Maintainability is not in obvious conflict with any of the higher prioritized attributes. But as for extendibility, current state of practice for achieving safety, dependability and predictability results in systems that often are hard to maintain [29]. Maintainability increases usability, while good predictability in turn increases maintainability since impacts of maintenance efforts can be predicted.

6.4.7 Efficiency

Efficiency is directly composable and architecture related. Efficiency is affected by the component technology, mainly through resource usage by the run-time system but also by interaction mechanisms. Good efficiency is equal to low memory, processor, and communication medium usage.

In the requirements for a software application it might often be the case that a certain amount of efficiency is a basic requirement, because of limited hardware resources, control performance, or user experienced responsiveness. In such cases the certain metrics must be achieved, but efficiency is potentially in conflict with many higher prioritized quality attributes. Safety related run-time mechanisms as safety kernels, and checking pre- and post conditions consume extra resources and are thus in conflict with efficiency. Reliability is often increased by redundancy, by definition conflicting with efficiency. Methods used for guaranteeing real-time behaviour are pessimistic and result in low utilization bounds [30], although it is a widely addressed research problem and improvements exist, e.g., [31, 32].

6.4.8 Testability

Testability is directly composable and architecture related. A general rule for testability is that simple systems are easier to test than complex systems; however, what engineers build is not directly related to the technology itself. Direct methods to increase testability provided by a component technology can be built in self tests in components, monitoring support in the run-time system, simulation environments, high and low level debugging information [33].

Testability is not in conflict with any of the higher prioritized quality attributes. On the contrary, it supports several other attributes, e.g., safety is increased by testing that certain conditions cannot occur, predictions are confirmed by testing, maintainability is increased if it is possible to test the impact of a change. However, efficiency tradeoffs might have to be done to enable testing. A problem with many common testing methods is the probe effect introduced by software probes used for observing the system [34]. If the probes used during testing are removed in the final product, it is not the same system that is delivered as the one tested. To avoid this problem, designers can choose to leave the probes in the final product and sacrifice efficiency, or possibly use some form of non-intrusive hardware probing methods, e.g., [35]. Reliability implemented by fault tolerance decrease testability, since faults may become hidden and complicate detection by testing.

6.4.9 Security

Security is usage dependent and dependent on the environment context, meaning that it is not directly affected by the component technology. However, mechanisms increasing security can be built in a component technology, e.g., encryption of all messages, authorization of devices that communicate on the bus.

Methods to increase security that can be built in a component technology are often in conflict with higher prioritized quality attributes, e.g., encryption is in conflict with efficiency since it require more computing, and with testability since it is harder to observe the system. Furthermore security has a low priority, and the methods to achieve it are not dependent on support from the component technology. Hence, security can be implemented without support from the component technology.

6.4.10 Flexibility

Flexibility is directly composable and architecture related. A component technology can support flexibility through the components, their interactions, and architectural styles to compose systems. Methods increasing flexibility in a component technology can be, e.g., dynamic run-time scheduling of activities based on events, run-time binding of resources, and component reconfiguration during run-time.

Flexibility has received the lowest priority of all quality attributes, and is in conflict with many higher prioritized attributes, e.g., with safety since the number of different hazardous conditions increases, with testability since the number of test cases increases and it may not be possible at all to create a realistic run-time situation thus not to test the actual system either. On the other hand flexibility increases maintainability, since a flexible system is easier to change during maintenance. It is not possible to use completely static systems with no flexibility at all when user interaction is involved, but regarding to the numerous conflicts with higher prioritized quality attributes it should be kept to a minimum in component technologies for this domain.

6.4.11 Quality Attribute Support in a Component Technology for the Automotive Domain

Having presenting the basic characteristics of quality attributes related to component technologies, and identification of present conflicts, and suggestions on how to resolve the conflict we give a brief description of the resulting suggestion of support for quality attributes in a component technology tailored for vehicular systems below:

Safety Safety cannot be fully supported by a component technology. However, safety kernels surrounding components and support for defining pre- and post conditions are suggested.

Reliability Reliability is supported to a large extent by a component technology. We suggest reliability attributes associated with components, path information including usage profile and assembly structure, and tools for analysis. There should also be support for redundant components when necessary.

Predictability Predictability is supported to a large extent. Associated to the components, attributes such as execution time, and memory consumption can be specified. Tools for automated analysis can be provided with the technology.

Usability Usability is not directly supported by a component technology.

Extendibility Extendibility is well supported. The interfaces should be easy to extend and it should be easy to add new components to an existing system. There should be no size related restrictions with respect to memory, code, and interface.

Maintainability Maintainability is well supported by a component technology. The support is provided through configuration management tools, and the fact that using well defined components gives a clear and maintainable architecture.

Efficiency Efficiency is suggested to be supported to a fairly high level. We suggest support through small and efficient run-time systems, however not to the cost of suggested safety and reliability related run-time mechanisms.

Testability Testability is supported to a large extent. The support is suggested to be monitoring possibilities in the run-time system, simulation and debug possibilities.

Security Security is not directly supported.

Flexibility Flexibility is not directly supported.

6.5 Future Work

We will continue with research towards enabling CBSE for automotive systems. One part is to continue investigating the requirements on quality attributes from the domain, with our present and other industrial partners. Another part is an analysis of particular component models to investigate their abilities of supporting these quality attributes. A third part is to enable support for quality attributes in the component technologies we are developing as prototypes suitable for the domain AutoComp [36] and SaveComp [12], but we will also asses to which extent other existing component technologies can be used in order to meet the industrial requirements.

6.6 Conclusions

We have presented a classification of the importance of quality attributes for software made by some companies in the vehicular domain; the results showed that the companies agreed upon the priority for most of the attributes. The most important concerns showed to be related to dependability characteristics (safety, reliability and predictability). Usability received a fairly high priority. Slightly less important attributes where those related to the life cycle (extendibility, maintainability), while security and flexibility received the lowest priority. We also included a company outside the domain in the investigation, it turned out that they also agreed upon the classification; it might be that the classification scale to a broader scope of embedded systems.

Furthermore, we have discussed how the attributes could be facilitated by a component technology, and were in the technology the support should be implemented: inside or outside the components, in the framework, or if the quality attributes are usage dependent. The discussion is concluded by a brief suggestion of quality attribute support for a component technology.

Acknowledgements We would like to thank our industrial partners for their time, and the interest they have shown in discussing quality attributes. Thanks to Joakim Fröberg from Volvo CE, Jakob Axelsson from Volvo Cars, Mattias Ekman from Bombardier Transportation, Ola Larses and Bo Neidenström from Scania, and Bertil Emertz from ABB Robotics.

Bibliography

[1] I. Crnkovic and M. Larsson. Classification of quality attributes for predictability in component-based systems. In *In DSN 2004 Workshop on Architecting Dependable Systems*, June 2004.

[2] G.A. Moreno, S.A. Hissam, and K.C. Wallnau. Statistical models for empirical component properties and assembly-level property predictions: Towards standard labeling. In *Proceedings of 5th Workshop on component based software engineering*, 2002.

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

[4] H.W. Schmidt. Trustworthy components: Compositionality and prediction. *Journal of Systems and Software*, 65(3):212–225, 2003.

[5] J. Stafford and J. McGregor. Issues in the reliability of composed components. In *5th workshop on component based software engineering (CBSE5)*, 2002.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] ISO/IEC. *Software engineering – Product quality – Part 1: Quality model, ISO/IEC 9126-1*, 2001.

[16] Volvo construction equipment homepage. http://www.volvo.com/-constructionequipment.

[17] Volvo cars homepage. http://www.volvocars.com/.

[18] Bombardier transportation homepage. http://www.transportation.-bombardier.com/.

[19] Scania homepage. http://www.scania.com/.

[20] Abb robotics homepage. http://www.abb.com/robotics.

[21] J. Rushby. Kernel for safety, in safe and secure computing systems. Technical report, Blackwell Scientific Publications, Londres, 1989.

[22] J. Chessman and J. Daniels. *UML Componets - A simple process for specifying Component-Based Software*. Reading, MA: Addison-Wesley, 2000.

[23] D. D'Souza and A.C. Wills. *Objects, Components and Frameworks: The Catalysis Approach*. Reading, MA: Addison-Wesley, 1998.

[24] 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.

[25] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, , and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems Journal*, 8(2/3):173–198, 1995.

[26] O. Redell and M. Törngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proc. Eighth IEEE Real-Time and Embedded Tech-nology and Applications Symposium*. IEEE, September 2002.

[27] J. Xu and D. L. Parnas. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Transactions on Software Engineerin*, 19(1):70–84, 1993.

[28] 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.

[29] A. Burns and J. A. McDermid. Real-time safety-critical systems: analysis and synthesis. *Software Engineering Journal*, 9(6):267–281, Nov 1994.

[30] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in hard-real-time environment. *Journal of the Association for Computing Machinery (ACM)*, 20(1):46–61, 1973.

[31] T. F. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transactions on Computers*, 53(3):334–350, Mar 2004.

[32] C. Deji, A. K. Mok, and K. Tei-Wei. Utilization bound revisited. *IEEE Transactions on Computers*, 52(3):351–361, March 2003.

[33] H. Thane. *Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, May 2000.

[34] J. Gait. A probe effect in concurrent programs. *Software Practise and Experience*, 16(3), March 1986.

[35] M. El Shobaki and L. Lindh. A hardware and software monitor for high-level system-on-chip verification. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, March 2001.

[36] 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.

Chapter 7

Paper C: 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.

7.1 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.

7.2 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.

7.3 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.

7.4 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 Ű 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).

7.5 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.

7.5.1 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.

7.5.2 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.

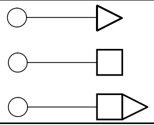
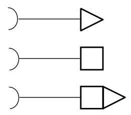
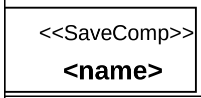
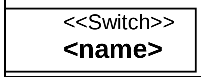
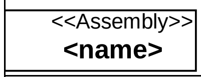
Symbol	Interpretation
	**Input ports** - The upper is an input port with a trigger, and no data. The middle symbol is an input port with data and no triggering, and the lower symbol is an input port with data and triggering.
	**Output port** - Similar to the input ports, the upper is symbol is an output 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 7.1: Graphical Syntax of SaveCCM

7.5.3 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 7.1 are used.

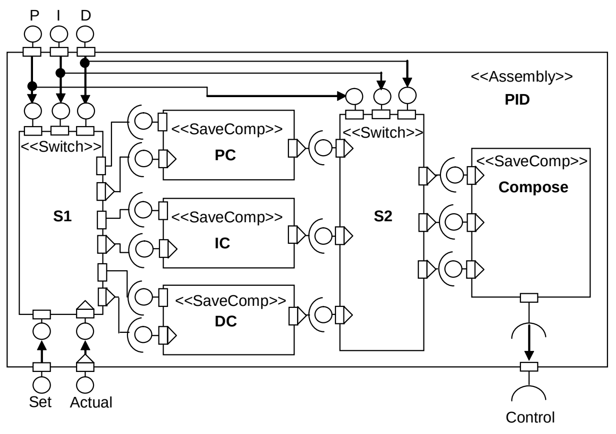


Figure 7.2: Generic PID Controller

7.5.4 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 7.2.

The following is the same example as in Figure 7.2 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;
}
```

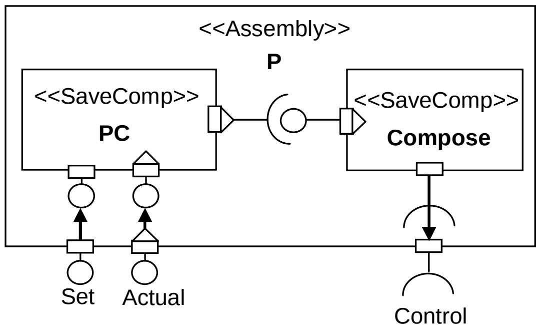


Figure 7.3: Generic PID, statically configured as a P controller

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 7.3.

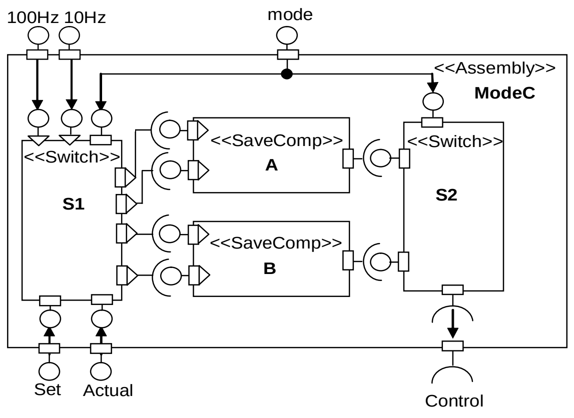


Figure 7.4: Switch effectuating mode-switches, with different execution rates

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 7.4.

7.6 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

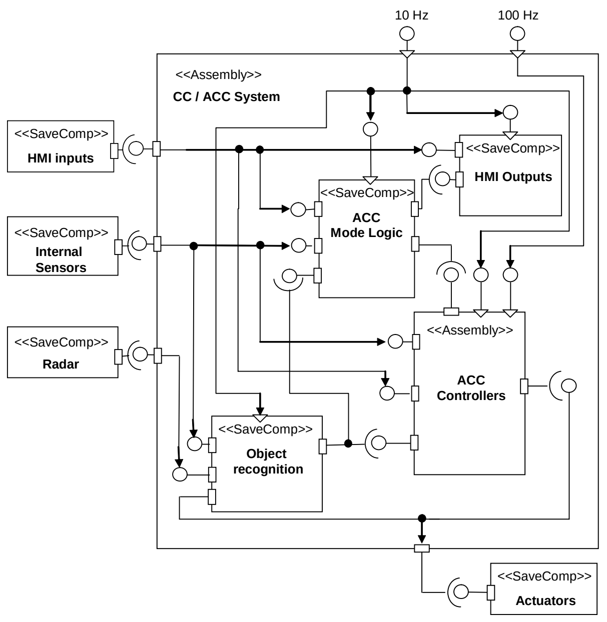


Figure 7.5: ACC system

and distance of the vehicle in front. Figure 7.5 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.

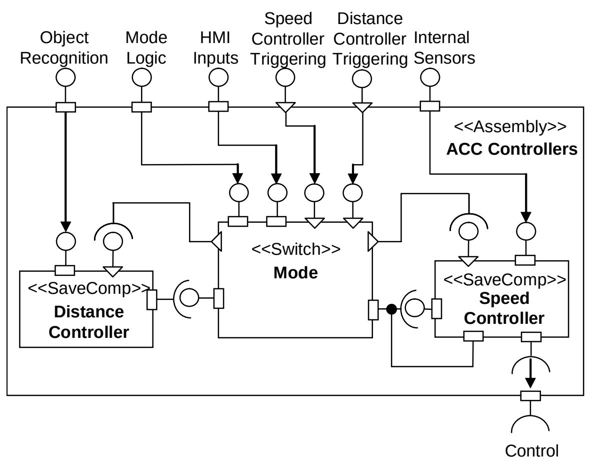


Figure 7.6: ACC controllers assembly

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 7.6.

In Figure 7.6 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 7.1 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 7.1: 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.

7.7 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. IEEE, 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: 2005-01-18).

[8] East, embedded electronic architecture project. http://www.east-eea.net/ (Last Accessed: 2005-01-18).

[9] O. Bridal C. Norström S. Larsson H. Lönn M. Strömberg H. Hansson, H. Lawson. 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 8

Paper D: Introducing a Component Technology for Safety Critical Embedded Real-Time Systems

Kristian Sandström, Johan Fredriksson, and Mikael Åkerholm
In International Symposium on Component-based Software Engineering (CBSE7), Edinburgh, Scotland, May 2004

Abstract

Safety critical embedded real-time systems represent a class of systems that has attracted relatively little attention in research addressing component based software engineering. Hence, the most widely spread component technologies are not used for resource constrained safety critical real-time systems. They are simply to resource demanding, to complex and to unpredictable. In this paper we show how to use component based software engineering for low footprint systems with very high demands on safe and reliable behaviour. The key concept is to provide expressive design time models and yet resource effective run-time models by statically resolve resource usage and timing by powerful compile time techniques. This results in a component technology for resource effective and temporally verified mapping of a component model to a commercial real-time operating system.

8.1 Introduction

The vehicle domain represents a class of embedded real-time systems where the requirements on safety, reliability, resource usage, and cost leaven all through development. Historically, the development of such systems has been done using only low level programming languages, to guarantee full control over the system behaviour. As the complexity and the amount of functionality implemented by software increase, so does the cost for software development. Therefore it is important to introduce software development paradigms that increase software development productivity. Furthermore, since product lines are common within the domain, issues of commonality and reuse is central for reducing cost as well as increasing reliability.

Component based software engineering is a promising approach for efficient software development, enabling well defined software architectures as well as reuse. Although component technologies have been developed addressing different demands and domains, there are few component technologies targeting the specific demands of safety critical embedded real-time systems. Critical for the safe and reliable operation of these systems is the real-time behaviour, where the timeliness of computer activities is essential. To be able to guarantee these properties it is necessary to apply real-time systems theory. Thus, a component technology to be used within this domain has to address specification, analysis, and implementation of real-time behaviour.

A typical real-time constraint is a deadline on a transaction of co-operating activities. A transaction in these systems would typically sample information about the environment, perform calculations based on that information and accordingly apply a response to the environment, all within a limited time frame. Also important is the ability to constrain the variation in periodicity of an activity (jitter). The reason for this is that variations in periodicity of observations of the environment and responses to the same, will affect the control performance. Hence, a component technology for this domain should have the ability to clearly express and efficiently realize these constraints [1],[2],[3],[4].

The work described in this paper present a component technology for safety critical embedded real-time systems that is based on experience from our previous work with introducing state-of-the-art real-time technology in the vehicle industry. The benefits in development have been discussed in [5] and have also been proven by long industrial use. That real-time technology has been incorporated in the Rubus development suite and has been further developed [6]. Experience from the industrial application of the research reveals that a proper component model is not enough; success requires an unbroken chain of models, methods, and tools from early design to implementation and run-time environment.

The contribution of the work presented in this paper includes a component technology for resource effective and temporally verified mapping of a component model to a resource structure such as a commercial Real-Time Operating System (RTOS). This is made possible by introduction of a component model that support specification of high level real-time constraints, by presenting a mapping to a real-time model permitting use of standard real-time theory. Moreover, it supports synthesis of run-time mechanisms for predictable execution according to the temporal specification in the component model. Furthermore, in this work some limitations in previous work with respect to specification and synthesis of real-time behaviour are removed. These limitations are partially discussed in [5] and is mainly related to jitter and execution behaviour.

Many common component technologies are not used for resource constrained systems, nor safety critical, neither real-time systems. They are simply to resource demanding, to complex and unpredictable. The research community has paid attention to the problem, and recent research has resulted in development of more suitable technologies for these classes of systems. Philips use Koala [7], designed for resource constrained systems, but without support for real-time verification. Pecos [8] is a collaboration project between ABB and University partners with focus on a component technology for field devices. The project considers different aspects related to real-time and resource constrained systems, during composition they are using components without code introspection possibilities that might be a problem for safety critical applications. Rubus OS [6] is shipped with a component technology with support for prediction of real-time behaviour, though not directly on transactions and jitter constraints and not on sporadic activities. Stewart, Volpe, and Khosla suggest a combination of object oriented design and port automaton theory called Port Based Objects [9]. The port automaton theory gives prediction possibilities for control applications, although not for transactions and jitter constraints discussed in this paper. Schmidt and Reussner propose to use transition functions to model and predict reliability in [10]; they are not addressing real-time behaviour. Wallnau et al. suggest to restrict the usage of component technologies, to enable prediction of desired run-time attributes in [11], the work is general and not focused on particular theories and methods like the work presented in this paper.

The outline of the rest of this paper is as follows; section 2 gives an overview of the component technology. In section 3 the component model is described and its transformation to a real-time model is explained in section 4. Section

5 presents the steps for synthesis of real-time attributes and discusses run-time support. Finally, in section 6, future work is discussed and the paper is concluded.

8.2 Component Technology

In this section we will give an overview of the component technology facilitating component based software development for safety-critical embedded real-time systems. We will hereafter refer to this component technology as the AutoComp technology. A key concept in AutoComp is that it allows engineers to practise Component Based Software Engineering (CBSE) without involving heavy run-time mechanisms; it relies on powerful design and compile-time mechanisms and simple and predictable run-time mechanisms. AutoComp is separated into three different parts; component model, real-time model and run-time system model. The component model is used during design time for describing an application. The model is then transformed into a real-time model providing theories for synthesis of the high level temporal constraints into attributes of the run-time system model. An overview of the technology can be seen in Fig. 8.1. The different steps in the figure is divided into design time, compile time, and run-time to display at which point in time during development they are addressed or used.

During design time, developers are only concerned with the component model and can practise CBSE fully utilizing its advantages. Moreover, high level temporal constraints in form of end-to-end deadlines and jitter are supported. Meaning that developers are not burdened with the task of setting artificial requirements on task level, which is essential [12], [5]. It is often natural to express timing constraints in the application requirements as end-to-end constraints.

The compile time steps, illustrated in Fig. 8.1, incorporate a transition from the component based design, to a real-time model enabling existing real-time analysis and mapping to a RTOS. During this step the components are replaced by real-time tasks. Main concerns in this phase are allocation of components to tasks, assignment of task attributes, and real-time analysis. During attribute assignment, run-time attributes that are used by the underlying operating system are assigned to the tasks. The attributes are determined so that the high level constraints specified by the developer during the design step are met. Finally, when meeting the constraints of the system, a synthesis step is executed. It is within this step the binary representation of the system is created, often the operating system and run-time system are also included with the application code in a single bundle

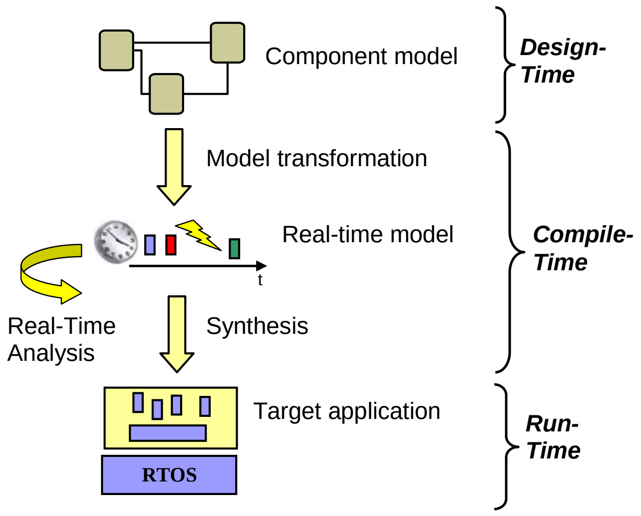


Figure 8.1: The AutoComp component technology

The run-time system is assumed to be a traditional RTOS with Fixed Priority Scheduling (FPS) of tasks. Most commercial RTOS can be classified into this category; furthermore they are simple, resource efficient and many real-time analysis techniques exist. In some cases a layer providing run-time support for the tasks has to be implemented in order to fully support FPS models used in real-time theory.

8.3 Component Model

Vehicles present a heterogeneous environment where the interaction between the computer system and the vehicle take different forms. Some vehicle functionality requires periodic execution of software, e.g., feedback control, whereas other functionality has a sporadic nature, e.g., alarms. Although vehicle control plays a central role, there is also an abundance of other functionality in vehicles that is less critical and has other characteristics, e.g., requires more flexibility. Although less critical, many of these functions will still interact with other more critical parts of the control system, consider for example diagnostics. We present a model that in a seamless way allows the integration of different functionality, by supporting early specification of the high level temporal constraints that a given functionality has to meet. Moreover, the computational model is based on a data flow style that results in simple application descriptions and system implementations that are relatively straightforward to analyse and verify. The data flow style is commonly used within the embedded systems domain, e.g., in IEC 61131 used for automation [13] and in Simulink used for control modelling [14].

The definition of the AutoComp component model is divided into components, component interfaces, composition, the components invocation cycle, transactions and system representation. In Fig. 8.2 the component model is illustrated using UML2, which could be a possible graphical representation during design.

The components are defined as *glass box*, meaning that a developer can see the code of a component for introspection purposes. It does not mean that a developer has to look into a component during normal composition, and not that it is allowed to modify a component. The introspection possibility is a requirement during verification of safety critical applications in order to gain complete knowledge about components behaviour. Furthermore, the components can only exchange data with each others through data ports. A component can be a composite containing a complete subsystem, or a basic component with an entry function. Composite components can be treated as any other component during composition, but it is also possible to enter a composite and change timing requirements and other properties. The entry function provided by non-composite components can be compared to the entry function for a computer program, meaning that the contained number of functions of the component can be arbitrary.

The interfaces offered by a component can be grouped into the two classes data and control interfaces. The data interfaces are used to specify the data flow between components, and consist of data ports. Data ports have a specified type and can be either provided or required. Provided ports are the ports provided by components for input, i.e., the ports a component reads data from. Required ports are the ports a component writes data to. A component also has a control interface with a mandatory control sink, and an optional control source. The control interface is used for specifying the control flow in the application, i.e., when or as a response to what component should be triggered. The control sink is used for triggering the functionality inside the component, while the control source is used for triggering other components.

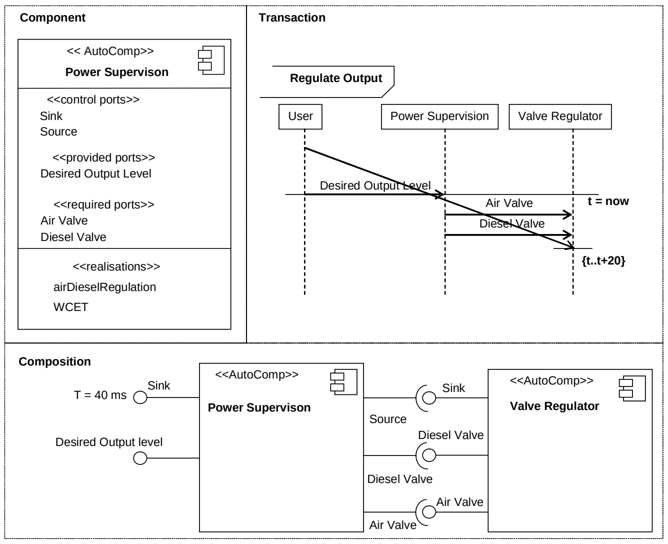


Figure 8.2: In the upper left part of the figure there is a UML 2 component diagram for modelling of a component. The lower part of the figure is a composition diagram showing a composition of two components. Finally the upper right part of the figure is a sequence diagram with a timing constraint that is used to express the end-to-end deadline for a transaction

During composition the developer has three main techniques to work with. The data flow is specified through connection of provided and required data ports. The rules are as follows; required ports must be wired to provided ports with a compatible type. It is possible to make abstractions through definition of composite components. Composite components can be powerful abstractions for visualizing and understanding a complex system, as well as they provide larger units of reuse. The control flow is specified through binding the control sinks to period times for periodic invocation, to external events for event invocation, or to control sources of other components for invocation upon completion of the other components.

A components invocation cycle can be explained as in the following sentences. Upon stimuli on the control sink, in form of an event from a timer, an external source or another component; the component is invoked. The execution begins with reading the provided ports. Then the component executes the contained code. During the execution, the component can use data from the provided ports and write to the required ports as desired, but the writes will only have local effect. In the last phase written data become visible on the required ports, and if the control source in the control interface is present and wired to the control sink of another component stimulus is generated.

Transactions allow developers to define and set end-to-end timing constraints on activities involving several components. A transaction in AutoComp can be defined as:

A transaction Tr_i is defined by a tuple $< C, D, J_s, J_c >$ where:

C - represent an ordered sequence of components;

D - represent the end-to-end deadline of the transaction;

J_s - represent the constraint on start jitter of the transaction;

J_c - represent the constraint on completion jitter of the transaction.

The end-to-end deadline is the latest point in time when the transaction must be completed, relative to its activation. Jitter requirements are optional and can be specified for transactions involving time triggered components. Start jitter is a constraint of the periodicity of the transactions starting point, while completion jitter is a constraint on the periodicity of a transactions completion point. Both types of jitter are expressed as a maximum allowed deviation from the nominal period time. A restriction, necessary for real-time analysis, is that components directly triggered by an external event can only be part of a transaction as the first component.

A system can be described with the UML class diagram in Fig. 8.3. A system is composed of one or several components, each with a data interface, a control interface and a realization as a subsystem or an entry function. A system also has zero or more data couplings, describing a connected pair of required and provided data ports. Furthermore, systems have zero or more control couplings which describe a connected pair of control sink and source. Finally, the last part of a system is zero or more transactions with the included components, an end-to-end deadline and the possibility to specify jitter requirements.

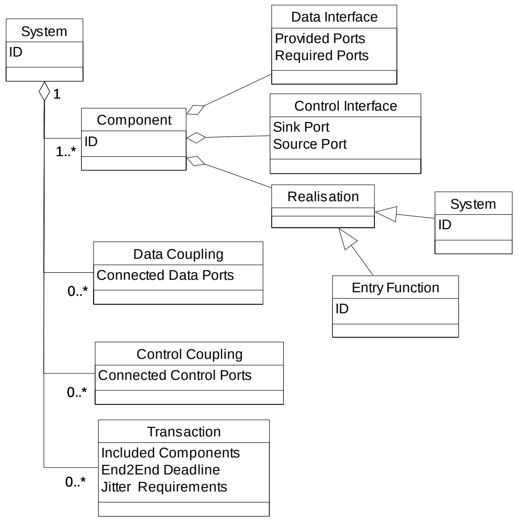


Figure 8.3: UML class diagram showing the static view of the component model

8.4 Model Transformation

Model transformation involves the steps necessary in order to transit from the component model allowing an efficient and powerful design phase, to a run-time model enabling verification of temporal constraints and usage of efficient and deterministic execution environments. As previous stated in section 2 we assume a FPS run-time model. The FPS model defines a system as a set of tasks with the attributes period time, priority, offset, and WCET. Hence, it is necessary to translate the component model with its temporal constraints in to tasks holding these attributes. The translation is performed in two separate steps; the first step is to make a transformation between components and task (task allocation), the second step is to assign attributes to the tasks (attribute assignment). To assign the FPS model attributes in such a way that the high level temporal constraints on transactions are met is non-trivial and has been addressed in research by e.g., [1], [3].

8.4.1 Task Allocation

The easiest approach for task allocation is a one to one relationship between components and tasks, but that is not necessarily optimal. In fact the task allocation step has a lot of different tradeoffs. Such tradeoffs can be found between reliability and run time overhead; few tasks reduce run time overhead at the cost of memory protection (usually at task level) between components. Testability and schedulability are examples of other properties that are affected by the allocation scheme.

In this paper we introduce a task allocation strategy that strives to reduce the number of tasks considering schedulability and reliability. Components are not allocated to the same task if schedulability is obviously negatively affected and structurally unrelated components are not allocated to the same task in order to cater for memory protection and flexibility.

The first step in the allocation process is to convert all composite components to a flat structure of the contained basic components. Secondly the following rules are applied:

1. All instances of components are allocated to separate tasks, Worst Case Execution Time (WCET) is directly inherited from a component to the corresponding task

2. The start jitter Js corresponding to a transaction with jitter requirements is set as a requirement on the task allocated for the first component in the ordered sequence C, while the completion jitter Jc is set to the task allocated for the last component in the sequence

3. Tasks allocated for components with connected pairs of control sink and sources, where the task with the source do not have any jitter requirements, and both tasks are participating in the same and only that transaction are merged. The resulting WCET is an addition from all integrated tasks WCET

4. Tasks allocated for time triggered components that have the same period time, not have any jitter constraints and are in a sequence in the same and only that transaction are merged. The resulting WCET is an addition from all integrated tasks WCET

The situation after application of the allocation rules is a set of real-time tasks. The high level timing requirements are still expressed in transactions, but instead of containing an ordered set of components a transaction now contain an ordered set of tasks. The rest of the attributes, those that cannot be mapped directly from the component model to the real-time model are taken care of in the following attribute assignment step. In Fig. 8.4, given the two transactions $Tr_1 =< C, D, J_s, J_c >=< A, B, C, 60, -, 25 >$ and $Tr_2 =< C, D, J_s, J_c >=< D, E, F, 40, 5, - >$ the task allocation step for the components in Table 8.1 is shown. The resulting task set is in Table 8.2.

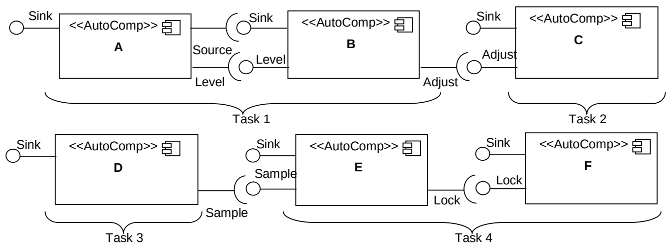


Figure 8.4: Task allocation example

	Sink Bound To	WCET
A	T = 100	5
B	A.Source	10
C	T = 60	5
D	T = 40	5
E	T = 40	6
F	T = 40	9

Table 8.1: A component set

8.4.2 Attribute Assignment

After the components have been assigned to tasks, the tasks must be assigned attributes so that the high level temporal requirements on transactions are met. Attributes that are assigned during task allocation are WCET for all tasks, a period time for periodic tasks and a Minimum Interarrival Time (MINT) for event triggered tasks.

	Trigger	Jitter	WCET
Task 1	T = 100		15
Task 2	T = 60	25	5
Task 3	T = 40	5	5
Task 4	T = 40		15

Table 8.2: The resulting task set

The scheduling model that is used throughout this paper is FPS, where tasks have their priorities and offsets assigned using an arbitrary task attribute assignment methodology. Examples of existing methods that can be used for priority assignment are Bate and Burns [1], Sandström and Norström [3] or by combination of Yerraballi [15] or Cheng and Agrawala [16] with Dobrin, Fohler and Puschner [17]. In this paper it is assumed that task attributes are assigned using the algorithm proposed by Bate and Burns [1], and it is showed that the component model described in this paper is applicable to their analysis model. Weather the tasks are time triggered or event triggered is not considered in the Bate and Burns analysis but is required during the mapping to the FPS model, where periodic and event triggered (sporadic) tasks are separated. The attributes that are relevant, considering this work, in the Bate and Burns approach are listed below.

For tasks:

T (Period) - All periodic tasks have a period time that is assigned during the task allocation. Sporadic tasks have a MINT that analytically can be seen as a period time;

J (Jitter) - The jitter constraints for a task is the allowed variation of task completion from precise periodicity. This type of jitter constraint is known as completion jitter. Jitter constraints can be set on the first and last task in a transaction;

R (Worst Case Response time) - The initial Worst Case Response time for a task is the WCET for the task, i.e., the longest time for a task to finish execution from its starting point in time.

For transactions:

T (Period) - The period of a transaction is the least common multiple of the period times of the participating tasks of the transaction;

End-to-End deadline - Transactions have a requirement that all tasks have finished their execution within a certain time from the transactions point of start in time.

In Bate and Burns approach additional attributes, such as deadline and separation for tasks and jitter requirements for transactions are considered. In this paper those attributes are disregarded since there are no such requirements in the previously described component model. It is trivial to see that from the component model, the period and jitter constraints match the model proposed by Bate and Burns. The initial worst case response time R is assigned the WCET value in the component model. For the transaction the end-to-end deadline requirements match the transaction deadline of the Bate and Burns model. The period time of the transaction is derived from the least common multiple of the period of the tasks participating in the transaction.

The next step is naturally to assign the FPS model with run-time and analysis attributes. The new attributes priority and offsets will be derived through existing analysis methods [1]. The new parameters for the FPS model are described below.

P (Priority) - The priority is an attribute that indicates the importance of the task relative to other tasks in the system. In a FPS system tasks are scheduled according to their priority, the task with the highest priority is always executed first. All tasks in the system are assigned a priority;

O (Offset) - The offset is an attribute that periodic tasks with jitter constraints are assigned. The earliest start time is derived by adding the offset to the period time.

In Table 8.3 it is summarized what attributes belonging to time triggered and event triggered tasks in the FPS model.

Attribute	Time triggered	Event triggered
Period	X	
MINT		X
Priority	X	X
Offset	X (Upon Jitter Constraints)	
WCET	X	X

Table 8.3: Attributes associated with time and event triggered tasks

Applying the Bate and Burns algorithm determines task attributes from the tasks and transactions described in Table 8.2. The resulting run-time attributes priority, offset period and WCET are shown in Table 8.4. The attributes offset and priority are determined with the Bate and Burns analysis, whilst the period and WCET are determined in the task allocation.

	Priority	Offset	Period	WCET
Task 1	2	0	100	15
Task 2	1 (Lowest)	35	60	5
Task 3	4 (Highest)	0	40	5
Task 4	3	0	40	15

Table 8.4: Assigned task attributes

In Fig. 8.5 a run-time trace for an FPS system is shown and the transactions Tr_1 and Tr_2 are indicated.

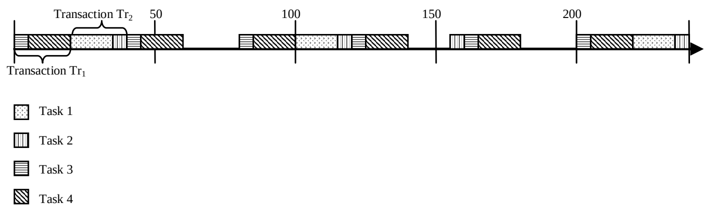


Figure 8.5: Trace of an FPS schedule

When the FPS model has been assigned its attributes it has to be verified. The verification of the model is performed by applying real-time scheduling analysis to confirm that the model is schedulable with the assigned parameters. This is necessary since attribute assignment does not necessarily guarantee schedulability, but only assigns attributes considering the relation between the tasks.

8.4.3 Real-Time Analysis

To show that the FPS tasks will meet their stipulated timing constraints, schedulability analysis must be performed. Much research has been done with respect

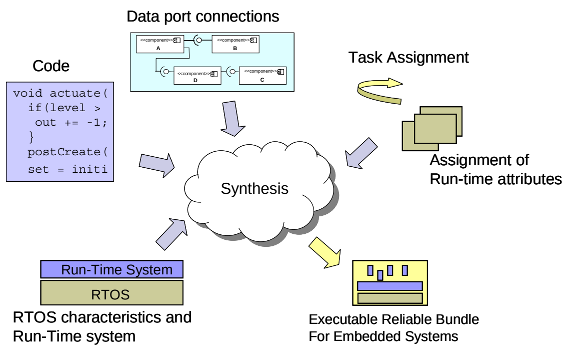


Figure 8.6: The steps of synthesizing code for the run-time system

to analysis of different properties of FPS systems, and all those results are available for use, once a FPS model has been established. The temporal analysis of an FPS system with offsets, sporadic tasks and synchronization has been covered in research by e.g., Palencia et al. [18], [19] and Redell [20].

The output from the analysis is whether the system is feasible or not in the worst case. If the analysis shows that the system is infeasible, the parts that can not keep its requirements are either changed and reanalysed or emphasised for the developer to make changes.

8.5 Synthesis

The next step after the model transformation and real-time analysis is to synthesise code for the run-time system. This includes mapping the tasks to operating system specific task entities, mapping data connections to an OS specific communication, modifying the middleware, generating glue code, compiling, linking and bundling the program code (see Fig. 8.6).

The synthesis is divided into two major parts. Given a task set and necessary information about the run-time system, the synthesis generates code considering communication, synchronization.

- The first part in synthesis is to resolve the communication within and between tasks. Two communicating components that are assigned to different tasks will form an Inter Task Communication (ITC) while com-

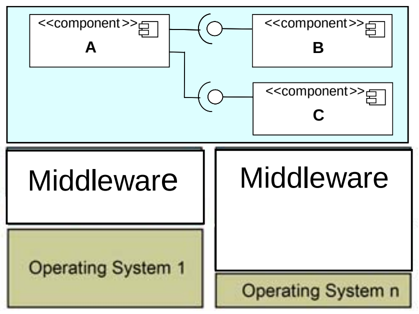


Figure 8.7: A component model with adjustments for different operating systems to promote platform independence

munication between components assigned to the same task are realized with shared data spaces within the task. The ITC is later mapped to operating system specific communication directives.

- The other part in the synthesis is to resolve the control couplings, i.e., the sink and source. If a tasks starting point is dependent on the former tasks finishing point the tasks have to be synchronized. The synchronization is solved through scheduling. The synthesis will generate code for scheduling periodic tasks, handle the control flow between tasks and consider offsets. The code generated for the periodic scheduling and offsets is dependent on the middleware and can be realized as a configuration file or actual code in each task. Invocations of sporadic tasks are mapped to event handlers in the middleware or the operating system.

It is assumed that a middleware is present as shown in Fig. 9.3, for each platform and that it provides functionality that the component model needs but the operating system does not provide. The more functionality the operating system provides, the smaller the middleware has to be. The middleware encapsulates core communication and concurrency services to eliminate many non-portable aspects of developing and is hence platform specific in favour of a platform independent component model. Typical functionality that is not provided by most commercial RTOS is periodicity and support for offsets. The middleware also need to support sink and source couplings since task coupled with its source need to be able to invoke the corresponding task. The run-time system conforms to FPS and hence the run-time task model is similar to the previously described FPS model with some exceptions. The worst case execution time is merely an analysis attribute and is not needed in the run-time model. The MINT is usually a requirement on the environment rather than a task attribute, and is thus also analytical and unnecessary. Hence the run-time task model is for periodic tasks Period time, Priority, Offset and for sporadic tasks Priority.

8.6 Conclusions and Future Work

In this paper we show how to use component based software engineering for low footprint systems with very high demands on safe and reliable behaviour. The key concept is to provide expressive design time models and yet resource effective run-time models by statically resolve resource usage and timing by powerful compile time techniques.

The work presented in this paper introduces a component technology for resource effective and temporally verified mapping of a component model to a resource structure such as a commercial RTOS. This is made possible by introduction of a component model that support specification of high level real-time constraints, by presenting a mapping to a real-time model, permitting use of standard real-time theory, and by synthesis of run-time mechanisms for predictable execution according to the temporal specification in the component model.

Although the basic concept has been validated by successful industrial application of previous work [5], it is necessary to further validate the component technology presented here. In order to facilitate this, a prototype implementation of the component technology is under development where the core part has been completed. The prototype will enable evaluation of different technology realisations with respect to performance. Moreover, parts of the model transformation need additional attention, foremost the strategies for allocation of components to tasks. Furthermore, we will make efforts in extending the component model making it more expressive and flexible while still keeping the ability for real-time analysis. Interesting is also to investigate trade-offs between run-time foot print and flexibility with respect to e.g., adding functionality post production. Finally, the component technology will be evaluated in a larger, preferably industrial, case.

Bibliography

[1] A. Bate and I. Burns. An approach to task attribute assignment for uniprocessor systems. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*. IEEE, 2002.

[2] K. Mok, D. Tsou, and R. C. M. De Rooij. The msp.rtl real-time scheduler synthesis tool. In *In Proc. 17th IEEE Real-Time Systems Symposium*, pages 118–128. IEEE, 1996.

[3] K. Sandström and C. Norström. Managing complex temporal requirements in real-time control systems. In *In 9th IEEE Conference on Engineering of Computer-Based Systems Sweden*. IEEE, April 2002.

[4] J. Würtz and K. Schild. Scheduling of time-triggered real-time systems. In *In Constraints, Kluwer Academic Publishers*, pages 335–357, October 2000.

[5] 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.

[6] Arcticus. Arcticus homepage: http://www.arcticus.se.

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

[8] G. Nierstrasz, S. Arevalo, R. Ducasse, A. Wuyts, P. Black, C. Müller, T. Zeidler, R. Genssler, and A. van den Born. Component model for field

devices. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment, Germany*, June 2002.

[9] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. In *IEEE Transactions on Software Engineering*, pages 759–776. IEEE, December 1997.

[10] W. H. Schmidt and R. H. Reussner. Parameterised contracts and adaptor synthesis. In *Proc. 5th International Workshop of Component-Based Software Engineering (CBSE5)*, May 2002.

[11] S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau. Packaging predictable assem-bly with prediction-enabled component technology. Technical report, November 2001.

[12] D .K . Hammer and M. R. V. Chaudron. Component-based software engineering for resource constraint systems: What are the needs? In *6th Workshop on Object-Oriented Real-Time Dependable Systems, Rome, Italy*, January 2001.

[13] IEC. International standard IEC 1131: Programmable controllers, 1992.

[14] Mathworks. Mathworks homepage : http://www.mathworks.com.

[15] R. Yerraballi. *Scalability in Real-Time Systems.* PhD thesis, Computer Science Department, old Dominion University, August 1996.

[16] S. T. Cheng. and Agrawala A. K. Allocation and scheduling of real-time periodic tasks with relative timing constraints. In *Second International Workshop on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1995.

[17] R. Dobrin, G. Fohler, and P. Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. In *In Real-Time Systems Symposium London, UK, December*, 2001.

[18] J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proc. of the 19th Real-Time Systems Symposium*, December 1998.

[19] J. C. Palencia and M. Gonzalez Harbour. Exploiting precedence relations in the schedulabil-ity analysis of distributed real-time systems. In *Proc. of the 20th Real-Time Systems Symposium*, December 1999.

[20] O. Redell and M. Törngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proc. Eighth IEEE Real-Time and Embedded Tech-nology and Applications Symposium*. IEEE, September 2002.

Chapter 9

Paper E: 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.

9.1 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 9.3), together with an example application that illustrates its use (Section 9.4). Based on experiences with the example application, we provide an evaluation of the technology (Section 9.5).

9.2 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].

9.3 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

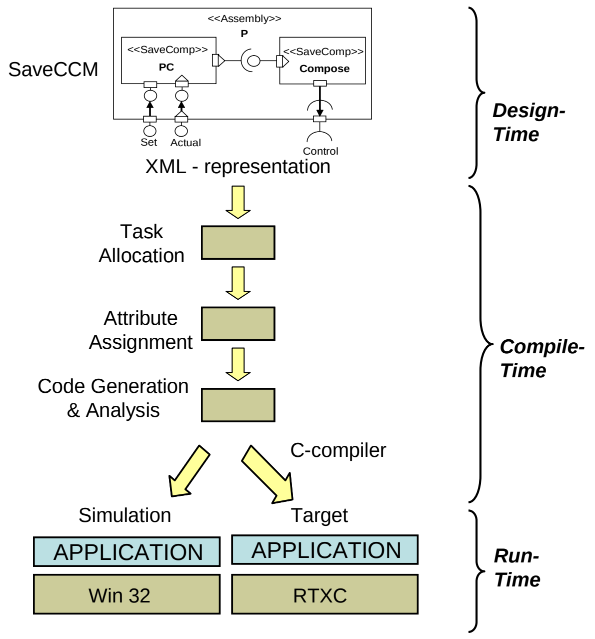


Figure 9.1: An overview of our current component technology

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 9.1, 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.

9.3.1 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 9.2, 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/

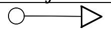
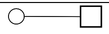
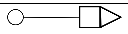
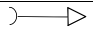
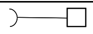
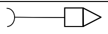
Symbol	Interpretation
	**Input port** - with triggering only
	**Input port** - with data only
	**Input port** – combined with data and triggering
	**Output port** - with triggering
	**Output port** - with data
	**Output port** - combined with data and triggering
<<SaveComp>> **<name>**	**Component** - 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 9.2: Graphical syntax of SaveCCM

9.3.2 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 9.1) 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 9.3 right) and the Win32 operating system (see Figure 9.3 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.

9.3.3 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.

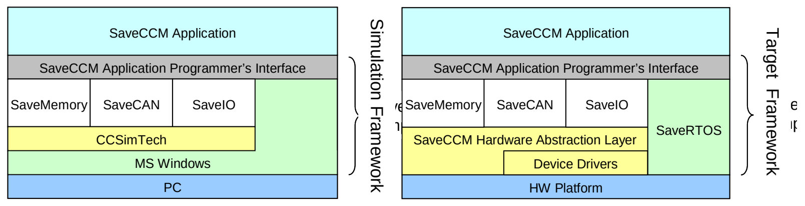


Figure 9.3: System architecture for simulation and target

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.

9.4 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.

9.4.1 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.

9.4.2 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 9.4) 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 9.4, right.

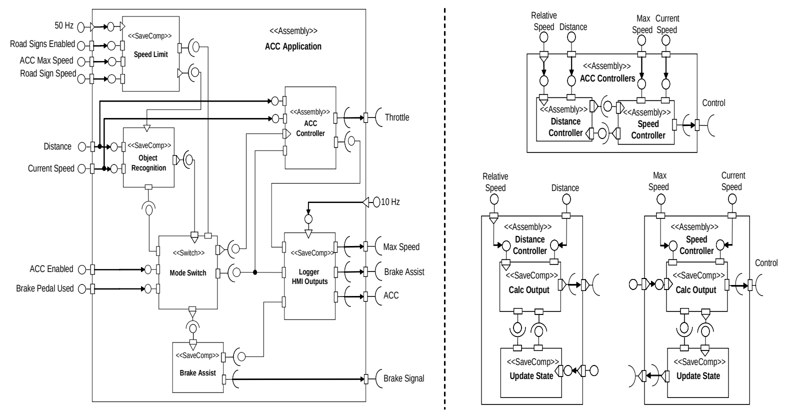


Figure 9.4: ACC Application implementation

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 9.4, 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].

9.4.3 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.

9.5 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].

9.5.1 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.

9.5.2 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.

9.5.3 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.

9.6 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 *Proceedings of 30th Euromicro Conference, Special Session Component Models for Dependable Systems*, 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.