

Mälardalen University Licentiate Thesis  
No.42

# Software Component Technologies for Heavy Vehicles

Anders Möller

January 2005



**MÄLARDALEN UNIVERSITY**

Department of Computer Science and Electronics  
Mälardalen University  
Västerås, Sweden

Copyright © Anders Möller, 2005  
ISSN 1651-9256  
ISBN 91-88834-88-3  
Printed by Arkitektkopia, Västerås, Sweden  
Distribution by Mälardalen University Press

# Abstract

Control-systems for heavy vehicles have advanced from an area where mainly mechanic and hydraulic solutions were used, to a highly computerised domain using distributed embedded real-time computer systems.

To cope with the increasing level of end-customer demands on advanced features and functions in future vehicle systems, sophisticated development techniques are needed. The development techniques must support software in numerous configurations and facilitate development of systems with requirements on advanced functionality, timeliness, and safety-criticality. In order to meet these requirements, we propose the use of component-based software engineering.

However, the software component-technologies available on the market have not yet been generally accepted by the vehicular industry. In order to better understand why this is the case, we have conducted a survey – identifying the industrial requirements that are deemed decisive for introducing a component technology. We have used these requirements to evaluate a number of existing component technologies, and one of our conclusions is that none of the studied technologies is a perfect match for the industrial requirements.

In addition, we have implemented and evaluated the novel component model SaveCCM, which has been designed for safety-critical automotive applications. Our evaluation indicates that SaveCCM is a promising technology which has the potential to fulfil the industrial requirements. However, tools are still immature and incomplete.

In the final part of this work, we propose the use of monitored software components, as a general approach for engineering of embedded systems. In our approach, a component's execution is continuously monitored and experience regarding the behaviour is accumulated. As more and more experience is collected the confidence in the component grows.



# Preface

The work presented in this thesis has been performed within the HEAVE (A Component Technology for Heavy Vehicles) project. The work has been supported by CC Systems, Volvo Construction Equipment, and by the KK Foundation, and has been accomplished at Mälardalen Real-Time Research Centre, Mälardalen University, Sweden.

Firstly, I would like to thank Dr. Mikael Nolin for extraordinary supervision and for turning my confused thoughts into publishable research papers. Secondly, I would like to thank Jörgen Hansson at CC Systems and Prof. Hans Hansson at Mälardalen Real-Time Research Centre for making this research project possible.

I owe my co-authors (especially Mikael Åkerholm, Joakim Fröberg, Daniel Sundmark, and Johan Fredriksson) many thanks for helping me realising research ideas and for sharing memorable conference trips all around the globe.

Also, many thanks to Nils-Erik Bänkestad and Robert Larsson at Volvo Construction Equipment for fruitful research discussions, and for their support during my stay at Volvo.

Finally, thanks to my colleagues, both at the department at Mälardalen University and at CC Systems, and to my friends and beloveds for making life great fun. After all, that is what it is all about!

Anders Möller  
Västerås, January 10, 2005



# List of Publications

## Publications Included in This Licentiate Thesis

- Paper A:** *Industrial Requirements on Component Technologies for Embedded Systems*; Anders Möller, Joakim Fröberg and Mikael Nolin; In Proceedings of the 7<sup>th</sup> International Symposium on Component-Based Software Engineering, pages 146–161, Edinburgh, Scotland, May 2004.
- Paper B :** *Evaluation of Component Technologies with Respect to Industrial Requirements*; Anders Möller, Mikael Åkerholm, Johan Fredriksson and Mikalel Nolin; In Proceedings of the 30<sup>th</sup> Euromicro Conference, Component-Based Software Engineering Track, pages 56–63; Rennes, France, September 2004.
- Paper C:** *Towards a Dependable Component Technology for Embedded System Applications*; Mikael Åkerholm, Anders Möller, Hans Hansson and Mikael Nolin; To Appear in the Proceedings of the Workshop on Object-Oriented Real-time Dependable Systems, Sedona, Arizona, USA, February 2005.
- Paper D:** *Monitored Software Components – A Novel Software Engineering Approach*; Daniel Sundmark, Anders Möller and Mikael Nolin; In Proceedings of the 11<sup>th</sup> Asian-Pasific Conference on Software Engineering, Workshop on Software Architectures and Component Technologies, pages 624–631; Busan, Korea; November 2004.

## Other Publications by the Author

### Journals

- *A Simulation Technology for CAN-based Systems*; Anders Möller and Per Åberg, CAN Newsletter, nr 4, CAN in Automation, December 2004.

### Conferences and Workshops

- *Developing and Testing Distributed CAN-based Real-Time Control-Systems in a single PC, – An Industrial Experience Paper*; Anders Möller, Per Åberg, Fredrik Löwenhielm, Jakob Engblom and Jörgen Hansson; To Appear in the Proceedings of the International CAN Conference, CAN in Automation; Roma, Italy, February 2005.
- *Software Component Technologies for Real-Time Systems – An Industrial Perspective*; Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin; In Proceedings of the Work-in-Progress Session of the 24<sup>th</sup> IEEE Real-Time System Symposium (RTSS), Cancun, Mexico, December 2003.
- *Using Components to Facilitate Stochastic Schedulability Analysis*; Thomas Nolte, Anders Möller, Mikael Nolin; In Proceedings of the Work-In-Progress Session of the 24<sup>th</sup> IEEE Real-Time Systems Symposium (RTSS), Cancun, Mexico, December 2003.
- *What are the needs for components in vehicular systems? – An industrial perspective*; Anders Möller, Joakim Fröberg and Mikael Nolin; In Real-Time in Sweden (RTiS), Västerås, Sweden, August, 2003.
- *What are the needs for components in vehicular systems? – An industrial perspective*; Anders Möller, Joakim Fröberg and Mikael Nolin; In Proceedings of the Work-in-Progress Session of the 15<sup>th</sup> Euromicro Conference on Real-Time Systems, Porto, Portugal, July 2003.



## Technical Reports

- *SAVEComp - a Dependable Component Technology for Embedded Systems Software*, Mikael Åkerholm, Anders Möller, Hans Hansson and Mikael Nolin, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-165-/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, December 2004.
- *Predictable Assemblies using Monitored Software Components*; Daniel Sundmark, Anders Möller, Mikael Nolin; MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-160/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden, February 2004.
- *An Industrial Evaluation of Component Technologies for Embedded Systems*; Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin; MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-150/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden, February 2004
- *Requirements on Component Technologies for Heavy Vehicles*; Anders Möller, Joakim Fröberg, Mikael Nolin; MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-150/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden, January 2004.
- *Component Based Software Engineering for Embedded Systems – A literature Survey*; Mikael Nolin, Johan Fredriksson, Jerker Hammarberg, Joel Huselius, John Håkansson, Annika Karlsson, Ola Larses, Markus Lindgren, Goran Mustapic, Anders Möller, Thomas Nolte, Jonas Norberg, Dag Nyström, Aleksandra Tesanovic, and Mikael Åkerholm; MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-104/203-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden, June 2003.



# Contents

<b>List of Publications</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	2
1.1.1 CBSE for Embedded Systems . . . . .	2
1.1.2 Heavy Vehicle Systems . . . . .	6
1.2 Motivation . . . . .	8
1.3 Thesis Outline . . . . .	10
<b>2 Contribution</b>	<b>13</b>
2.1 Research Questions . . . . .	13
2.2 Contribution . . . . .	15
2.3 Included Papers . . . . .	16
<b>3 Research Work and Method</b>	<b>19</b>
3.1 Preliminary Literature Study . . . . .	20
3.2 Industrial Requirements Case-Study . . . . .	20
3.3 Evaluation of Existing Technologies . . . . .	21
3.4 Implementing and Evaluating a Component Technology . . . . .	22
3.5 Monitoring Software Components . . . . .	22
<b>4 Conclusion and Future Work</b>	<b>25</b>
<b>Bibliography</b>	<b>27</b>
<b>5 Paper A:</b>	

<b>Industrial Requirements on Component Technologies for Embedded Systems</b>	<b>33</b>
5.1 Introduction . . . . .	35
5.2 Introducing CBSE in the Vehicular Industry . . . . .	36
5.3 A Component Technology for Heavy Vehicles . . . . .	38
5.3.1 The Business Segment of Heavy Vehicles . . . . .	38
5.3.2 System Description . . . . .	40
5.4 Requirements on a Component Technology for Heavy Vehicles	43
5.4.1 Technical Requirements . . . . .	44
5.4.2 Development Requirements . . . . .	47
5.4.3 Derived Requirements . . . . .	49
5.4.4 Discussion . . . . .	50
5.5 Conclusions . . . . .	51
Bibliography . . . . .	52
<b>6 Paper B:</b>	
<b>Evaluation of Component Technologies with Respect to Industrial Requirements</b>	<b>57</b>
6.1 Introduction . . . . .	59
6.2 Requirements . . . . .	60
6.2.1 Technical Requirements . . . . .	60
6.2.2 Development Requirements . . . . .	62
6.2.3 Derived Requirements . . . . .	63
6.3 Component Technologies . . . . .	64
6.3.1 PECT . . . . .	65
6.3.2 Koala . . . . .	66
6.3.3 Rubus Component Model . . . . .	67
6.3.4 PBO . . . . .	68
6.3.5 PECOS . . . . .	69
6.3.6 CORBA Based Technologies . . . . .	70
6.4 Summary of Evaluation . . . . .	71
6.5 Conclusion . . . . .	73
Bibliography . . . . .	73
<b>7 Paper C:</b>	
<b>Towards a Dependable Component Technology for Embedded System Applications</b>	<b>77</b>
7.1 Introduction . . . . .	79
7.2 CBSE for Embedded Systems . . . . .	80

7.3	Our Component Technology . . . . .	81
7.3.1	Design-Time - The Component Model . . . . .	83
7.3.2	Compile-Time Activities . . . . .	86
7.3.3	The Run-Time System . . . . .	88
7.4	Application Example . . . . .	89
7.4.1	Introduction to ACC functionality . . . . .	89
7.4.2	Implementation using SaveCCM . . . . .	90
7.4.3	Application Test-Bed Environment . . . . .	92
7.5	Evaluation and Discussion . . . . .	92
7.5.1	Structural Properties . . . . .	93
7.5.2	Behavioural Properties . . . . .	94
7.5.3	Process Related . . . . .	95
7.6	Conclusions and Future Work . . . . .	95
	Bibliography . . . . .	96

**8 Paper D:**

	<b>Monitored Software Components - A Novel Software Engineering Approach</b>	<b>101</b>
8.1	Introduction . . . . .	103
8.2	A Life-Cycle Approach to Component-Based Systems . . . . .	104
8.3	Embedded Systems . . . . .	105
8.3.1	CBSE for Embedded Systems . . . . .	106
8.3.2	Embedded System Example . . . . .	106
8.3.3	Prerequisites for Monitoring Component-Based Embedded Systems . . . . .	107
8.4	Related Work . . . . .	109
8.4.1	Monitoring Techniques for Component-Based Systems . . . . .	109
8.4.2	Monitoring Support in Commercial Component Technologies . . . . .	111
8.5	Monitoring Software Components . . . . .	113
8.5.1	Temporal Behaviour . . . . .	113
8.5.2	Memory Usage . . . . .	114
8.5.3	Event Ordering . . . . .	115
8.5.4	Sanity Check . . . . .	115
8.6	Using Monitored Information . . . . .	116
8.7	Conclusion and Future Work . . . . .	116
	Bibliography . . . . .	117



# Chapter 1

## Introduction

The business segment of heavy vehicles (including, e.g., forest harvesters, rock-drilling equipment, and wheel loaders) has entered a new era, where the traditional mechanic and hydraulic solutions are supplemented with highly sophisticated distributed embedded computer control-systems. Introducing these control-systems facilitates the use of advanced technical functions, such as support for advanced engine-control, built-in diagnostic systems and anti-lock braking systems. The control-systems does also prolong the lifetime of the vehicle, by optimising, e.g., engine- and transmission-control.

Ever increasing end-customer demands on advanced features and functions in future control-systems (e.g., to increase productivity in forest harvesting or at a construction site) require new technical solutions. These demands will call for even more advanced electronic control-systems, comprising electronics and software in numerous configurations and variants, most likely supplied from many different vendors.

However, most embedded system developers are in fact, to a large extent, using monolithic and platform dependent software development techniques, in spite of the fact that this make software systems hard to maintain, upgrade, and modify. In order to introduce the new functionality while at the same time increase control-system reliability and decrease development time and costs – the developers call for improved tools and methods. Using software components, and component-based development, is seen as a promising way to meet the requirements on high functionality, reliability, real-timeliness, and safety criticality while at the same time lower development costs due to an improved development process and improved conditions for reuse.

Within this licentiate thesis, we have investigated the industrial requirements on a component technology from the perspective of the business segment of heavy vehicles. We have also evaluated the state-of-the-art software component technologies with respect to these requirements, and based on the evaluation, and the requirements, implemented a new technology. To be able to predict the run-time behaviour of a component assembly' pre-run-time, we also present an engineering method to collect essential component information by monitoring the system during execution.

## 1.1 Background

This section aims at providing a background to the research in this thesis by describing Component-Based Software Engineering (CBSE) for embedded systems, and by illustrating the industrial settings for the intended domain (i.e. heavy vehicles). For a more general and exhaustive presentation of component-based software engineering, see, e.g., [1, 2, 3].

### 1.1.1 CBSE for Embedded Systems

Component-based software engineering is the area of building system applications as assemblies of pre-fabricated software components. To make component-based development attractive, mature techniques, methodologies, and processes are needed. However, within the embedded system domain, many of these are not mature. Some of the remaining challenges, like the lack of widely adopted component technology standards, unsatisfactory support for extra-functional properties (e.g. timing and memory consumption), and insufficient tools to support the component-based development, are described in [4].

The software components are, of course, at the heart of CBSE, and a component can be defined as a reusable unit of deployment and composition [1] (there are, however, an abundance of more detailed component definitions, e.g. by Szyperski [2]). The components must have well specified interfaces, and should be easy to understand, adapt and deliver. Especially for embedded systems, the components must have well specified resource requirements, as well as a specification of other relevant properties, e.g., timing, reliability, safety, and dependability.

When assembling these components into software systems, a *component model* typically defines the different component types and the interaction schemes for components. Typically, in an embedded system component technology,



the component model also clarifies how different resources are bound to the software components. Based on the component assembly, and the component model, a compiler is usually used to generate the executable software.

Component-based development can be approached from two, conceptually different, points of view; distinguished by whether the components are used as a *design philosophy* independent from any concern for reusing existing components, or seen as *reusable (off-the-shelf) building blocks* used to design and implement a component-based system [5]. Irrespective of whether the developer uses software components as a design philosophy or as reusable off-the-shelf building blocks, efficient development of applications is supported by the component-based strategy (for more details, see Sect. 5.2 and Sect. 5.3). Also, component-based development distinguishes *component development* from *system development*. Component development is the process of creating components that can be used and reused in many applications, and system development is concerned with assembling components into applications that meet the system requirements.

In many cases, software reuse is seen as the main driver for introducing a component-based development approach. Component-based reuse is by many software engineers (and managers) seen as a promising approach to reduce high costs of building complex software. LEGO<sup>1</sup> blocks is often used to describe the component-based design, where different kinds of blocks can be used for constructing an endless variety of structures. However, as, e.g., [6] and [7] points out - the context of use for software components is determined by the software architecture, and for a software project to develop generally reusable components the context of its use must be very well understood. Therefore, (according to [7]) component-based reuse is only possible as a consequence of architecture-based reuse, and this understanding must be shared by software engineers as well as product and project managers.

Also, maintenance is supported by CBSE 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, e.g., 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 Internet/office applications (e.g. Enterprise Java Beans [8], and .NET [9]), but for the domain of

---

<sup>1</sup>LEGO, Home page: <http://www.lego.com/>

embedded systems CBSE has not yet 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 [10, 11], and from academia [12, 13]. However, as Crnkovic points out in [4], there are many more issues to solve before a CBSE discipline for embedded systems can be established.

Component technologies are a concrete implementation of a component model and a component framework, and can be used for building component-based applications. To assemble the components into software systems, different activities are performed, and the central technical concepts and activities for a typical embedded system setting, as approached within our research, are summarised in Fig.1.1, and further described in the remainder of this chapter.

**Design-Time** actions ((1.1) in Fig. 1.1) comprise putting the software components together into a component assembly (i.e. an application). This is the industrial engineering phase of the component-based development process, and building with LEGO blocks often serve as a metaphor for describing the component-based software design. The components are assembled based on the component interface, which can be defined as a specification of its access point [2], and based on rules of the components interaction. These rules are specified within the component model, and do usually define the different component types and the interaction schemes between the components. In a typical embedded system component technology, the component model also clarifies how different resources are bound to the specific components. The rules defined within the component model should also impose that systems built from the components are predictable with respect to important properties in the intended domain (e.g. timing and memory attributes).

**Compile-Time** activities ((1.2) in Fig. 1.1) for an embedded system component technology typically include support for transferring the component assembly (i.e. the application) into an intermediate compile-time model. These activities provide algorithms for synthesis of the high level models into attributes of the run-time model, e.g., operating system calls, task attributes, and real-time constraints. The compile-time activities usually include task allocation ((1.2.1) in Fig. 1.1), attribute assignment ((1.2.2)), and code generation and analysis ((1.2.3)). For more details of the different compile-time activities, see Sect. 7.3.2 of this thesis. For CBSE to be attractive for the embedded system industry this phase should, to the

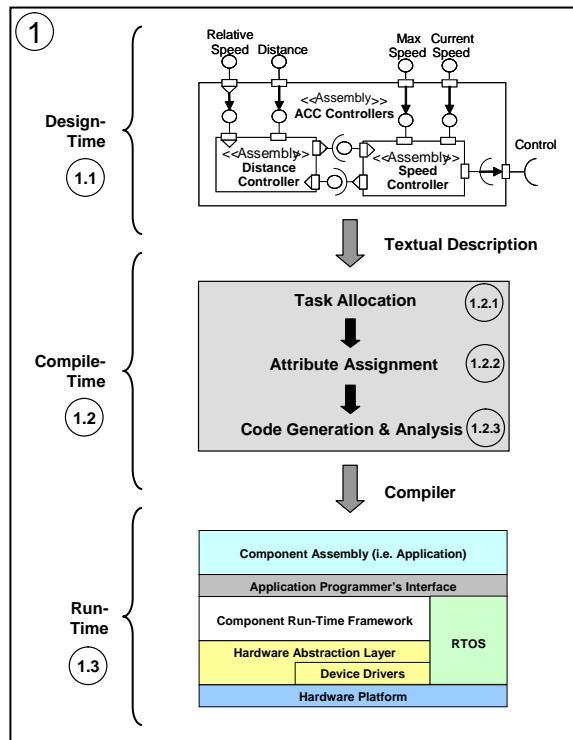


Figure 1.1: Overview of a component technology suitable for embedded systems

highest degree possible, be automated – and tools and mature methods must be provided to the software engineers [4].

**Run-Time** activities ((1.3) in Fig. 1.1) include the compiled component assembly, a run-time component framework, and typically an operating system and a set of device drivers. The component framework 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 (compared to, e.g., [14]) from the run-time system to the pre-run-time compile stages.

### 1.1.2 Heavy Vehicle Systems

Our industrial partners, CC Systems<sup>2</sup> and Volvo Construction Equipment<sup>3</sup>, develop control-systems for heavy vehicles (like, e.g., wheel loaders, forest harvesters, articulated haulers, and rock-drilling equipment). These systems are typically built to endure rough physical environments, and are characterised by safety criticality, advanced functionality, and the requirements on robustness and availability are high. The control-systems are typically dependable distributed embedded real-time systems, which must perform in an extreme physical environment with limited hardware resources.

Control-systems within the business segment of heavy vehicles are, compared to, e.g., passenger cars, often less complex (a short presentation of a typical heavy vehicle system is available in Paper A, Sect. 5.3, of this thesis, and a more detailed description can be found in [15]). The systems are usually built up from a set of electronic control units communicating via, one or more, Controller Area Networks [16], and is typically used for feedback control, discrete control, diagnostics and service, infotainment, and telematics [15].

The product volumes of heavy vehicles are rather moderate (typically in the range of thousands per year), compared to those of passenger cars (in the order of millions per year). Also, customers tend to be more demanding with respect to the technical specification (e.g., engine torque and payload) of the vehicles, and less demanding with respect to design, feel, and look. This causes a lower emphasis on product cost and optimisation of control-systems, compared to automotive industry in general. The lower volumes, and relatively small number of customers, also make the manufacturers more willing to design vehicle variants to meet customer specific requests [15].

Companies developing control-systems for heavy vehicles are challenged by demands on shorter development time along with minimised electronics and software costs, while at the same time having to support increasing customer demands of vehicle features and functions, high demands on reliability and a need to support many configurations, variants and suppliers.

---

<sup>2</sup>CC Systems, Home page: <http://www.cc-systems.com>

<sup>3</sup>Volvo Construction Equipment, Home page: <http://www.volvoce.com>

### Industrial Partners

The work presented in this licentiate thesis is performed in cooperation with CC Systems and Volvo Construction Equipment. These two companies represent different types of actors in the heavy vehicle industry. CC Systems acts as a sub-contractor developing both electronics and software whereas Volvo Construction Equipment is an Original Equipment Manufacturer (OEM) developing the main part of the vehicle in-house. The companies' knowledge and experiences from using software components, and component-based development, is also different.

- CC Systems is developing and supplying distributed embedded real-time control-systems for mobile applications, like, e.g., forest harvesters<sup>4</sup>, rock-drilling equipment<sup>5</sup>, and combat-vehicles<sup>6</sup>.

CC Systems' goal is to use a component-based approach towards software construction, to enhance the ability to reuse and analyse applications, and because it increases predictability by reducing the degrees of freedom for application developers. This reduction of freedom, in turn, will minimise the risk for software errors, since component assembly can only be done in a predefined manner. CC Systems has not yet launched the use of a component technology for embedded systems, but by participating in this research – they wishes to strengthen their knowledge about CBSE.

- Volvo Construction Equipment is one of the world's major manufacturers of construction equipment, with a product range encompassing wheel loaders, excavators, motor graders, and more. The products vary from moderately small compact equipment (1.4 tons) all the way up to huge construction equipment (52 tons) [15].

To accommodate reuse of software components and methodology between products, Volvo Construction Equipment has incorporated a component model for the real-time application domain [10]. However, they wish to strengthen their competence in component-based development in general. The results from this research project will be used to extend their current practices within CBSE.

---

<sup>4</sup>Timberjack, Home page: <http://www.timberjack.com/>

<sup>5</sup>Atlas Copco, Home page: <http://www.atlascopco.com/>

<sup>6</sup>Land Systems Hägglunds, Home page: <http://www.haggeve.com/>

## 1.2 Motivation

We are surrounded by computers. The majority of these computers are not the ones we immediately think of, i.e. desktop- or laptop-computers. In fact, more than 99.8% [17] of the total number of central processing units (CPUs) produced today are embedded into other products than personal computers. The applications of embedded computers range all the way from passenger cars and consumer electronics down to small gadgets and toys.

Most OEMs, developing these embedded systems, face challenges of increased customer-demands on functionality and features, while at the same time having to meet customer expectations, based on the market competitiveness, on reduced costs. To facilitate the increased demands on functionality, more and more electronics and software are introduced. In, e.g., BMW's<sup>7</sup> new 7-series luxury cars there are more than 65 ECUs (and [18] indicates that more than half of the total development cost constitutes development of electronics and software). In the Volvo XC90 (introduced in 2002), the maximum configuration contains about 40 ECUs [15] connected via two Controller Area Networks [16], one MOST ring [19] and a set of Local Interconnect Networks [20]. And – most astounding – a kid's PlayStation 2<sup>8</sup> has more computer power than NASA<sup>9</sup> had for its moon landings [17]

Today, within the embedded system market, software is often seen as *the* way to provide the required functionality in short time and at a reasonable price. And, according to Moore's law<sup>10</sup> hardware is getting cheaper, still offering more and more processing power. Hence, software constitute a growing part of the total development costs, see Fig. 1.2 on the facing page, [21].

In response to this fact, industry calls for immediate improvement of software development methods and tools. Software components and component-based development is by industry, as well as by academia, seen as a promising way to address these issues. Component-based software engineering is a method that supports software reuse, fast development, enhanced software integration support, more flexible configurations, and good reliability predictions of component assemblies [1].

During the last decade, the Internet-/office-oriented software community

---

<sup>7</sup>BMW, Home Page: <http://www.bmw.com>

<sup>8</sup>Sony PlayStation 2, Home Page: <http://www.sony.com>

<sup>9</sup>National Aeronautics and Space Administration, NASA, Home Page: <http://www.nasa.gov>

<sup>10</sup>Moore observed an exponential growth in the number of transistors per integrated circuit and predicted that this trend would continue. Through the processor developers relentless technology advances, Moore's law – the doubling of transistors every couple of years – has been maintained, and still holds true today.

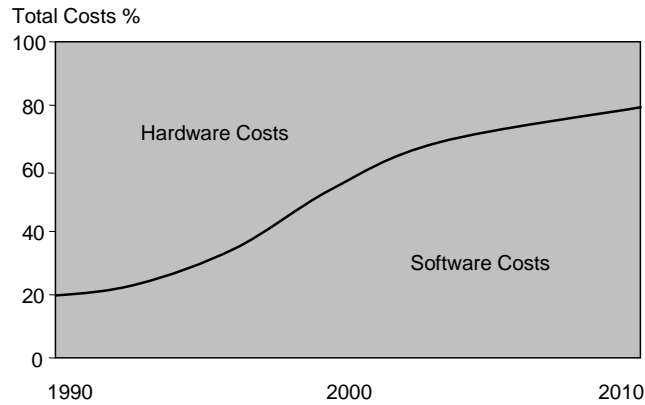


Figure 1.2: Estimation of the distribution between hardware and software development cost [21]

has proven that CBSE is a method with great potential, and the community has achieved remarkable progress with software components, and component-based design. Today, it is possible to download components on the fly and have them integrated, and executed, within the context of another program (such a web browser or a word processor). When developing, e.g., Internet applications today, it is possible to purchase off-the-shelf components and combine them into assemblies forming new software products. Technologies like, e.g., CORBA [14], Enterprise Java Beans [8], and .NET [9] are frequently used to build applications out of software components. However, these component technologies are not applicable to most embedded real-time computer systems, due to extensive memory usage and unsatisfactory timing behaviour.

Some attempts have been made to adapt Internet/office component technologies to embedded systems (like, e.g., minimumCORBA [22]). However, these adaptations have not been generally accepted by the embedded system developers, and the reason for this failure is mainly the diversified nature of the embedded system market. Different market segments have different requirements on a component technology, and often these requirements are not fulfilled simply by stripping down existing component technologies.

There are also some international cooperation on standardising software middleware for vehicular systems that might (and probably will) have influ-

ence on future control-systems for heavy vehicles, like, e.g., the EAST-EEA [23] project and the AUTOSAR [24] consortium. The, nowadays completed, EAST-EEA project was aiming to enable proper electronic integration through definition of an open architecture allowing hardware and software interoperability, and was the predecessor to AUTOSAR.

AUTOSAR is a consortium working to establish an open standard for the automotive electrical engineering architecture serving as a basic infrastructure for the management of functions within both future applications and standard software modules.

It is important to follow this standardising work in order to be prepared for the future system design philosophy. Hence, a component technology as suggested within this licentiate thesis has to be flexible and adaptable in order to cope with these standardisations.

Facing this reality, CC Systems and Volvo Construction Equipment initiated research cooperation with Mälardalen Real-Time Research Centre<sup>11</sup> by launching the HEAVE<sup>12</sup>, "A Component Technology for Heavy Vehicles", project. This licentiate thesis is produced within the HEAVE project.

### 1.3 Thesis Outline

Section 2 aims at presenting the contribution of this thesis by introducing the project hypothesis, the research questions, an outline of the included papers, and a summary of the contribution from an academic point of view as well as from an industrial point of view. Section 3 provides a summary of the research and the methodologies used during different phases of the work. Section 4 concludes the thesis and suggests future work.

The final parts (Section 5 to Section 8) of this thesis summarises the work by presenting four papers performed within this research project. The papers are summarised below:

Paper A, *Industrial Requirements on Component Technologies for Embedded Systems*, presents a requirements case-study on component-based software engineering for heavy vehicles. The purpose of the study was to build a solid research platform for the continuous work within the HEAVE project. In paper B, *Evaluation of Component Technologies with Respect to Industrial Requirements*, we present a component technology evaluation, based on the requirements collected during the industrial case-study. The idea was to discover

---

<sup>11</sup>Mälardalen Real-Time Research Centre, Home Page: <http://www.mrtc.mdh.se>

<sup>12</sup>HEAVE project, Home page: <http://www.mrtc.mdh.se/-projects/heave/>



which of the requirements that are fulfilled by existing technologies, and which are not. The study also includes a short survey description of each of the evaluated component technologies, and a table summarising the evaluation. In Paper C, *Towards a Dependable Component Technology for Embedded System Applications*, a prototype component technology, developed with safety-critical automotive applications in mind, is presented. The technology is illustrated as a case-study performed at CC Systems. Paper D, *Monitored Software Components - A Novel Software Engineering Approach -*, describes monitoring of software components, and the use of monitored software components as a general approach for engineering of embedded computer systems.



## Chapter 2

# Contribution

This chapter presents the contribution of this thesis by introducing the research questions and a summary of the contribution, together with an outline of the included papers.

### 2.1 Research Questions

The predefined goal of our research project is to identify, define and evaluate a suitable component technology for the business segment for heavy vehicles. Our assumption is that there is no *single component-technology* suitable for all segments of the embedded systems market, neither can an existing component technology for the Internet/office applications be adapted in order to satisfy the embedded system developer requirements. Instead, our idea was that different segments of the embedded systems market is best served by different technologies, and that the best way to find out if the assumption is valid – is to start unbiased and ask the involved companies about their specific needs, before looking too deep into different technical solutions.

These answers were then to be used as the research platform for the continued work in which we evaluate existing component technologies and implement proposed changes in a new, or modified, component technology.

The issues considered in our research project can be summarised by the following research questions:

*Why are existing software component technologies for embedded system development not used more frequently in industry?*

*(MainQuestion)*

This question can be considered the main topic of this work. Trying to find the answer to this question, we must examine the industrial development process of today, the industrial requirements on component-based software development, and the existing component technologies that could be suitable for embedded systems. However, this question is very broad and, strictly speaking, not suitable as a research question. Hence, the main question serve as a guideline but is split up into four sub-questions that is more appropriate for research, trying to identify the different aspects of the main question.

*Which are the most important requirements on a component technology for heavy vehicle developers in order to cope with the increasing demands on functionality and product costs?*

*(Q1)*

This question aims at finding the most important industrial requirements on a component technology for the specific business segment of heavy vehicles. The idea, stating this as the first research question, is that by finding the industrial prerequisites to introduce a component technology before looking too deep into technical solutions we can present an unbiased overview of the actual industrial requirements.

*What is (is not) offered in the existing component technologies, and how does this match the industrial requirements?*

*(Q2)*

Based on the requirements, i.e. (Q1), this question aims at finding parts of the component technologies that are lacking, or parts of existing component technologies that are particularly well addressed, and – in those cases – if possible identify satisfying technical solutions. To find the answer to this question, we must study a set of component technologies, and evaluate those technologies based on the collected industrial requirements.

*Is it possible, and sensible, to improve, extend, or simplify, existing component technologies (or parts of existing technologies) in order to fulfil the industrial requirements?*

*(Q3)*

This question is based on the answer to (Q1), and can be seen as an extension of (Q2). We aim at realising a deeper study, and a further investigation, of specific parts of a smaller set of the existing technologies (i.e. the question does not address the issues of developing a new component technology). The answer to this question will, possibly, include areas that need to be improved in order for the embedded systems developers to introduce a component technology. This research question might also point out areas for future work and/or include additional suggestions not thought of within the other research questions.

*Is it possible to combine the industrial requirements and the technical solutions in the state-of-the-art (and state-of-practice) component technologies, in order to find a custom-made component technology for heavy vehicles?*

(Q4)

This question intend to, based on existing techniques and the specified industrial requirements, find a technical, as well as development process related, suggestion/solution to whether or not it is possible to define and implement a component technology suitable for the market segment of heavy vehicles. This question is an extension of (Q3) and does possibly addresses the development of a new component technology.

## 2.2 Contribution

The contributions of this thesis are divided into two parts, the scientific contributions and the contributions for the participating companies.

### Scientific Contributions

The scientific contributions of this thesis project are mainly:

- The study of actual requirements from a specific industrial segment, and the survey of to what extent those requirements are fulfilled by existing component technologies.
- The implementation of a test-bed component technology, and a pilot-project, have a scientific value, illustrating how a technology based on industrial requirements can be used to solve problems that are not solved by commodity technologies.

- The proposed technique for monitoring software components, and reuse of monitored components as a general approach towards engineering of resource constrained embedded real-time control-systems. This approach illustrates a pragmatic engineering solution to often discussed scientific problems, e.g., how to ascertain worst-case execution-times.

### Industrial Contributions

For the participating companies, the main contributions are:

- The compilation of requirements and the assessment of the suitability of existing technologies.
- The test-bed implementation of an appropriate component technology (based on the industrial requirements and solutions from publicly available documentation about existing component technologies).
- The industrial pilot project implementing an embedded control application using the suggested component technology, will also provide valuable insight into how a component technology can be used at the participating companies.

## 2.3 Included Papers

This section summarises, and presents my contribution, of the included papers in this thesis.

### Paper A

*Industrial Requirements on Component Technologies for Embedded Systems*; Anders Möller, Joakim Fröberg and Mikael Nolin; In Proceedings of the 7<sup>th</sup> International Symposium on Component-Based Software Engineering, pages 146–161, Springer Verlag, Edinburgh, Scotland, May 2004.

**Summary:** This paper presents a study of requirements on component-based software engineering for heavy vehicles. The study was performed at Volvo Construction Equipment<sup>1</sup> and at CC Systems<sup>2</sup>, and the purpose of the study

---

<sup>1</sup>Volvo Construction Equipment, Eskilstuna, Sweden, <http://www.volvoce.com/>

<sup>2</sup>CC Systems, Uppsala, Sweden, <http://www.cc-systems.com/>

was to build a solid platform for the continuous research on component-based software for heavy vehicles.

**My contribution:** The study was initiated and accomplished by Anders. The work writing this paper was divided between the authors, but Anders was the driving author and responsible for putting the requirements together.

### Paper B

*Evaluation of Component Technologies with Respect to Industrial Requirements*; Anders Möller, Mikael Åkerholm, Johan Fredriksson and Mikael Nolin; In Proceedings of the 30<sup>th</sup> Euromicro Conference, Component-Based Software Engineering Track, pages 56–63; Rennes, France, September 2004.

**Summary:** This evaluation of component technologies is based on the requirements collected in Paper A. The idea was to study which of the requirements that are fulfilled by existing technologies, and which are not. The study also includes a short survey description of each of the evaluated component technologies, and a table summarising the evaluation.

**My contribution:** The evaluation was initiated by Anders. Anders was also responsible for summarising the industrial requirements, but the writing and evaluation part of the paper was equally divided between the authors.

### Paper C

*Towards a Dependable Component Technology for Embedded System Applications*; Mikael Åkerholm, Anders Möller, Hans Hansson and Mikael Nolin; To Appear in the Proceedings of the Workshop on Object-Oriented Real-time Dependable Systems, Sedona, Arizona, USA, February 2005.

**Summary:** In this paper, a prototype component technology, developed with safety-critical automotive applications in mind, is presented. The technology is based on a restrictive modelling language, and the technology implementation is illustrated as a technical case-study performed at CC Systems.

**My contribution:** The case-study was initiated by Anders. The major part of the component technology's compile-time activities was implemented by Mikael Å, and the run-time framework was implemented by Anders. Writing was equally distributed between the authors.

### **Paper D**

*Monitored Software Components – A Novel Software Engineering Approach*; Daniel Sundmark, Anders Möller and Mikael Nolin; In Proceedings of the 11<sup>th</sup> Asian-Pacific Conference on Software Engineering, Workshop on Software Architectures and Component Technologies, pages 624–631; Busan, Korea; November 2004.

**Summary:** The paper describes monitoring of software components, and use of monitored software components as a general approach for engineering of embedded computer systems.

**My contribution:** The idea writing this paper, and the idea of using monitoring as a pragmatic approach towards predictable assemblies came from Anders. Daniel was responsible for describing the monitoring techniques, and Anders for describing the existing techniques and the embedded-system settings. Writing the paper was joint work between the authors.



## Chapter 3

# Research Work and Method

Instead of starting from an existing component technology (like, e.g., CORBA [14] or Enterprise Java Beans [8]) and try to embed it into a heavy vehicle system, this project took a different approach in that we started unbiased by identifying specific industrial requirements from the heavy vehicle market segment.

Based on these requirements, we studied to what extent existing component technologies fulfilled those industrial desires. We did also assess to what extent existing technologies could be adapted in order to fulfil the requirements, or whether selected parts (like, e.g., tools, middlewares, and file-formats) could be reused if a new component technology were to be developed.

Equipped with this knowledge, we initiated the work of specifying a suitable component technology for the specific business segment of heavy vehicles. This specification covered issues like, e.g., component modelling, component-framework functionality, analysability, and component interoperability. Based on these specifications, and on similar work [10, 12, 25], we prepared a test-bed implementation of the component technology.

The work can be divided into five different parts, in which different research methods have been used. All phases have been performed in close cooperation with industry, but also with a lot of influences from, and cooperation with, other research groups, like, e.g., the SAVE<sup>1</sup> project. In the following, we discuss the specific research methods used in the different phases.

---

<sup>1</sup>SAVE project, Home Page: <http://www.mrtc.mdh.se/SAVE>

### 3.1 Preliminary Literature Study

The research presented within this thesis started with a preliminary literature study, summarised in the state-of-the-art report [26]. The report is based on about 30 articles summarising the area of component-based software engineering for safety critical embedded applications, and is divided into six different parts. The first part is a general part describing CBSE and embedded systems. The second part describes different component technology independent techniques that are considered useful for CBSE for embedded systems. The third part presents a set of existing component models and technologies. Section four describes general low-level technical issues of CBSE for embedded systems. Part five presents work done on architecture description languages, and the last section, section six, presents aspect oriented design/programming.

The literature study aimed at establishing basic knowledge about the existing component technologies for embedded systems. Understanding the state-of-the-art and state-of-practice component technologies was a prerequisite for the subsequent work. All the papers reviewed in the report have been read, presented, and discussed by all the authors during several workshop meetings.

### 3.2 Industrial Requirements Case-Study

This part of the research was aiming at finding the most important industrial requirements on a component technology for the business segment of heavy vehicles. The idea was to find the industrial prerequisites to introduce a component technology, before looking too deep into technical solutions. However, there are many different aspects and methods to consider when looking into questions regarding how to capture the most important requirements on a component technology suited for heavy vehicles.

Based on the preliminary literature study - a qualitative case-study interview protocol (i.e. a case-study questionnaire) [27] was put together focusing on finding the answer to the research question ( $Q1$ ), as stated in Sect. 2.1. Qualitative research methods aims to give clear understanding of the phenomenon studied without generalising, and can be performed by collecting information from a relatively small set of research objects. The qualitative methods are often relatively unsystematic and unstructured [28]. However, the case-study protocol is very important - and is used to keep the investigator targeted on the subject. This is done by including an overview of the case-study project, together with a description of the field procedures (i.e. having access to the in-

interviewees, having enough resources, etc) and a guide for the case-study report, in the protocol in addition to the actual case-study questions [27].

The case-study was performed at Volvo Construction Equipment and at CC Systems, and the respondents were senior technical staff from different parts of the organisation (like, e.g., project managers, development process specialists, programmers, and testing specialists). The case-study protocol questions were open – meaning that attendant questions were dependent on the respondent’s answer [27].

We also made an investigation to validate the reliability of our case-study results. This was realised by conducting interviews with industrial representatives, and by participating in discussions with engineers and researchers with heavy vehicle domain knowledge. The investigation confirmed our case-study work and further strengthened our conclusion that not only technical issues are of importance – also the development process related issue is deemed decisive for introducing a component technology in an industrial context.

### **3.3 Evaluation of Existing Technologies**

The next phase in our research was to look deeper into a smaller set of component technologies, and evaluate those technologies based on the collected industrial requirements. The technologies were selected based on the initial literature study, and were examined in great detail. Many of the published papers available from each component technology project were carefully studied.

The technologies described and evaluated are PECT [13], Koala [11], Rubus Component Model [10], PBO [29], PECOS [30] and CORBA-CCM [14]. We have chosen CORBA-CCM to represent the set of technologies existing in the Internet/office domain (other examples are, .NET [9] and Enterprise Java Beans [8]) since it is a technology that partly addresses embedded and real-time issues. Also, the Windows CE version of .NET [9] is omitted, since it is mainly targeted towards embedded display-devices, which only constitute a small subset of the devices in vehicular systems.

These technologies originate both from academia and industry, and the selection criterion has firstly been that there is enough information available (the evaluation is based on existing, publicly available, documentation), secondly that the authors claim that the technology is suitable for embedded systems, and finally we have tried to achieve a combination of both academic and industrial technologies.

The evaluation work was performed in small workshops, where the au-

thors discussed and evaluated the available written material from each of the chosen component technologies, and finally compared it with the industrial requirements. The appropriateness of the technologies solution to each of the requirements were summarised in a graded table (Sect. 6.4).

### **3.4 Implementing and Evaluating a Component Technology**

By combining the knowledge and experience collected from the previous parts of the research, together with results from the SAVE project, we implemented a prototype component technology. The component technology was based on the SaveComp Component Model [25], suggested within the SAVE project, developed with safety-critical dependable vehicle applications in mind.

The component technology is intended to provide three main benefits for developers of embedded systems: efficient development, predictable behaviour, and run-time efficiency. The technology implementation includes design-time, compile-time and run-time mechanisms and was implemented in cooperation with CC Systems.

To evaluate the suitability of the component technology, we implemented a test-bed application using the company's tools and techniques. The evaluation can be divided into three categories, the structural properties, the behavioural properties, and the process related properties. The evaluation was accomplished using a check-list assembled from requirements for automotive component technologies collected within this work, risks with using CBSE for embedded systems by Larn and Vickers [31], and from identified needs by Crnkovic [4].

### **3.5 Monitoring Software Components**

The requirements study and the component technology evaluation, as well as the evaluation of the component technology implementation, showed that one of the most central issues when introducing component-based development is the ability to analyse and predict the behaviour of a component assembly pre-run-time [13].

We studied related work (e.g., [32, 33, 34]) and some work done within the respective component technologies (e.g., [10, 35]). We found that not much

focus was put on monitoring as a solution to reach predictable component assemblies, and hence we presented a general engineering proposal to facilitate certifiable components, system-level testing and debugging, run-time contract checking and enhanced observability.

Hence, in the final part of this thesis – we propose a pragmatic method to monitor software components, and use of monitored software components, as a general approach for engineering of embedded computer systems. Continuous monitoring is to be used as the base for contract checking, and provides means for post-mortem crash analysis [36]; important prerequisites for many companies to start use 3rd party components in their dependable systems.

Monitoring software, as suggested, comprise (full or partial) solutions to many of the collected requirements, like analysability (Sect. 5.4.1) with respect to the enhanced ability to collect the information needed to perform schedulability and memory-consumption analysis. Monitoring can also be used to support replay debugging [36], where erroneous system-executions are recreated in a lab environment to allow tracing of bugs. Enhanced reusability (Sect. 5.4.2) and maintainable (Sect. 5.4.2) are one of the main benefits using monitored software components, since the components are continuously observed and at the end certified. However, there are contradicting aspects of monitoring. The limited resources (Sect. 5.4.1) are put at risk since resources (e.g. memory and CPU) are needed to drive the monitor.



## Chapter 4

# Conclusion and Future Work

To be able to address the main question (i.e. *Why are existing software component technologies for embedded system development not used more frequently in industry?*) of this thesis, we divided the work into smaller parts and tried to answer the different sub-questions.

One of the main contributions with this thesis is that it straightens out some of the question-marks regarding the actual industrial requirements placed on a component technology within the business segment of heavy vehicles. When trying to find an answer to (Q1) (Sect. 2.1), comprising the industrial requirements on a component technology, we have noticed that – for a component technology to be fully accepted by industry, the whole system development context needs to be considered. It is not only the technical properties that need to be addressed, but also development process related requirements.

The requirements collected are used to evaluate a set of component technologies, so that the risks with component-based development can be minimised before being introduced in an industrial context. Thus, we hope that this thesis can help companies take the step into tomorrow's technology. This evaluation helped us answering research question (Q2) in Sect. 2.1, and the conclusion is that non of the evaluated component technologies fulfil all the requirements and that no single component technology stands out as being a obvious best match for the requirements. However, it is interesting to see that most requirements are fulfilled by one or more techniques, implying that there exists solutions to each of the requirements. During the evaluation work we identified different areas where component technology improvements could be done. We also gathered valuable experience from this evaluation that was later

used when implementing a new component technology and when outlining future work.

To answer research question (*Q3*) and (*Q4*), comprising the possible areas of improvements within CBSE for embedded systems, we have described the initial implementation of our component technology suitable for vehicular systems. This work also includes an evaluation of the results in an industrial environment, using requirements identified in related research.

One area within component-based software engineering that we observed to be slightly weaker than most other technical areas is the ability to predict the component assembly behaviour pre-run-time. In this thesis we propose monitoring of software components, and reuse of monitored components, as a pragmatic engineering approach to facilitate predictability. The concept is general and addresses not only the development phase, but rather the whole product life-cycle. This work can be seen as a preliminary answer to research question (*Q3*) but also as an example of a possible area for future work.

Our plans for future work include different extensions of the component technology. We will be looking deeper into issues covering support for, e.g., multiple nodes, integration of legacy code with the components [37], enhanced run-time monitoring support [38], and a real-time database for structured handling of shared data [39].

An indication of the potential of our work within the HEAVE project is that the companies involved, i.e. CC Systems and Volvo Construction Equipment, find our ideas promising and has expressed a keen interest to continue the cooperation.



# Bibliography

- [1] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.
- [2] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, ISBN: 0201745720, 1998.
- [3] G. T. Heineman and W. T. Councill. *Component-based Software Engineering, Putting the Pieces Together*. Prentice-Hall, 2001. ISBN: 0-201-70485-4.
- [4] I. Crnkovic. Component-Based Approach for Embedded Systems. In *Proceedings of 9<sup>th</sup> International Workshop on Component-Oriented Programming*, June 2004. Oslo, Norway.
- [5] A. Brown and K. Wallnau. The Current State of CBSE. *IEEE Software*, September/October 1998.
- [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. Seattle, USA.
- [7] A. Ran. Software isn't built from LEGO blocks – Towards Architecture Based Reuse. Keynote speech by Alexander Ran (Nokia Research Center) at the Symposium on Software Reusability, Collocated with the International Conference on Software Engineering, May 1999. Los Angeles, USA.
- [8] Sun Microsystems. Enterprise Java Beans Technology. <http://java.sun.com/products/ejb/>.

- [9] Microsoft Component Technologies. COM/DCOM/.NET. <http://www.microsoft.com>.
- [10] K.L. Lundbäck, J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. In *Real-Time in Sweden – Presentation of Component-Based Software Development Based on the Rubus concept, Arcticus Systems*: <http://www.arcticus.se>. Västerås, Sweden.
- [11] R. van Ommering et al. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.
- [12] 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 6<sup>th</sup> International Workshop on Component-Based Software Engineering*, May 2003. Portland, Oregon, USA.
- [13] K. C. Wallnau. Volume III: A Component Technology for Predictable Assembly from Certifiable Components. Technical report, Software Engineering Institute, Carnegie Mellon University, April 2003. Pittsburg, USA.
- [14] CORBA Component Model 3.0. Object Management Group, June 2002. <http://www.omg.org/technology/documents/formal/components.htm>.
- [15] 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 Research Centre, Mälardalen University, March 2004. Västerås, Sweden.
- [16] International Standards Organisation (ISO). Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication, November 1993. vol. ISO Standard 11898.
- [17] J. Turely. The Two Percent Solution. *Embedded Systems Programming*, <http://www.embedded.com>, December 2002.
- [18] N. Andersson. Halva bilens värde är elektronik. *Automobil, NyTeknik*, September 2002. Swedish Technical Magazine.
- [19] MOST. Specification framework rev 1.1. MOST Coopertion, <http://www.mostnet.de>, November 1999.

- [20] LIN. – Protocol, Development Tools, and Software for Local Interconnect Networks. In 9th International Conference on Electronic Systems for Vehicles, October 2000. Baden-Baden, Germany.
- [21] I. Crnkovic, U. Askerlund, and A. Persson-Dahlqvist. *Implementing and Integrating Product Data Management and Software Configuration Management*. Artech House Software Engineering Library, 2002. ISBN: 1-58053-498-8.
- [22] Object Management Group. MinimumCORBA 1.0, August 2002. [http://www.omg.org/technology/documents/formal/minimum\\_CORBA.htm](http://www.omg.org/technology/documents/formal/minimum_CORBA.htm).
- [23] EAST-EEA. ITEA-Project-Number 0009. <http://www.east-eea.net/>.
- [24] AUTOSAR. The AUTOSAR consortium – Automotive Open System Architecture. <http://www.autosar.org/>.
- [25] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - a Component Model for Safety-Critical Real-Time Systems. In *Proceedings of 30<sup>th</sup> Euromicro Conference, Special Session Component Models for Dependable Systems*, September 2004.
- [26] M. Nolin, J. Fredriksson, J. Hammarberg, J. Huselius, J. Håkansson, A. Karlsson, O. Larses, M. Lindgren, G. Mustapic, A. Möller, T. Nolte, J. Norberg, D. Nyström, A. Tesanovic, and M. Åkerholm. Component-Based Software for Embedded Systems - A Literature Survey. Technical report, MRTC Report No 104, ISSN 1404-3041, ISRN MDH-MRTC-104/203-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June 2003. Västerås, Sweden.
- [27] R.K. Yin. *Case Study Research – Design and Methods*. Applied Social Research Methods Series, Volume 5, SAGE Publications, 2003. ISBN 0-7619-2553-8.
- [28] I.M. Holme and B.K. Solvang. *Forskningsmetodik - Om kvalitativa och kvantitativa metoder*. Studentlitteratur, Lund, ISBN 9144002114, 1997. Andra Upplagan.
- [29] 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 759 – 776, December 1997.

- [30] M. Winter, T. Genssler, et al. Components for Embedded Software – The PECOS Approach. In *The 2<sup>nd</sup> International Workshop on Composition Languages, in conjunction with the 16<sup>th</sup> ECOOP*, June 2002. Malaga, Spain.
- [31] W. Lam and A.J. Vickers. Managing the Risks of Component-Based Software Engineering. In *Proceedings of the 5<sup>th</sup> International Symposium on Assessment of Software Tools*, June 1997. Pittsburgh, USA.
- [32] J. Gao, E. Zhu, and S. Shim. Tracking component-based software. In *Proceedings of the International Conference on Software Engineering, 2000's COTS Workshop: Continuing Collaborations for Successful COTS Development*, 2000.
- [33] A. Jhumka, M. Hiller, and N. Suri. An Approach to Specify and Test Component-Based Dependable Software. In *Proceedings of the 7<sup>th</sup> IEEE International Symposium on High Assurance Systems Engineering*, pages 211 – 218, 2002.
- [34] J. Hörnstein and H. Edler. Test Reuse in CBSE Using Built-in Tests. In *Proceedings of Workshop on Component-based Software Engineering*, April 2002.
- [35] T. Genssler, A. Christoph, B. Schuls, M. Winter, et al. PECOS in a Nutshell. PECOS project <http://www.pecos-project.org>.
- [36] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288 – 295). ACM, April 2003.
- [37] 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.
- [38] D. Sundmark, A. Möller, and M. Nolin. Monitored Software Components – A Novel Software Engineering Approach –. In *Proceedings of the 11<sup>th</sup> Asia-Pacific Software Engineering Conference, Workshop on Software Architectures and Component Technologies*, November 2004. Pusan, Korea.

- [39] 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 Research Centre, Mälardalen University, May 2003. Mälardalen University Press.



## **Chapter 5**

# **Paper A: Industrial Requirements on Component Technologies for Embedded Systems**

Anders Möller, Joakim Fröberg and Mikael Nolin  
In Proceedings of the 7<sup>th</sup> International Symposium on Component-Based Software Engineering, pages 146–161, Edinburgh, Scotland, May 2004

### **Abstract**

Software component technologies have not yet been generally accepted by embedded-systems industries. In order to better understand why this is the case, we present a set of requirements, based on industrial needs, that are deemed decisive for introducing a component technology. The requirements we present can be used to evaluate existing component technologies before introducing them in an industrial context. They can also be used to guide modifications and/or extensions to component technologies, to make them better suited for industrial deployment. One of our findings is that a major source of requirements is non-technical in its nature. For a component technology to become a viable solution in an industrial context, its impact on the overall development process needs to be addressed. This includes issues like component life-cycle management, and support for the ability to gradually migrate into the new technology.



## 5.1 Introduction

During the last decade, Component-Based Software Engineering (CBSE) for embedded systems has received a large amount of attention, especially in the software engineering research community. In the office/Internet area CBSE has had tremendous impact, and today components are downloaded and on the fly integrated into, e.g., word processors and web browsers. In industry however, CBSE is still, to a large extent, envisioned as a promising future technology to meet industry specific demands on improved quality and lowered cost, by facilitating software reuse, efficient software development, and more reliable software systems [1].

CBSE has not yet been generally accepted by embedded-system developers. They are in fact, to a large extent, still using monolithic and platform dependent software development techniques, in spite of the fact that this make software systems difficult to maintain, upgrade, and modify. One of the reasons for this status quo is that there are significant risks and costs associated with the adoption of a new development technique. These risks must be carefully evaluated and managed before adopting a new development process.

The main contribution of this paper is that it straightens out some of the question-marks regarding actual industrial requirements placed on a component technology. We describe the requirements on a component technology as elicited from two companies in the business segment of heavy vehicles. Many of the requirements are general for the automotive industry, or even larger parts of the embedded systems market (specifically segments that deal with issues about distributed real-time control in safety-critical environments), but there are also some issues that are specific for the business segment of heavy vehicles.

The list of requirements can be used to evaluate existing component technologies before introducing them in an industrial context, therefore minimising the risk when introducing a new development process. Thus, this study can help companies to take the step into tomorrow's technology today. The list can also be used to guide modifications and/or extensions to component technologies, to make them better suited for industrial deployment within embedded system companies. Our list of requirements also illustrates how industrial requirements on products and product development impact requirements on a component technology.

This paper extends previous work, studying the requirements for component technologies, in that the results are not only based on our experience, or experience from a single company [2, 3]. We base most of our results on inter-

views with senior technical staff at the two companies involved in this paper, but we have also conducted interviews with technical staff at other companies. Furthermore, since the embedded systems market is so diversified, we have limited our study to applications for distributed embedded real-time control in safety-critical environments, specifically studying companies within the heavy vehicles market segment. This gives our results higher validity, for this class of applications, than do more general studies of requirements in the embedded systems market [4].

## 5.2 Introducing CBSE in the Vehicular Industry

Component-Based Software Engineering arouses interest and curiosity in industry. This is mainly due to the enhanced development process and the improved ability to reuse software, offered by CBSE. Also, the increased possibility to predict the time needed to complete a software development project, due to the fact that the assignments can be divided into smaller and more easily defined tasks, is seen as a driver for CBSE.

CBSE can be approached from two, conceptually different, points of view; distinguished by whether the components are (1) used as a design philosophy independent from any concern for reusing existing components, or (2) seen as reusable off-the-shelf building blocks used to design and implement a component-based system [5]. When talking to industrial software developers with experience from using a CBSE development process [6], such as Volvo Construction Equipment<sup>1</sup>, the first part, (1), is often seen as the most important advantage. Their experience is that the design philosophy of CBSE gives rise to good software architecture and significantly enhanced ability to divide the software in small, clearly-defined, development subprojects. This, in turn, gives predictable development times and shortens the time-to-market. The second part, (2), are by these companies often seen as less important, and the main reason for this is that experience shows that most approaches to large scale software reuse is associated with major risks and high initial costs. Rather few companies are willing to take these initial costs and risks since it is difficult to guarantee that money is saved in the end.

On the other hand, when talking to companies with less, or no, experience from component-based technologies, (2) is seen as the most important motivation to consider CBSE. This discrepancy between companies with and without CBSE experience is striking.

---

<sup>1</sup>Volvo Construction Equipment, Home Page: <http://www.volvo.com>

However, changing the software development process to using CBSE does not only have advantages. Especially in the short term perspective, introducing CBSE represents significant costs and risks. For instance, designing software to allow reuse requires (sometimes significantly) higher effort than does designing for a single application [7]. For resource constrained systems, design for reuse is even more challenging, since what are the most critical resources may vary from system to system (e.g. memory or CPU-load). Furthermore, a component designed for reuse may exhibit an overly rich interface and an associated overly complex and resource consuming implementation. Hence, designing for reuse in resource constrained environments requires significant knowledge not only about functional requirements, but also about non-functional requirements. These problems may limit the possibilities of reuse, even when using CBSE.

With any software engineering task, having a clear and complete understanding of the software requirements is paramount. However, practice shows that a major source of software errors comes from erroneous, or incomplete, specifications [7]. Often incomplete specifications are compensated for by engineers having good domain knowledge, hence having knowledge of implicit requirements. However, when using a CBSE approach, one driving idea is that each component should be fully specified and understandable by its interface. Hence, the use of implicit domain knowledge not documented in the interface may hinder reuse of components. Also, division of labour into smaller projects focusing on single components, require good specifications of what interfaces to implement and any constraints on how that implementation is done, further disabling use of implicit domain knowledge. Hence, to fully utilise the benefits of CBSE, a software engineering process that do not rely on engineers' implicit domain knowledge need to be established.

Also, when introducing reuse of components across multiple products and/or product families, issues about component management arise. In essence, each component has its own product life-cycle that needs to be managed. This includes version and variant management, keeping track of which versions and variants is used in what products, and how component modifications should be propagated to different version and variants. Components need to be maintained, as other products, during their life cycle. This maintenance needs to be done in a controlled fashion, in order not to interfere adversely with ongoing projects using the components. This can only be achieved using adequate tools and processes for version and variant management.

## 5.3 A Component Technology for Heavy Vehicles

Existing component technologies are in general not applicable to embedded computer systems, since they do not consider aspects such as safety, timing, and memory consumption that are crucial for many embedded systems [8, 9]. Some attempts have been made to adapt component technologies to embedded systems, like, e.g., MinimumCORBA [10]. However, these adaptations have not been generally accepted in the embedded system segments. The reason for this is mainly due to the diversified nature of the embedded systems domain. Different market segments have different requirements on a component technology, and often, these requirements are not fulfilled simply by stripping down existing component technologies; e.g. MinimumCORBA requires less memory than does CORBA, however, the need to statically predict memory usage is not addressed.

It is important to keep in mind that the embedded systems market is extremely diversified in terms of requirements placed on the software. For instance, it is obvious that software requirements for consumer products, telecom switches, and avionics are quite different. Hence, we will focus on one single market segment: the segment of heavy vehicles, including, e.g., wheel loaders and forest harvesters. It is important to realise that the development and evaluation of a component technology is substantially simplified by focusing on a specific market segment. Within this market segment, the conditions for software development should be similar enough to allow a lightweight and efficient component technology to be established [11].

### 5.3.1 The Business Segment of Heavy Vehicles

Developers of heavy vehicles faces a situation of (1) high demands on reliability, (2) requirements on low product cost, and (3) supporting many configurations, variants and suppliers. Computers offer the performance needed for the functions requested in a modern vehicle, but at the same time vehicle reliability must not suffer. Computers and software add new sources of failures and, unfortunately, computer engineering is less mature than many other fields in vehicle development and can cause lessened product reliability. This yields a strong focus on the ability to model, predict, and verify computer functionality.

At the same time, the product cost for volume products must be kept low. Thus, there is a need to include a minimum of hardware resources in a product (only as much resources as the software really needs). The stringent cost requirements also drive vehicle developers to integrate low cost components from

suppliers rather than develop in-house. On top of these demands on reliability and low cost, vehicle manufacturers make frequent use of product variants to satisfy larger groups of customers and thereby increase market share and product volume.

In order to accommodate (1)-(3), as well as an increasing number of features and functions, the electronic system of a modern vehicle is a complex construction which comprise electronic and software components from many vendors and that exists in numerous configurations and variants.

The situation described cause challenges with respect to verification and maintenance of these variants, and integration of components into a system. Using software components, and a CBSE approach, is seen as a promising way to address challenges in product development, including integration, flexible configuration, as well as good reliability predictions, scalability, software reuse, and fast development. Further, the concept of components is widely used in the vehicular industry today. Using components in software would be an extension of the industry's current procedures, where the products today are associated with the components that constitute the particular vehicle configuration.

What distinguishes the segment of heavy vehicles in the automotive industry is that the product volumes are typically lower than that of, e.g., trucks or passenger cars. Also the customers tend to be more demanding with respect to technical specifications such as engine torque, payload etc, and less demanding with respect to style. This causes a lower emphasis on product cost and optimisation of hardware than in the automotive industry in general. The lower volumes also make the manufacturers more willing to design variants to meet the requests of a small number of customers.

However, the segment of heavy vehicles is not homogeneous with respect to software and electronics development practices. For instance, the industrial partners in this paper face quite different market situations and hence employ different development techniques:

- CC Systems<sup>2</sup> (CCS) is developing and supplying advanced distributed embedded real-time control systems with focus on mobile applications. Examples, including both hardware and software, developed by CCS are forest harvesters, rock drilling equipment and combat vehicles. The systems developed by CCS are built to endure rough environments, and are characterised by safety criticality, high functionality, and the requirements on robustness and availability are high.

---

<sup>2</sup>CC Systems, Home page: <http://www.cc-systems.com>

CCS works as a distributed software development partner, and cooperates, among others, with Alvis Hägglunds<sup>3</sup>, Timberjack<sup>4</sup> and Atlas Copco<sup>5</sup>. Experience from these companies are included in this paper, this makes our findings more representative for the business segment of heavy vehicles.

CCS' role as subcontractor requires a high degree of flexibility with respect to supported target environments. Often, CCS' customers have requirements regarding what hardware or operating systems platforms to use, hence CCS cannot settle to support only some predefined set of environments. Nevertheless, to gain competitive advantages, CCS desires to reuse software between different platforms.

- Volvo Construction Equipment (VCE) is one of the world's major manufacturers of construction equipment, with a product range encompassing wheel loaders, excavators, motor graders, and more. What these products have in common is that they demand high reliability control systems that are maintainable and still cheap to produce. The systems are characterised as distributed embedded real-time systems, which must perform in an environment with limited hardware resources.

VCE develops the vehicle electronics and most software in house. Some larger software parts, such as the operating system, are bought from commercial suppliers. VCE's role as both system owner and system developer gives them full control over the system's architecture. This, in turn, has given them the possibility to select a small set of (similar) hardware platforms to support, and select a single operating systems to use. Despite this degree of control over the system, VCE's experience is that software reuse is still hindered; for instance by non-technical issues like version and variant management, and configuration management.

### 5.3.2 System Description

In order to describe the context for software components in the vehicular industry, we will first explore some central concepts in vehicle electronic systems. Here, we outline some common and typical solutions and principles used in the design of vehicle electronics. The purpose is to describe commonly used solu-

---

<sup>3</sup>Alvis Hägglunds, Home page: <http://www.alvishagglunds.se>

<sup>4</sup>Timberjack, Home page: <http://www.timberjack.com>

<sup>5</sup>Atlas Copco, Home page: <http://www.atlascopco.com>

tions, and outline the de facto context for application development and thereby also requirements for software component technologies.

The system architecture can be described as a set of computer nodes called Electronic Control Units (ECUs). These nodes are distributed throughout the vehicle to reduce cabling, and to provide local control over sensors and actuators. The nodes are interconnected by one or more communication bus forming the network architecture of the vehicle. When several different organisations are developing ECUs, the bus often acts as the interface between nodes, and hence also between the organisations. The communication bus is typically low cost and low bandwidth, such as the Controller Area Network (CAN) [12].

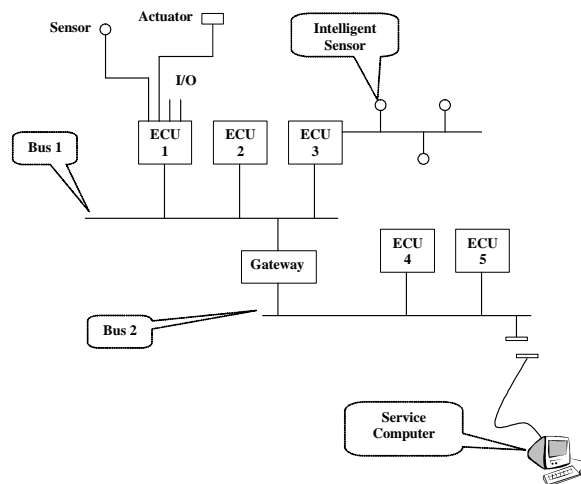


Figure 5.1: Example of a vehicle network architecture

In the example shown in Fig. 5.1, the two communication busses are separated using a gateway. This is an architectural pattern that can be used for several reasons, e.g., separation of criticality, increased total communication bandwidth, fault tolerance, compatibility with standard protocols [13, 14, 15], etc. Also, safety critical functions may require a high level of verification, which is usually very costly. Thus, non-safety related functions might be separated to reduce cost and effort of verification. In some systems the network is required to give synchronisation and provide a fault tolerance mechanisms.

The hardware resources are typically scarce due to the requirements on low

product cost. Addition of new hardware resources will always be defensive, even if customers are expected to embrace a certain new function. Because of the uncertainty of such expectations, manufacturers have difficulties in estimating the customer value of new functions and thus the general approach is to keep resources at a minimum.

In order to exemplify the settings in which software components are considered, we have studied our industrial partner's currently used nodes. Below we list the hardware resources of a typical ECU with requirements on sensing and actuating, and with a relatively high computational capacity (this example is from a typical power train ECU):

Processor:	25 MHz 16 bit processor (e.g. Siemens C167)
Flash:	1 MB used for applications
RAM:	128 kB used for the runtime memory usage
EEPROM:	64 kB used for system parameters
Serial interfaces:	RS232 or RS485, used for service purpose
Communications:	Controller Area Network (CAN) (one or more interfaces)
I/O:	There is a number of digital and analogue in and out ports

Also, included in a vehicle's electronic system can be display computer(s) with varying amounts of resources depending on product requirements. There may also be PC-based ECU's for non-control applications such as telematics, and information systems. Furthermore, in contrast to these resource intense ECU's, there typically exists a number of small and lightweight nodes, such as, intelligent sensors (i.e. processor equipped, bus enabled, sensors).

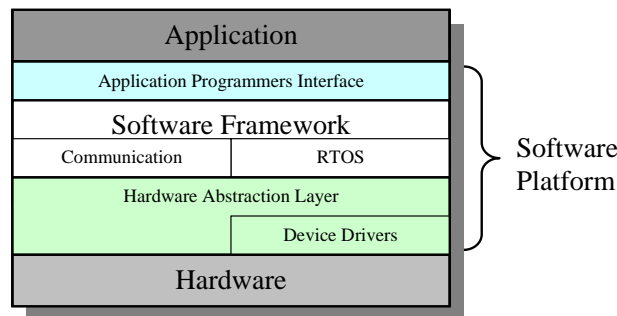


Figure 5.2: Internals of an ECU - A software platform



Figure 5.2 on the facing page depicts the typical software architecture of an ECU. Current practice typically builds on top of a reusable "software platform", which consists of a hardware abstraction layer with device drivers and other platform dependent code, a Real-Time Operating System (RTOS), one or more communication protocols, and possibly a software (component) framework that is typically company (or project) specific. This software platform is accessible to application programmers through an Application Programmers Interface (API). Different nodes, presenting the same API, can have different realisation of the different parts in the software platform (e.g. using different RTOSs).

Today it is common to treat parts of the software platform as components, e.g. the RTOS, device drivers, etc, in the same way as the ECU's bus connectors and other hardware modules. That is, some form of component management process exists; trying to keep track of which version, variant, and configuration of a component is used within a product. This component-based view of the software platform is however not to be confused with the concept of CBSE since the components does not conform to standard interfaces or component models.

## 5.4 Requirements on a Component Technology for Heavy Vehicles

There are many different aspects and methods to consider when looking into questions regarding how to capture the most important requirements on a component technology suited for heavy vehicles. Our approach has been to cooperate with our industrial partners very closely, both by performing interviews and by participating in projects. In doing so, we have extracted the most important requirements on a component-based technique from the developers of heavy vehicles point of view.

The requirements are divided in two main groups, the technical requirements (Sect. 5.4.1) and the development process related requirements (Sect. 5.4.2). Also, in Sect. 5.4.3 we present some implied (or derived) requirements, i.e. requirements that we have synthesised from the requirements in sections 5.4.1 and 5.4.2, but that are not explicit requirements from industry. In Sect. 5.4.4 we discuss, and draw conclusions from, the listed requirements.

### 5.4.1 Technical Requirements

The technical requirements describe the needs and desires that our industrial partners have regarding the technically related aspects and properties of a component technology.

#### **Analysable**

Vehicle industry strives for better analyses of computer system behaviour in general. This striving naturally affects requirements placed on a component model. System analysis, with respect to non-functional properties, such as the timing behaviour and the memory consumption, of a system built up from well-tested components is considered highly attractive. In fact, it is one of the single most distinguished requirements defined by our industrial partners.

When analysing a system, built from well-tested, functionally correct, components, the main issues is associated with composability. The composability problem must guarantee non-functional properties, such as the communication, synchronisation, memory, and timing characteristics of the system [1].

When considering timing analysability, it is important to be able to verify (1) that each component meet its timing requirements, (2) that each node (which is built up from several components) meet its deadlines (i.e. schedulability analysis), and (3) to be able to analyse the end-to-end timing behaviour of functions in a distributed system.

Because of the fact that the systems are resource constrained (Sect. 5.3), it is important to be able to analyse the memory consumption. To check the sufficiency of the application memory, as well as the ROM memory, is important. This check should be done pre-runtime to avoid failures during runtime.

In a longer perspective, it is also desirable to be able to analyse properties like reliability and safety. However, these properties are currently deemed too difficult to address on a component level and traditional methods (like testing and reviewing) are considered adequate.

#### **Testable and debuggable**

It is required that there exist tools that support debugging both at component level, e.g. a graphical debugging tool showing the components in- and out-port values, and at the traditional white-box source code debugging level. The test and debug environment needs to be "component aware" in the sense that port-values can be monitored and traced and that breakpoints can be set on component level.

Testing and debugging is by far the most commonly used technique to verify software systems functionality. Testing is a very important complement to analysis, and it should not be compromised when introducing a component technology.

In fact, the ability to test embedded-system software can be improved when using CBSE. This is possible because the component functionality can be tested in isolation. This is a desired functionality asked for by our industrial partners. This test should be used before the system tests, and this approach can help finding functional errors and source code bugs at the earliest possible opportunity.

### **Portable**

The components, and the infrastructure surrounding them, should be platform independent to the highest degree possible. Here, platform independent means hardware independent, RTOS independent and communication protocol independent.

Components are kept portable by minimising the number of dependencies to the software platform. Such dependencies are of 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.

Ideally, components should also be independent of the component framework used during run-time. This may seem far fetched, since traditionally a component model has been tightly integrated with its component framework. However, support for migrating components between component frameworks is important for companies cooperating with different customers, using different hardware and operating systems, such as CC Systems.

### **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.

Systems are resource constrained to lower the production cost and thereby increase profit. When companies design new ECUs, future profit is the main concern. Therefore the hardware is dimensioned for anticipated use but not more.

Provided that the customers are willing to pay the extra money, to be able to use more complex software functionality in the future, more advanced hardware may be appropriate. This is however seldom the case, usually the customers are very cost sensitive. The developer of the hardware rarely takes the extra cost to extend the hardware resources, since the margin of profit on electronics development usually is low.

One possibility, that can significantly reduce resource consumption of components and the component framework, is to limit the possible run-time dynamics. This means that it is desirable to allow only 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 at run-time. However, this dynamic behaviour comes at the price of increased resource consumption.

### **Component Modelling**

A component technology should be based on a standard modelling language like UML [16] or UML 2.0 [17]. The main reason for choosing UML is that it is a well known and thoroughly tested modelling technique with tools and formats supported by third-party developers.

The reason for our industrial partners to have specific demands in these details, is that it is believed that the business segment of heavy vehicles does not have the possibility do develop their own standards and practices. Instead they preferably relay on the use of simple and mature techniques supported by a welth of third party suppliers.

### **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 believed to enhance reusability, and to limit resource consumption. The computational model should be focused on a pipe-and-filter model [18]. 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.

However, experience from VCE shows that the pipe-and-filter model does not fit all parts of the system, and that force fitting applications to the pipe-and-filter model may lead to overly complex components. Hence, it is desirable to

have support for other computational models; unfortunately, however, which models to support is not obvious and is an open question.

### 5.4.2 Development Requirements

When discussing requirements for CBSE technologies, the research community often overlooks requirements related to the development process. For software developing companies, however, these requirements are at least as important as the technical requirements. When talking to industry, earning money is the main focus. This cannot be done without having an efficient development processes deployed. To obtain industrial reliance, the development requirements need to be considered and addressed by the component technology and tools associated with the technology.

#### Introducible

It should be possible for companies to gradually migrate into a new development technology. It is important to make the change in technology as safe and inexpensive as possible.

Revolutionary changes in the development technique used at a company are associated with high risks and costs. Therefore a new technology should be possible to divide into smaller parts, which can be introduced separately. For instance, if the architecture described in Fig. 5.2 is used, the components can be used for application development only and independently of the real-time operating system. Or, the infrastructure can be developed using components, while the application is still monolithic.

One way of introducing a component technology in industry, is to start focusing on the development process related requirements. When the developers have accepted the CBSE way of thinking, i.e. thinking in terms of reusable software units, it is time to look at available component technologies. This approach should minimise the risk of spending too much money in an initial phase, when switching to a component technology without having the CBSE way of thinking.

#### Reusable

Components should be reusable, e.g., for use in new applications or environments than those for which they were originally designed [19]. The requirement of reusability can be considered both a technical and a development

process related requirement. Development process related since it has to deal with aspects like version and variant management, initial risks and cost when building up a component repository, etc. Technical since it is related to aspects such as, how to design the components with respect to the RTOS and HW communication, etc.

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 simplified further by using input parameters to the components. Parameters that are fixed at compile-time, should allow automatic reduction of run-time overhead and complexity.

A clear, explicit, and well-defined component interface is crucial to enhance the software reusability. To be able to replace one component in the software system, a minimal amount of time should be spent trying to understand the component that should be interchanged.

It is, however, both complex and expensive to build reusable components for use in distributed embedded real-time systems [1]. The reason for this is that the components must work together to meet the temporal requirements, the components must be light-weighted since the systems are resource constrained, the functional errors and bugs must not lead to erroneous outputs that follow the signal flow and propagate to other components and in the end cause unsafe systems. Hence, reuse must be introduced gradually and with great care.

### **Maintainable**

The components should be easy to change and maintain, meaning that 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 where the component is used.

In essence, this requirement is a product of the previous requirement on reusability. The flip-side of reusability is that the ability to reuse and reconfigure the components using parameters leads to an abundance of different configurations used in different vehicles. The same type of vehicle may use different software settings and even different component or software versions. So, by introducing reuse we introduce more administrative work.

Reusing software components lead to a completely new level of software management. The components need to be stored in a repository where different

versions and variants need to be managed in a sufficient way. Experiences from trying to reuse software components show that reuse is very hard and initially related with high risks and large overheads [1]. These types of costs are usually not very attractive in industry.

The maintainability requirement also includes sufficient tools supporting the service of the delivered vehicles. 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. Also, focusing on an understandable model makes the development process faster and it is likely that there will be fewer bugs.

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. It is widely known that many software errors occur in code that deals with synchronisation, buffer management and communications. However, when using component technologies such code can, and should, be automatically generated; leaving application engineers to deal with application functionality.

### **5.4.3 Derived Requirements**

Here, we present two implied requirements, i.e. requirements that we have synthesised from the requirements in sections 5.4.1 and 5.4.2, but that are not explicit requirements from the vehicular industry.

#### **Source Code Components**

A component should be source code, i.e., no binaries. The reasons for this include that companies are used to have access to the source code, to find functional errors, and enable support for white box testing (Sect. 5.4.1). Since

source code debugging is demanded, even if a component technology is used, black box components is undesirable.

Using black-box components would, regarding to our industrial partners, lead to a feeling of not having control over the system behaviour. However, the possibility to look into the components does not necessary mean that you are allowed to modify them. In that sense, a glass-box component model is sufficient.

Source code components also leaves room for compile-time optimisations of components, e.g., stripping away functionality of a component that is not used in a particular application. Hence, source code components will contribute to lower resource consumption (Sect. 5.4.1).

### **Static Configuration**

For a component model to better support the technical requirements of analysability (Sect. 5.4.1), testability (Sect. 5.4.1), and light-weightiness (Sect. 5.4.1), the component model should be configured pre-runtime, i.e. at compile time. Component technologies for use in the office/Internet domain usually focus on a dynamic behaviour [8, 9]. This is of course appropriate in this specific domain, where powerful computers are used. Embedded systems, however, face another reality - with resource constrained ECU's running complex, dependable, control applications. Static configuration should also improve the development process related requirement of understandability (Sect. 5.4.2), since there will be no complex run-time reconfigurations.

Another reason for the static configuration is that a typical control node, e.g. a power train node, does not interact directly with the user at any time. The node is started when the ignition key is turned on, and is running as a self-contained control unit until the vehicle is turned off. Hence, there is no need to reconfigure the system during runtime.

### **5.4.4 Discussion**

Reusability is perhaps the most obvious reason to introduce a component technology for a company developing embedded real-time control systems. This matter has been the most thoroughly discussed subject during our interviews. However, it has also been the most separating one, since it is related to the question of deciding if money should be invested in building up a repository of reusable components.



Two important requirements that has emerged during the discussions with our industrial partners are safety and reliability. These two are, as we see it, not only associated with the component technology. Instead, the responsibility of designing safe and reliable system rests mainly on the system developer. The technology and the development process should, however, give good support for designing safe and reliable systems.

Another part that has emerged during our study is the need for a quality rating of the components depending on their success when used in target systems. This requirement can, e.g., be satisfied using Execution Time Profiles (ETP's), discussed in [20]. By using ETPs to represent the timing behaviour of software components, tools for stochastic schedulability analysis can be used to make cost-reliability trade offs by dimensioning the resources in a cost efficient way to achieve the reliability goals. There are also emerging requirements regarding the possibilities to grade the components depending on their software quality, using for example different SIL (Safety Integrity Levels) [21] levels.

## 5.5 Conclusions

Using software components and a CBSE approach is, by industry, seen as a promising way to address challenges in product development including integration, flexible configuration, as well as good reliability predictions, scalability, reliable reuse, and fast development. However, changing the software development process to using CBSE does not only have advantages. Especially in the short term perspective, introducing CBSE represents significant costs and risks.

The main contribution of this paper is that it straightens out some of the question-marks regarding actual industrial requirements placed on a component technology. We describe the requirements on a component technology as elicited from two companies in the business segment of heavy vehicles. The requirements are divided in two main groups, the technical requirements and the development process related requirements. The reason for this division is mainly to clarify that the industrial actors are not only interested in technical solutions, but also in improvements regarding their development process.

The list of requirements can be used to evaluate existing component technologies before introducing them in an industrial context, therefore minimising the risk when introducing a new development process. Thus, this study can help companies to take the step into tomorrow's technology today. They can also be used to guide modifications and/or extensions to component technolo-

gies, to make them better suited for industrial deployment within embedded system companies.

We will continue our work by evaluating existing software component technologies with respect to these requirements. Our initial findings from this evaluation can be found in [22]. Using that evaluation we will (1) study to what extent existing technologies can be adapted in order to fulfil the requirements of this paper, (2) investigate if selected parts of standard technologies like tools, middleware, and message-formats can be reused, (3) make a specification of a component technology suitable for heavy vehicles, and (4) build a test bed implementation based on the specification.

## **Acknowledgements**

A special thanks to Nils-Erik Bånkestad and Robert Larsson, at Volvo Construction Equipment, for fruitful discussions and for their helpfulness during our stay. We would also like to thank Jörgen Hansson at CC Systems for interesting discussions, new ideas, and for making this research project possible.

# Bibliography

- [1] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.
- [2] M. Winter, T. Genssler, et al. Components for Embedded Software – The PECOS Approach. In *The 2<sup>nd</sup> International Workshop on Composition Languages, in conjunction with the 16<sup>th</sup> ECOOP*, June 2002. Malaga, Spain.
- [3] R. van Ommering et al. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, March 2000.
- [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. Pittsburg, USA.
- [5] A. Brown and K. Wallnau. The Current State of CBSE. *IEEE Software*, September/October 1998.
- [6] C. Nordström, M. Gustafsson, et al. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In *Eighth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, April 2001. Washington, USA.
- [7] S. R. Schach. *Classical and Object-Oriented Software Engineering*. McGraw-Hill Science/Engineering/Math; 3rd edition, 1996. ISBN 0-256-18298-1.
- [8] Microsoft Component Technologies. COM/DCOM/.NET. <http://www.microsoft.com>.

- [9] Sun Microsystems. Enterprise Java Beans Technology. <http://java.sun.com/products/ejb/>.
- [10] Object Management Group. MinimumCORBA 1.0, August 2002. [http://www.omg.org/technology/documents/formal/minimum\\_CORBA.htm](http://www.omg.org/technology/documents/formal/minimum_CORBA.htm).
- [11] A. Möller, J. Fröberg, and M. Nolin. What are the needs for components in vehicular systems? – An Industrial Perspective –. In *Proceedings of the Euromicro Conference on Real-Time Systems – Work-in-Progress Session*. IEEE Computer Society, July 2003. Porto, Portugal.
- [12] International Standards Organisation (ISO). Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication, November 1993. vol. ISO Standard 11898.
- [13] CiA. CANopen Communication Profile for Industrial Systems, Based on CAL, October 1996. CiA Draft Standard 301, rev 3.0, <http://www.canopen.org>.
- [14] SAE Standard. SAE J1939 Standards Collection. <http://www.sae.org>.
- [15] SAE Standard. SAE J1587, Joint SAE/TMC Electronic Data Interchange Between Microcomputer Systems In Heavy-Duty Vehicle Applications. <http://www.sae.org>.
- [16] B. Selic and J. Rumbaugh. Using UML for modelling complex real-time systems, 1998. Rational Software Corporation.
- [17] Object Management Group. UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003. <http://www.omg.com/uml/>.
- [18] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall; 1 edition, 1996. ISBN 0-131-82957-2.
- [19] 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. Seattle, USA.
- [20] T. Nolte, A. Möller, and M. Nolin. Using Components to Facilitate Stochastic Schedulability. In *Proceedings of the 24<sup>th</sup> Real-Time System Symposium – Work-in-Progress Session*. IEEE Computer Society, December 2003. Cancun, Mexico.

- [21] SIL. Safety Integrity Levels – Does Reality Meet Theory?, 2002. Report f. seminar held at the IEE, London, on 9 April 2002.
- [22] A. Möller, M Åkerholm, J. Fredriksson, and M. Nolin. Software Component Technologies for Real-Time Systems – An Industrial Perspective. In *Proceedings of the 24<sup>th</sup> Real-Time System Symposium – Work-in-Progress Session*. IEEE Computer Society, December 2003. Cancun, Mexico.



## **Chapter 6**

# **Paper B: Evaluation of Component Technologies with Respect to Industrial Requirements**

Anders Möller, Mikael Åkerholm, Johan Fredriksson and Mikael Nolin  
In Proceedings of the 30<sup>th</sup> Euromicro Conference, Component-Based Software  
Engineering Track, pages: 56–63, 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.



## 6.1 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.

## 6.2 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 (Sect. 6.2.1) and the development process related requirements (Sect. 6.2.2). In addition, Sect. 6.2.3 contains derived requirements, i.e. requirements that we have synthesised from the requirements in sections 6.2.1 and 6.2.2 but that are not explicitly stated requirements from the vehicular industry [1].

### 6.2.1 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 predictable [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.

### **6.2.2 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 were 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 (Sect. 6.2.2) 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.

**6.2.3 Derived Requirements**

Here, we present requirements that we have synthesised from the requirements in sections 6.2.1 and 6.2.2, 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 (Sect. 6.2.1). 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!<sup>1</sup>

Using black-box components would lead to a fear of loosing control over the system behaviour (Sect. 6.2.2). 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 (Sect. 6.2.1), 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 (Sect. 6.2.1), testability (Sect. 6.2.1), and resource consumption (Sect. 6.2.1), 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 (Sect. 6.2.2), since each possible configuration is known before run-time.

## **6.3 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

---

<sup>1</sup>This can be view 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 existing, publically available, documentation.

### 6.3.1 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.<sup>2</sup> 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.

---

<sup>2</sup>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.

### 6.3.2 Koala

The Koala component technology [10] is designed and used by Philips<sup>3</sup> 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

---

<sup>3</sup>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.

### 6.3.3 Rubus Component Model

The Rubus Component Model (Rubus CM) [12] is developed by Arcticus systems.<sup>4</sup> 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 (Sect. 6.2.1), 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 (Sect. 6.2.1). 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.

---

<sup>4</sup>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).

### 6.3.4 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.<sup>5</sup> 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

---

<sup>5</sup>Carnegie Mellon University; Home Page <http://www.cmu.edu>

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

### 6.3.5 PECOS

PECOS<sup>6</sup> (PErvasive COmponent Systems) [16, 17] is a collaborative project between ABB Corporate Research Centre<sup>7</sup> 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 [18] to model concurrent activities like component compositions, scheduling of components, and synchronisation of shared ports [19]. Debugging can be performed using COTS debugging and monitoring tools. However, the component technology does not support debugging on component level as described in Sect. 6.2.1.

The PECOS component technology uses a layered software architecture, which enhance portability (Sect. 6.2.1). 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.

---

<sup>6</sup>PECOS Project, Home Page: <http://www.pecos-project.org/>

<sup>7</sup>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 (Sect. 6.2.2) 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 [20]. The devices are statically configured.

### 6.3.6 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<sup>8</sup> (OMG). There are different versions of CORBA available, e.g., MinimumCORBA [21] for resource constrained systems, and RT-CORBA [22] 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) [23], which extends the CORBA object model by defining features and services that enables application developers to implement, manage, 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

---

<sup>8</sup>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.

## 6.4 Summary of Evaluation

In this section we assign numerical grades to each of the component technologies described in Sect. 6.3, grading how well they fulfil each of the requirements of Sect. 6.2. The grades are based on the discussion summarised in Sect. 6.3. 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 Fig. 6.1). For the CORBA-based technologies we have listed the best grade applicable to any of the CORBA flavours mentioned in Sect. 6.3.6.

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-

	Analysable	Testable and debuggable	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 6.1: Grading of component technologies with respect to the requirements

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.

## 6.5 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 handling 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

# Bibliography

- [1] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proceedings of the 7<sup>th</sup> International Symposium on Component-Based Software Engineering*. 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.
- [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. Seattle, USA.
- [7] Microsoft Component Technologies. COM/DCOM/.NET. <http://www.microsoft.com>.
- [8] Sun Microsystems. Enterprise Java Beans Technology. <http://java.sun.com/products/ejb/>.



- [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. Pittsburg, USA.
- [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, J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. In *Real-Time in Sweden – Presentation of Component-Based Software Development Based on the Rubus concept, Arcticus Systems: <http://www.arcticus.se>*. Västerås, Sweden.
- [13] K.L. Lundbäck. Rubus OS Reference Manual – General Concepts. Arcticus Systems: <http://www.arcticus.se>.
- [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 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 Approach. In *The 2<sup>nd</sup> International Workshop on Composition Languages, in conjunction with the 16<sup>th</sup> ECOOP*, June 2002. Malaga, Spain.
- [17] T. Genssler, A. Christoph, B. Schuls, M. Winter, et al. PECOS in a Nutshell. PECOS project <http://www.pecos-project.org>.
- [18] M. Sgroi. Quasi-Static Scheduling of Embedded Software Using Free-Choice Petri Nets. Technical report, University of California at Berkely, May 1998. Berkely, USA.
- [19] O. Nierstrass, G. Arevalo, S. Ducasse, et al. A Component Model for Field Devices. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, June 2002. Germany.

- [20] 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.
- [21] Object Management Group. MinimumCORBA 1.0, August 2002. [http://www.omg.org/technology/documents/formal/minimum\\_CORBA.htm](http://www.omg.org/technology/documents/formal/minimum_CORBA.htm).
- [22] D.C. Schmidt, D.L. Levine, and S. Mungee. The Design of the tao real-time object request broker. *Computer Communications Journal*, Summer 1997.
- [23] CORBA Component Model 3.0. Object Management Group, June 2002. <http://www.omg.org/technology/documents/formal/components.htm>.

## **Chapter 7**

# **Paper C: Towards a Dependable Component Technology for Embedded System Applications**

Mikael Åkerholm, Anders Möller, Hans Hansson and Mikael Nolin  
To Appear in Pre-Prints of the Proceedings of the Workshop on Object-Oriented  
Real-Time Dependable Systems, Sedona, Arizona, USA, February 2005

### **Abstract**

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

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

## 7.1 Introduction

Software is central to enable functionality in modern electronic products, but it is also the source of a number of quality problems and constitutes a major part of the development cost. This is further accentuated by the increasing complexity and integration of products. Improving quality and predictability of Embedded Computer Systems (ECS) are prerequisites to increase, or even maintain, profitability. Similarly, there is a call for predictability in the ECS engineering processes; keeping quality under control, while at the same time meeting stringent cost and time-to-market constraints. This calls for new systematic engineering approaches to design, develop, and maintain ECS software. Component-Based Software Engineering (CBSE) is such a technique, currently used in office applications, but with a still unproven potential for embedded dependable software systems. In CBSE, software is structured into components and systems are constructed by composing and connecting these components. CBSE can be seen as an extension of the object-oriented approach, where components may have additional interfaces compared to traditional method invocation of objects. Similarly to objects, simpler components can be aggregated to produce more complex components.

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

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

algorithms to ensure that a feasible system is generated.

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

## 7.2 CBSE for Embedded Systems

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

**Software components** that have well specified interfaces, and are easy to understand, adapt and deliver. Especially for embedded systems, the components must have well specified resource requirements, as well as specification of other, for the application relevant properties, e.g., timing, memory consumptions, reliability, safety, and dependability.

**Component models** that define different component types, their possible interaction schemes, and clarify how different resources are bound to components. For embedded systems the component models should impose design restrictions so that systems built from components are predictable with respect to important properties in the intended domain.

**Component frameworks** i.e., run-time systems that supports the components execution by handling component interactions and invocation of the different services provided by the components. For embedded systems, the component framework typically must be light weighted, and use predictable mechanisms. To enhance predictability, it is desirable to move as much as possible of the traditional framework functionality from the run-time system to the pre-run-time compile stages.

**Component technologies** i.e., concrete implementations of component models and frameworks that can be used for building component-based applications. Two of the most well known component technologies are Mi-

Microsoft's Components Object Model (COM)<sup>1</sup> for desktop applications, and Sun's Enterprise Java Beans (EJB)<sup>2</sup> for distributed enterprise applications.

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

CBSE has been successfully applied in development of desktop and enterprise business applications, but for the domain of embedded systems CBSE has not been widely adopted. One reason is the inability of the existing commercial technologies to support the requirements of the embedded applications. Component technologies supporting different types of embedded systems have recently been developed, e.g., from industry [1, 2], and from academia [3, 4]. However, as Crnkovic points out in [5], there are much more issues to solve before a CBSE discipline for embedded systems can be established, e.g., basic issues such as light-weighted component frameworks and identification of which system properties that can be predicted by component properties.

Based on risks and requirements for applying CBSE for our class of applications, we have collected a check-list with evaluation points that we have used to evaluate our component technology in an industrial environment. In Section 5 we provide a summary of the evaluation, for more details we refer to [6].

### **7.3 Our Component Technology**

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

---

<sup>1</sup>Microsoft Corporation, The Component Object Model, <http://www.microsoft.com>

<sup>2</sup>Sun Microsystems, Enterprise JavaBeans Specification, <http://www.sun.com>

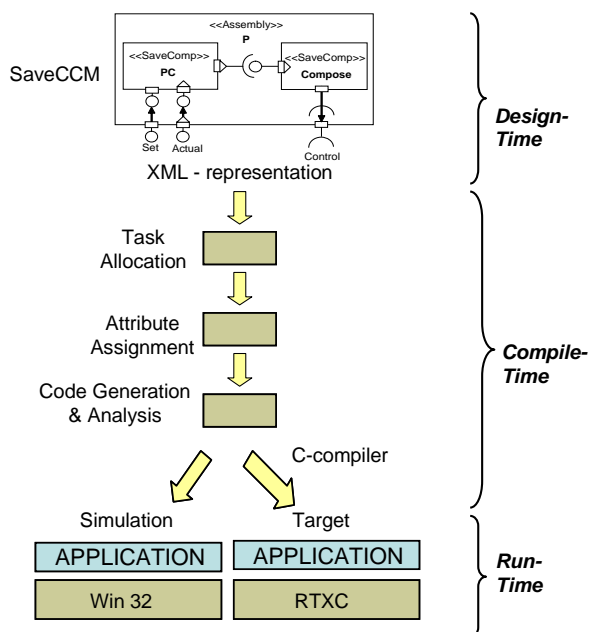


Figure 7.1: An overview of our current component technology

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

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

Run-time efficiency is important in embedded systems, since these systems usually are produced in high volumes using inexpensive hardware. We employ compile-time mappings of the component-based application to the used operating systems, which eliminates the need for a run-time component framework. As shown in Figure 7.1, three different phases can be identified, where different



pieces of the component technology are used:

**Design-time** SaveCCM is used during design-time for describing the application.

**Compile-time** during compile-time the high-level model of the application is transformed into entities of the run-time model, e.g., tasks, system calls, task attributes, and real-time constraints.

**Run-time** during run-time the application uses the execution model from an underlying operating system. Currently our component technology supports the RTX operating system<sup>3</sup> and the Microsoft Win32 environment<sup>4</sup>. The Win32 environment is intended for functional test and debug activities (using CCSimTech [15]), but it does not support real-time tests.

### 7.3.1 Design-Time - The Component Model

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

**Components** which are encapsulated units of behaviour.

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

**Switches** which allow static and dynamic reconfiguration of component interconnections.

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

- Support for implementation of feedback control, with a possibility to separate calculation of a control signal, from the update of the controller state. Something which is common in control applications to minimise latency between sampling and control.

---

<sup>3</sup>Quadros Systems Inc, RTX Kernel User's Guide, <http://www.quadros.com>

<sup>4</sup>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 distinguishes between these two aspects, and allow three types of ports:

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

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

The basis of the execution model is a control-flow (pipes-and-filters) paradigm [9]. On a high level, an element is either waiting to be activated (triggered) or executing. In the first phase of its execution an element read all its inputs, secondly it performs all computations, and finally it generates outputs.

### Components

Components are the basic units of encapsulated behaviour. Components are defined by an entry function, input and output ports, and, optionally, quality

attributes. The entry function defines the behaviour of the component during execution. Quality attributes are used to describe particular characteristics of components (e.g. worst-case execution-time and reliability). A component is not allowed to have any dependencies to other components, or other external software (e.g. the operating system), except the visible dependencies through its input- and output-ports.

### **Switches**

A switch provides means for conditional transfer of data and/or triggering between components. A switch specifies a set of connection patterns, each defining a specific way of connecting the input and output ports of the switch. Logical expressions (guards; one for each pattern), based on the data available at some of the input ports, are used to determine which connection pattern that is to be used.

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

### **Assemblies**

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

### **SaveCCM Syntax**

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

---

<sup>5</sup>Object Management Group, UML 2.0 Superstructure Specification, <http://www.omg.com/uml/>

<sup>6</sup>World Wide Web Consortium (W3C), XML, <http://www.w3.org/XML/>

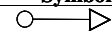
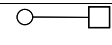
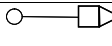
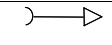
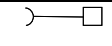
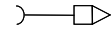
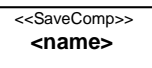
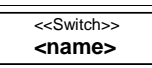
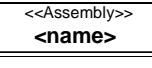
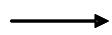
Symbol	Interpretation
	<b>Input port</b> - with triggering only
	<b>Input port</b> - with data only
	<b>Input port</b> - combined with data and triggering
	<b>Output port</b> - with triggering
	<b>Output port</b> - with data
	<b>Output port</b> - combined with data and triggering
	<b>Component</b> -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

Figure 7.2: Graphical syntax of SaveCCM

### 7.3.2 Compile-Time Activities

During compile-time, the XML-description of the application is used as input. The XML description contains no dependencies to the underlying system software or hardware, all code that is dependent on the execution platform is automatically generated during the compile-step. In the compiler, the modules (see Figure 7.1) that are independent of the underlying execution platform are separated from modules that are platform dependent. When changing platform, it is possible to replace only the platform dependent modules of the compiler.

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

### Task Allocation

During the task-allocation step, components are assigned to operating-system tasks. This part of the compile-time activities is independent of the execution platform, and the algorithm used for allocation of components to tasks strives to reduce the number of tasks. This is done by allocating components to the same task whenever possible, i.e. (i) when the components execute with the same period-time, or are triggered by the same event, and, (ii) when all precedence relations between interacting components are preserved. A description of the algorithm is available in [6].

### Attribute Assignment

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

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

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

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

**Deadline (D)** used during analysis.

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

### Analysis

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

Examples of analysis include schedulability analysis [10], memory consumption analysis [11], and reliability analysis [12].

Attributes that are usage and environment dependent cannot be analysed in this automated step, since it only relies on information from the component

model. There are no usage profiles or physical environment descriptions included in the component model. Additional information is needed to allow such analysis, e.g., safety analysis [13]. Safety is an important attribute of vehicular systems, and we plan to integrate safety aspects in future extensions.

In the current prototype implementation, schedulability analysis according to FPS theory is performed [14].

### **Code Generation**

The code generation module of the compile-time activities generates all source code that is dependent on the underlying operating system. The code generation module is dependent on the Application Programming Interface (API) of the component run-time framework. In the prototype implementation for the RTXC operating system (see Figure 7.3 right) and the Win32 operating system (see Figure 7.3 left), the code generation does not target any of the APIs directly. Instead, the automatic code generation generates source code for target independent APIs: the SaveOS and SaveIO APIs. The APIs are later translated using C-style defines to the desired target operating system.

### **7.3.3 The Run-Time System**

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

Pre-compilation settings are used to change the SaveCCM API behaviour depending on the target environment. If the application is to be simulated in a PC environment using CCSimTech [15], the SaveCCM API directs all calls to the SaveOS to the RTOS simulator in the Windows environment. If the system is to be executed on the target hardware using a RTOS (e.g. RTXC) the SaveCCM API directs all system calls to the RTOS.

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

hardware abstraction layer (HAL) is used. All communication between the component run-time framework and the hardware passes through the HAL.

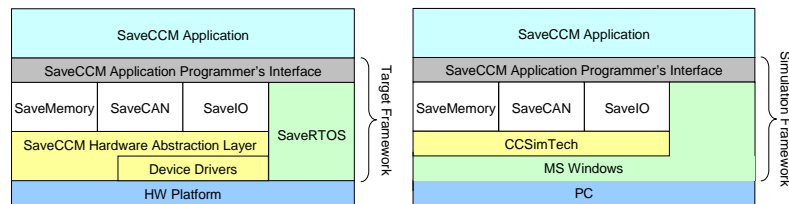


Figure 7.3: System architecture for simulation and target

The layered component run-time framework is designed to enhance portability, which is a strong industrial requirement [16]. This approach also enhances the ability to upgrade or update the hardware and change or upgrade the operating system. The requirements on product service and the short life-cycles of today's CPUs also make portability very important.

## 7.4 Application Example

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

### 7.4.1 Introduction to ACC functionality

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

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

#### 7.4.2 Implementation using SaveCCM

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

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

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

The ACC system is implemented as an assembly (*ACC Application* in left part of Figure 7.4) built-up from four basic components, one switch, and one sub-assembly. The sub-assembly (*ACC Controller*) is in turn implemented as shown in Figure 7.4, right.



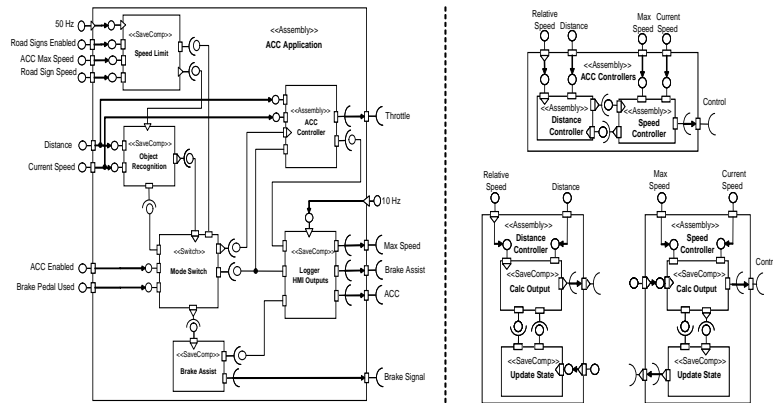


Figure 7.4: ACC Application implementation

### The ACC Application Assembly

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

The *Object Recognition* component is used to decide whether or not there is a car or another obstacle in front of the vehicle, and, in case there is, it calculates the relative speed to this car or obstacle. The component is also used to trigger *Mode Switch* and to provide *Mode Switch* with information indicating if there is a need to use the brake assist functionality or not.

*Mode Switch* is used to trigger the execution of the *ACC Controller* assembly and the *Brake Assist* component, based on the current system mode (*ACC Enabled*, *Brake Pedal Used*) and information from *Object Recognition*.

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

The *Logger HMI Outputs* component is used to communicate the ACC status to the driver via the HMI, and to log the internal settings of the ACC. The log-memory can be used for aftermarket purposes (black-box functionality), e.g., checking the vehicle-speed before a collision.

The *ACC Controller* assembly is built up of two cascaded controllers (see

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

The reason for using a control feedback solution between the two controllers is that since the calculation is very time critical, it is important to deliver the response (throttle lever level) as fast as possible. Hence, the controllers firstly calculate their output values and after these values have been sent to the actuators, the internal state is updated (detailed presentation can be found in [6]).

### 7.4.3 Application Test-Bed Environment

For the evaluation the RTX operating system was used together with a Cross FIRE ECU<sup>7</sup>. RTX is a pre-emptive multitasking operating system which permits a system to make efficient use of both time and system resources. RTX 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<sup>8</sup> IO-distributing ECU (Electronic Control Unit) designed for CAN-based real-time systems. The ECU is developed and produced by CC Systems, and intended for use on mobile applications in rough environments.

During functional testing and debugging, CC Systems use a simulation environment called CCSimTech [15], which also was incorporated in this work. Developing and testing of distributed embedded systems is very challenging in their target environments, due to poor observability of application state and internal behaviour. With CCSimTech, a complete system with several nodes and different types of interconnection media, can be developed and tested on a single PC without access to target hardware. This makes it possible to use standard PC tools, e.g., for debugging, automated testing, fault injection, etc.

## 7.5 Evaluation and Discussion

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

---

<sup>7</sup>CC Systems, Cross FIRE Electronic Control Unit, <http://www.cc-systems.com>

<sup>8</sup>Infineon, C-167 processor, <http://www.infineon.com>

representatives from the intended organisation, using existing tools, processes and techniques.

Our experiment was conducted using CC Systems' tools and techniques, however we have not used the company's development processes. Hence, we can only give partial answers (indications) concerning the suitability of 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 company's repertoire, and are thus highly realistic.

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

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

### 7.5.1 Structural Properties

Based on the experiment performed we conclude that the component model is sufficiently expressive for the studied application, and that it allows the software developer to focus on the core functionality when designing applications. The similarities with UML 2.0 provided important benefits by allowing us to use a slightly modified UML 2.0 editor for modelling applications. Also, issues

related to task mapping, scheduling, and memory allocation are taken care of by the compilations provided by the component technology. Further allowing the developer to concentrate on application functionality.

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

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

Configurability is essential for component reuse, e.g., within a Product Line Architecture (PLA) [18]. In SaveCCM, components can be configured by static binding of values to ports. However, there is currently no explicit architectural element to specify this. In our experiment, we could however achieve the same effect by directly editing the textual representation. For instance, a switch condition can be set statically during design-time, and partially evaluated during compile-time, to represent a configuration in a PLA. A future extension of SaveCCM is to add a new architectural element that makes it possible to visualise and directly express static configurations of input ports. This will additionally facilitate version and variant management.

### 7.5.2 Behavioural Properties

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

- The compilation resulted in four tasks: one task including components *speed-limit*, *object recognition*, and *mode-switch*; one task including *logger HMI outputs*; one task including brake assist; and one task including the four components in the ACC controller.
- The CPU utilisation in the different application modes are 7
- 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 support for verification and debugging of the application functionality.

### **7.5.3 Process Related**

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

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

## **7.6 Conclusions and Future Work**

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

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

Plans for future work include extending the component technology with support for multiple nodes, integration of legacy-code with the components [19], run-time monitoring support [20], and a real-time database for structured handling of shared data [21]. Implementation of more types of automated analysis to prove the concept of determining system attributes from component attributes is also a target for future work. However, there is also a need for methods to determine component attributes. Furthermore, to make the prototype useful in practice, there are needs for integrating our technology with supporting tools, e.g., automatic generation of XML descriptions from UML 2.0 drawings, and connectivity with configuration management tools.

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

#### **Acknowledgements**

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

# Bibliography

- [1] K.L. Lundbäck, J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. In *Real-Time in Sweden – Presentation of Component-Based Software Development Based on the Rubus concept, Arcticus Systems: <http://www.arcticus.se>*. Västerås, Sweden.
- [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 Consumption in Component-Based Software Systems. In *Proceedings of the 6<sup>th</sup> International Workshop on Component-Based Software Engineering*, May 2003. Portland, Oregon, USA.
- [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. Pittsburg, USA.
- [5] I. Crnkovic. Component-Based Approach for Embedded Systems. In *Proceedings of 9<sup>th</sup> International Workshop on Component-Oriented Programming*, June 2004. Oslo, Norway.
- [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, Mälardalen Real-Time Research Centre, 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 30<sup>th</sup> 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 the 7<sup>th</sup> International Symposium on Component-Based Software Engineering*, May 2004. Edinburgh, Scotland.
- [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 Architectures. In *Proceedings of 28<sup>th</sup> Euromicro Conference*, September 2002. Dortmund, Germany.
- [12] H.W. Schmidt and R.H. Reussner. Parameterized Contracts and Adapter Synthesis. In *Proceedings of the 5<sup>th</sup> International Conference on Software Engineering, Workshop on Component-Based Software Engineering*, May 2001. Toronto, Canada.
- [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 Systems. *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 7<sup>th</sup> International Symposium on Component-Based Software Engineering*. 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.



- [17] W. Lam and A.J. Vickers. Managing the Risks of Component-Based Software Engineering. In *Proceedings of the 5<sup>th</sup> International Symposium on Assessment of Software Tools*, June 1997. Pittsburgh, USA.
- [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 11<sup>th</sup> Asia-Pacific Software Engineering Conference, Workshop on Software Architectures and Component Technologies*, November 2004. Pusan, Korea.
- [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 Research Centre, Mälardalen University, May 2003. Mälardalen University Press.



## **Chapter 8**

# **Paper D: Monitored Software Components - A Novel Software Engineering Approach**

Daniel Sundmark, Anders Möller and Mikael Nolin  
In Proceedings of the 11<sup>th</sup> Asia-Pacific Software Engineering Conference, Work-  
shop on Software Architectures and Component Technology, pages: 624–631,  
Pusan, Korea, November 2004

### **Abstract**

We propose monitoring of software components, and use of monitored software components, as a general approach for engineering of embedded computer systems. In our approach, a component's execution is continuously monitored and experience regarding component behaviour is accumulated. As more and more experience is collected the confidence in the component grows; with the goal to eventually allow certification of the component. Continuous monitoring is also the base for contract checking, and provides means for post-mortem crash analysis; an important prerequisite for many companies to start use 3rd party component in their dependable systems.

In this paper we show how four software engineering goals can be reached by monitoring four component properties.

## 8.1 Introduction

In this paper we propose monitoring of software components and use of monitored software components as a general approach for engineering of embedded computer systems. Industrial developers of distributed, heterogeneous, reliable, resource constrained, embedded, real-time control systems (in this paper denoted embedded systems) are facing increased challenges with respect to demands on increased profitability, functionality and reliability, while at the same time having to decrease development times, project costs and time-to-market. Since development costs only constitute a fraction of the total project cost for software projects (about 20% [1]), a general approach for engineering embedded systems must consider not only the development phase; also the debugging, testing and maintenance phases need to be addressed. Furthermore, since most systems are developed incrementally, where new versions are based on previous versions, and product-line architectures [2] are becoming increasingly important, a general approach for engineering embedded systems needs to consider reuse of components between product versions and product variants. Another emerging key-issue in engineering of embedded systems is safe and predictable integration of third-party functions, and the associated legal matters regarding contract fulfilment and liability issues.

The main contributions of this paper are three-fold:

- We present a novel approach to engineering component-based systems, using monitored software components.
- Our approach takes a life-cycle perspective on the engineering process, and we identify four key-areas where monitored components will have significant impact.
- We present four measurable properties that can be used to impact these key-areas.

The outline of the rest of this paper is as follows: In Section 8.2 we present a life-cycle approach to engineering component-based systems, and Section 8.3 describes properties of embedded systems. In Section 8.4, we present a survey of related work in built-in monitoring support for component-based systems and existing monitoring practices in commercial component technologies. In Section 8.5, the impacts of monitorable components on predictable assemblies are discussed. Section 8.6 illustrates how to make use of the monitored information, and finally, in Section 8.7, we summarise and present our ideas on future work.

## 8.2 A Life-Cycle Approach to Component-Based Systems

This continuous increase of requirements for embedded-system developers can be mitigated by deploying suitable software engineering methods. It is our view that the whole system and component life-cycles need to be considered by an engineering method. Below we present four key-areas of engineering component-based systems where significant gains can be made by using our proposed concept of monitored components.

- **Certifiable components.** By monitoring component-based software, information about the component properties can be extracted. This information can be used to fully (or partially) describe the components by their externally visible properties. Such a description can be used as a basis for certifying components.<sup>1</sup> By reusing certified components, predictable component assemblies are facilitated.
- **System-level testing and debugging.** By monitoring individual components and component interactions, errors can be found and traced. Monitoring can also be used to support replay debugging [3], where erroneous system-executions are recreated in a lab environment to allow tracing of bugs.
- **Run-time contract checking.** This will allow surveillance of third party components. Both functional (e.g. range of output values) and non-functional (e.g. memory usage) properties can be monitored. During acceptance testing, the contract checking is used to validate that a component does not violate its specification. In systems that fail after system deployment, logs from the contract checking can be used in post-mortem analysis to identify failing or contract-breaking components.
- **Observability.** Computer systems in general, and embedded systems in particular, are infamous for the difficulty of observing their internal behaviour. This has drawbacks throughout the whole debugging, testing and maintenance phases. Systems whose behaviour is unobservable become very difficult to analyse and validate. Also after deployment, observability is an important feature, allowing inspection and performance tuning of running systems.

---

<sup>1</sup>For some high integrity systems, monitored properties need to be combined with static analysis to obtain safe bounds on properties.

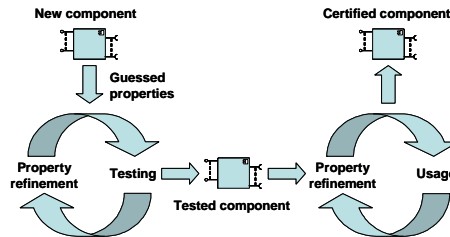


Figure 8.1: A conceptual overview of monitoring software components for certification.

The ultimate goal of component monitoring is to be able to compose predictable assemblies by reusing information gathered from well-tested software components. The proposal of this paper is that this can be achieved by the iterative process of refinement described in Figure 8.1. When a new component (or a 3rd party component) is included in an assembly, its run-time properties (such as execution time or memory consumption) are estimated by well-founded guesses. During testing, these guesses are validated and refined. As the tested component is deployed in a target-system assembly, its behaviour is continuously monitored, allowing for further refinement of the component run-time behaviour description. This refinement process will eventually lead to a certified component, which can be used to compose predictable assemblies.

Monitoring of components will allow information about the dynamic behaviour of the component to be recorded. This information allows static and dynamic properties of newly (or partly) constructed systems to be predicted. Interesting aspects to monitor (on component level) and predict (on system level) include timing properties, such as end-to-end response times, and resource utilisation, such as memory consumption.

## 8.3 Embedded Systems

This paper addresses software engineering aspects of resource-constrained, embedded, distributed real-time control systems. To make clear in what context the provided monitoring approach is supposed to work, we also provide a brief example of a typical embedded system and an introduction to component monitoring. In this section we also discuss prerequisites, placed on the component technology and on the hardware, to be able to monitor the embedded

system software.

### 8.3.1 CBSE for Embedded Systems

In Component-Based Software Engineering (CBSE), software applications are built by composing software components into component assemblies. CBSE is gaining more and more acceptance in the business segment of office/Internet applications [4, 5]. Unfortunately, the market segment of embedded real-time systems is, to a large extent, left behind this positive development. Reusing components, i.e. one of the main drivers for introducing CBSE, is both complex and expensive for embedded real-time systems [6].

However, by building embedded-system software out of well-tested components, we could gain an increase in the predictability of the behaviour of the software; provided that experience from component behaviour has been collected. In the area of embedded real-time systems, predictable run-time behaviour is crucial. A component assembly is predictable if its run-time behaviour can be predicted from the properties of its components and their patterns of interactions [7]. Predictability is achieved by analysis, and analysis techniques require information about the system. When analysing a system built from well-tested and functionally correct components, the main issues are associated with composability. The composition process should be able to guarantee the fulfilment of non-functional requirements of the system, such as communication, synchronisation, memory, and timing [6]. However, research projects tend to focus on how to design and analyse component technologies, leaving predictable assemblies using run-time information gathered from well-tested and trusted components unexplored [8].

### 8.3.2 Embedded System Example

In order to exemplify the typical settings, in which the software components are considered, we have studied some characteristic vehicular electronic systems [9]. An electronic vehicular control-system can be characterised as a resource constrained, safety-critical, distributed real-time system. The computer nodes, called Electronic Control Units (ECUs), are distributed to reduce cabling and to allow for division into subsystems. Vehicular systems are usually heterogeneous, meaning that nodes of different architecture and computational power cooperate in controlling the vehicle. The ECUs vary from extremely light-weighted nodes, like intelligent sensors (i.e. processor-equipped, bus-enabled



sensors), to PC-like hardware for non-control applications, such as telematics, and information systems.

Figure 8.2 gives an overview of the hardware resources of a typical ECU, with requirements on sensing and actuating, and with a relatively high computational capacity.

Example Power train ECU in a Vehicular Control-System	
>	Processor: 25 MHz 16-bit processor
>	Memory devices:
	✓ Flash: 1 MB used for application code
	✓ RAM: 128 kB used for the run-time memory usage
	✓ EEPROM: 64 kB used for system parameters
>	Serial interfaces: RS232 or RS485, used for service purpose
>	Communications: Controller Area Network (CAN) (one or more interfaces)
>	I/O: A number of digital and analogue in and out ports

Figure 8.2: Specification of a typical power train ECU.

An example of a typical vehicular system communication solution is shown in Figure 8.3, where two buses are separated by a gateway. The gateway is an architectural pattern that is used for several reasons, e.g., separation of criticality and real-timeliness, increased available bus bandwidth, increased fault tolerance, or compatibility with standards [10]. Communicating functions may require support for global synchronisation or fault tolerance mechanisms.

Looking at the software part of the system, there are some aspects that need to be considered when building the assembly out of monitorable components. A source of uncertainty is the frequency of interrupts in the assembly. Typically, a vehicular system is heavily loaded with interrupts. When interrupts hit the assembly, these will pre-empt the execution of the running component, thereby possibly perturbing its monitoring.

Dynamic memory allocation (and the garbage collection that this brings) is usually not allowed in control applications, since it compromises the determinism and predictability of the application behaviour. The only type of memory that is allowed to dynamically shrink and grow in the system is the stack space (albeit within a statically allocated stack memory area).

### 8.3.3 Prerequisites for Monitoring Component-Based Embedded Systems

Monitoring component-based software requires support in the component technology, and the framework used during run-time. Usually, when looking at

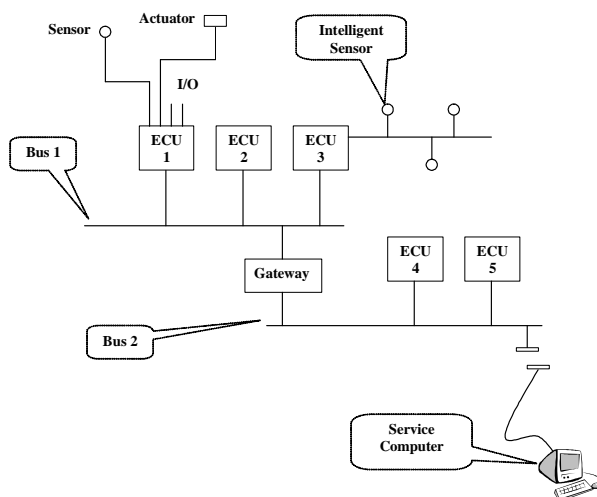


Figure 8.3: Example sketch of a vehicle network.

today's component technologies suitable for embedded systems with resource constrained ECUs, considerable code optimisations are done during compile time. This is mainly done to minimise the size of the application source code. This code optimisation might lead to a loss of the design-time component concept, meaning that clearly identifiable components with specified in- and outputs are reduced to regular source code functions, subjected to, e.g., function in-lining and redundant instruction-sequence coalescing.

Thus, to be able to monitor the components in the form described during design-time and to be able to reuse the information gathered during run-time in the next generation of applications, information about the design-time components have to be included in the source code. This should however not be a problem, if the component technology satisfies the requirements described in [9], i.e. a straight forward port-based object approach, illustrated in Figure 8.4, using a pipes-and-filters model of computation.

Component monitoring also poses some requirements on the system hardware. A fraction of the system memory needs to be allocated for monitor recording purposes. In addition, the target system should support some suitable means of communication through which monitor recordings can be uploaded to a host computer. Furthermore, some of the debugging techniques discussed in

this paper will benefit significantly from the support of an instruction counter.

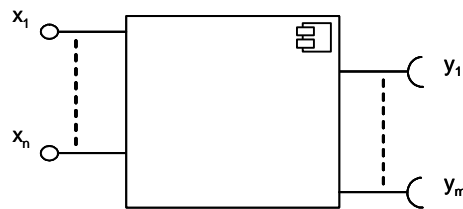


Figure 8.4: Component with required in-ports  $x_1 - x_n$  and provided out-ports  $y_1 - y_m$ .

## 8.4 Related Work

To summarise the available techniques that can be used to monitor software components, we have studied some commercial component technologies that include support for component monitoring and the state-of-the-art methods for component monitoring. The methods and the different technologies are described in Section 8.4.1 and Section 8.4.2.

### 8.4.1 Monitoring Techniques for Component-Based Systems

Currently, only a few component technologies provide support for run-time monitoring of component behaviour. However, there is numerous ways of performing this monitoring and there is a multitude of run-time aspects to monitor.

Gao et al. identify three different methods for component tracking and monitoring [8]: (A) framework-based code insertion, where monitoring code (e.g. from a class library) can be inserted by component engineers, (B) automatic code insertion, where monitoring code is inserted into the program by a specialised monitoring tool, and (C) automatic component wrapping, where monitoring code is automatically added to the external interface of components.

According to Gao et al., each of these methods has its own pros and cons. As for framework-based code insertion, it is highly flexible and can be used for all types of monitoring. However, the method requires access to the component source code, and the programming overhead is high. Automatic code insertion also requires access to the source code, and is much more complex and

inflexible compared to the framework-based code insertion. However, the programming overhead is low, since the tracking code is automatically inserted. Automatic component wrapping, on the other hand, has no need for component source code in order to insert tracking code. Therefore, not only in-house components, but also Commercial-Off-The-Shelf (COTS) components can be monitored. On the downside, automatic component wrapping is not suitable for monitoring anything within components, since the monitoring is performed exclusively outside the component.

Considering the use of these methods with respect to the restrictions posted by component-based embedded systems, it should be noted that automatic component wrapping can not be used in order to extract any component information other than that available at the component ports. This makes the method unsuitable for monitoring other component properties than those available from outside the component. Automatic code insertion, on the other hand, could be used for all types of monitoring, but would introduce a trade-off between the complexity of the instrumentation tool and the amount of data needed to record. Ideally, especially in resource-constrained systems, the amount of data to record should be minimised. However, this calls for an elaborate analysis of the internal workings of the component, requiring an inflexible (with respect to portability) and highly advanced instrumentation tool. Using framework-based code insertion, no instrumentation tool is required, allowing ad-hoc optimisations in the monitoring code. In a resource-constrained environment, this might be useful, but it must be kept in mind that such optimisations might lead to unpredictable probe-effects in system ordering and timing [11].

Jhumka et al. [12] propose the use of executable assertions in order to monitor component behaviour. The assertions are included in component wrappers, enabling them to test the validity of the input and output values of the component. By using these wrapper assertions, the pre- and post-condition sanity checks transforms a regular component into a fault-detecting component while at the same time simplifying unit-, integration- and system-level testing due to standardised means of extracting test information at component interfaces. Being relatively small and straightforward, executable assertions could well be used in order to perform sanity checks of embedded system components. However, executable assertions can not be used in order to monitor other properties, such as execution time or memory usage.

Hörnstein and Edler [13] propose the use of Built-In Test (BIT) components in the Component+ model [14] for reducing the time spent testing pre-fabricated components in new environments. In order to perform these built-in tests, the Component+ model makes use of three different types of components:

BIT Components, Testers and Handlers. BIT Components are regular software components with built-in test mechanisms, Testers are special components that use the BIT testing interfaces of the BIT components and Handlers are special components that can be used to obtain fault-tolerant systems by handling error signals from BIT or Tester components. On the assumption that BIT and Tester components are light-weighted, this can be an effective way of performing component sanity checks or run-time contract checking. Even though Handler components may be effective for achieving fault-tolerant systems, this is not the primary subject of this paper.

Traditionally, software monitoring can be performed by using either hardware- or software-based probes. Hardware probes come in the form of instrumentation tools, such as In-Circuit Emulators (ICE:s) or logic analysers, or in the form of System-On-Chip (SOC) solutions [15]. ICE:s or logic analysers are not suitable for component monitoring, since they cannot be included in deployed assemblies. SOC-based monitoring tools, however, are designed to be resident in deployed systems. Unfortunately, being designed for system-level event monitoring (e.g., task-switches), these tools are still far too inflexible for component-level monitoring. Therefore, today, software probes seem to be the preferred alternative for component monitoring. However, software-based monitoring is not without drawbacks. By including software monitoring in the component technology, we also introduce problems concerning instrumentation perturbation. Software-based monitoring is performed by means of software probes inserted in the code. These probes will consume execution time and memory space; increasing the spatial and temporal resource consumption of the components. Probes should be left permanently in deployed components for two reasons: (1) If the probes are removed, the testing performed on the component might no longer be valid [16], and (2) by leaving the probes in the deployed component, information concerning execution behaviour can be gathered over long periods of time, while the component operates in its field environment.

#### **8.4.2 Monitoring Support in Commercial Component Technologies**

There is a handful of available component technologies suitable for distributed embedded real-time systems. Some of these technologies include various supports for monitoring the software. The reason for choosing these is that they are deployed in industry today, and that they well satisfy the industrial requirements stated by the embedded-system domain [9].

The Rubus Component Model (CM) [17] and the Rubus Operating System (OS) have support for some of the described monitoring aspects. Rubus CM and OS are developed by Arcticus Systems<sup>2</sup> and are used for developing heavy vehicle software systems by, e.g., Volvo Construction Equipment<sup>3</sup> (VCE). When using the Rubus CM and OS, all resource allocation of the application and the operating system is done at compile-time.

The temporal properties needed to obtain static timing analysis and schedule generation, Best-Case Execution Time (BCET) and Worst-Case Execution Time (WCET), are monitored by the Rubus OS during run-time. Apart from the temporal aspects of the software, maximum stack usage for each thread and the peak usage of, e.g., queues can be monitored using Rubus. The OS also gives support for monitoring the CPU utilisation.

In multi-threaded embedded software, various types of relations, such as precedence and exclusion relations, exist. To be able to guarantee the behaviour of the system with respect to these issues, the Rubus CM includes support for monitoring event traces of the program execution, i.e., the execution order and the release times of the components. This information is dumped on an external interface (e.g., CAN or a serial interface like RS485) during run-time. Since events are related only via time-stamps, this service requires a high-resolution hardware timer. There will be a significant amount of data associated with this monitoring, and the accuracy of the log reflects the size of the buffer used to store it.

PECOS<sup>4</sup> (PErvasive COmponent Systems) [18] is a collaborative project between ABB Corporate Research Centre<sup>5</sup> and academia. The goal for the PECOS project is to enable component-based technology for embedded systems, especially for field devices, i.e., embedded reactive systems. The project tries to consider non-functional properties, such as memory consumption and timeliness, very thoroughly in order to enable assessment of the properties during construction time.

Non-functional properties cannot only be attached to components, but also to ports and connectors, e.g., examining the min and max values for an out port, (i.e., a built in sanity check). Since PECOS is developed to support resource constrained embedded real-time systems, scheduling information and memory consumption are crucial properties to monitor. Hence, PECOS enables support for instrumenting components during run-time. Every component is

---

<sup>2</sup>Arcticus Systems, [www.arcticus.se](http://www.arcticus.se)

<sup>3</sup>Volvo Construction Equipment, [www.volvo.com](http://www.volvo.com)

<sup>4</sup>The PECOS Project, [www.pecos-project.org](http://www.pecos-project.org)

<sup>5</sup>ABB Corporate Research, [www.abb.com](http://www.abb.com)

instrumented to extract information about the WCET and its cycle time. The components are also instrumented with respect to their code size and data (i.e., information on the heap).

## 8.5 Monitoring Software Components

Although a multitude of component properties are of interest when building reliable and reusable software components, there are some aspects that would significantly help increasing reusability and lower the time spent on integration testing. We have identified four main aspects to monitor, in order to support the key areas defined in Section 8.1.

### 8.5.1 Temporal Behaviour

Having knowledge of the temporal behaviour of an execution is particularly important for real-time systems. If the worst-case and best-case execution times of a set of reusable components are known, the possibility of successfully predicting the temporal behaviour of the component assembly will radically increase. Also, other execution time metrics, such as average execution time, standard deviation, execution time histogram or other types of statistical representations of component execution time behaviour can be helpful to estimate statistical temporal properties of component assemblies [19].

When considering timeliness for embedded real-time systems, it is important to be able to verify (1) that each component meets its timing requirements, (2) that each node (which is built up from several components) meets its deadlines, and (3) to be able to analyse the end-to-end timing behaviour of functions in a distributed system. In order to make sure that all deadlines are met, temporal analysis is needed.

This type of analysis is performed using schedulability analysis techniques, and requires information about the component's execution time. Ideally, the bounds for worst-case and best-case execution times should be statically computed by an analysis tool; this is the only way to be sure that the execution-time bounds are safe (i.e. guaranteed not to be violated at run-time), see e.g. [16]. Unfortunately, tools for execution-time analysis are immature and few commercial tools exist. Hence, the industrial practice is to rely on measurements of execution-times. However, structured measurement of execution-times is a tedious, error-prone and expensive process, which has to be re-done after each modification to a component. Using monitored components, the correctness of

the execution time values can be improved gradually, i.e., the more execution hours, the better the accuracy [19]; this is achieved without any extra effort for execution-time measurement.

In general, execution behaviour information is used for schedulability analysis and scheduling. In hard real-time systems, where it is mandatory that deadlines are met, deterministic schedulability analysis and scheduling (using worst-case assumptions for execution times) is preferable. However, in practice, many systems would settle for high probabilities instead of absolute deadline guarantees. Therefore, stochastic schedulability analysis and scheduling can be used. Depending on the type of analysis intended, either worst-case or statistical timing metrics should be collected during monitoring.

### 8.5.2 Memory Usage

Since we are targeting resource-constrained systems, it is important to be able to analyse the memory consumption and to check the sufficiency of the system memory, as well as the ROM memory. This check should be done pre-run-time to avoid failures during run-time. Memory is allocated in a static (pre-run-time or during run-time initialisation) or a dynamic (run-time) fashion. As mentioned in Section 8.3.1, dynamic memory allocation is usually not allowed when developing embedded real-time systems. In order to improve the possibility of achieving predictable assemblies, information of static memory allocation (e.g., component binary size) is necessary, but since this information can be provided by means of compiler output, this property is typically not necessary to monitor.

The stack memory, however, is statically allocated, but used in a dynamic fashion. In order not to end up in a stack overflow situation, stack size is often pessimistically over-dimensioned during system configuration. In resource-constrained environments, this might lead to a situation where the high percentage of unused memory leads to increased requirements on the system hardware. Therefore, monitoring the stack usage per component is most important, since this information can be used to predict the stack usage behaviour of future assemblies. Due to the high criticality of stack overflow, we are not interested in anything but worst-case usage during the execution of the component. However, in a system allowing dynamic allocation, also heap size monitoring would be important.



### 8.5.3 Event Ordering

When testing and debugging software, it is often helpful to be aware of the occurrence and ordering of system events, such as mutex- and semaphore operations, message receipts and interrupt occurrences. Using the information provided by an event log, system designers are able to detect improperly synchronised accesses to shared data or illegal pre-emption of non-reentrant code. In addition, by including event monitoring in the component model, we ensure that all components conforming to the model will produce event-trace logs of similar formats. This will reduce the problem of ad-hoc tracing code inserted by system developers.

Using event traces, we can gain substantial insight regarding the internal workings of current assemblies. This information can be used in order to guarantee precedence relations, mutual exclusion and to enhance the efficiency of shared resource usage (e.g., field bus or third-level storage usage) in future assemblies.

This type of monitoring provides a foundation to include full support for a replay debugging method [3, 20, 21] in the component technology. Replay debugging is a general term denoting methods for recording the execution behaviour of multi-tasking or truly parallel systems in order to use this information to reproduce system failures during debugging. Most replay methods require both event ordering information (such as interrupt, context switch and synchronisation information) and data flow information (such as task state and external input information) in order to reproduce executions. Provided that the assembly infrastructure (e.g., real-time operating system mechanisms) includes support for replay debugging, including sufficient monitoring in the components will ensure that the entire assembly can be debugged by means of execution replay.

### 8.5.4 Sanity Check

A sanity check is a way of determining the soundness of the functional operation of a component during run-time with respect to the component input and its current state. In other words, given a specific input, is the corresponding output realistic? During testing, having access to the input values of the component that produce erroneous output facilitates efficient testing.

Monitoring this during run-time will allow us to store erroneous operations of the component and (hopefully) to correct these errors in future assemblies. If we are unable to correct the faulty component, we could still be able to prevent

unsafe system behaviour by taking appropriate actions based on knowledge of the errors. This type of monitoring could also include properties like Mean-Time Between Failures (MTBF).

In addition, this type of monitoring could be used to ensure that 3rd party software components provide the service they are supposed to. Typically a component is equipped with a provided interface, specifying the services provided by that component, and a required interface, specifying the resources needed by the component in order to provide the correct services. Formalising and standardising these interfaces allows for contractual-based component development, where the behaviour of 3rd party components included in the assembly can be specified by contracts. Using sanity checks of the inputs and outputs at the component interfaces allows for run-time contract checking of 3rd party components.

## 8.6 Using Monitored Information

In Section 8.2, four key-areas that would benefit from component-level monitor support were listed. Table 8.1 maps these areas to the four monitored component aspects discussed in this Section. For instance, execution time- and memory information can be used in order to automatically check whether 3rd party components do not violate their required interface (e.g., by memory leaks or deadline misses), and sanity checks can be used to check the provided interface during run-time. By using event ordering and sanity check traces, the observability and ability to easily test and debug the assembly can be considerably enhanced. Regardless of whether replay debugging methods are used or not, event traces are helpful during debugging in order to visualise the behaviour of the component assembly during run-time.

As for certifiable components, all monitoring aspects can be helpful in order to successfully predict the future behaviour of components in different types of assemblies. Including monitoring in a component technology will ensure that all components conforming to that technology will include identical monitoring support. Hence, component properties can be easily compared using standardised means of comparison.

## 8.7 Conclusion and Future Work

In this paper we have proposed monitoring of software components, and reuse of monitored components, as a general approach towards engineering of re-

	Exec. Time	Memory	Event Ordering	Sanity Check
Certifiable Components	x	x	x	x
Debug/ Testing			x	x
R-T Contract Checking	x	x		x
Observability			x	x

Table 8.1: Mapping key-areas of interest to key component aspects.

source constrained, embedded, distributed, real-time control systems. The concept is general in the sense that it addresses not only the development phase; rather the whole product life cycle, including debugging, testing and maintenance, is considered. The concept also extends well into product-line settings, where components and architectures are reused over a set of related product and product variants. We have identified four key-areas within engineering of embedded systems: (i) certifiable components, (ii) system-level testing and debugging, (iii) run-time contract checking, and (iv) observability. We have also discussed how to meet the challenges within these areas, by identifying four main component-aspects that are of particular interest to monitor: (1) the execution time behaviour of the components, (2) the static and dynamic memory usage, (3) the event ordering of the execution and (4) a sanity check of the components output based on the input. We have provided a summary of the state-of-the-art of monitoring support for component models and presented a brief survey of the practices used in today's component models for embedded real-time systems.

As for future work, there are a number of issues we would like to address. We intend to look further into the problems of using the same component on top of different hardware platforms, where some old monitoring information might be reused, while other information needs to be discarded on the new platform. Furthermore, the trade-off between minimisation of monitoring memory and CPU usage and the level of detail of monitor information will be investigated. In order to evaluate our ideas, we plan to use the SaveComp Component Model, described in [22], as a test platform. The component model is designed for embedded control-systems in vehicles, and much focus has been spend on solving reliability, timelines and safety issues.

## Bibliography

# Bibliography

- [1] NIST Report. The economic impacts of inadequate infrastructure for software testing, May 2002.
- [2] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN 0-201-70332-7.
- [3] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *Proceedings of Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 288 – 295). ACM, April 2003.
- [4] Sun Microsystems. Enterprise Java Beans Technology. <http://java.sun.com/products/ejb/>.
- [5] Microsoft Component Technologies. COM/DCOM/.NET. <http://www.microsoft.com>.
- [6] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.
- [7] PACC Project Home Page. Home Page: <http://www.sei.cmu.edu/pacc>.
- [8] J. Gao, E. Zhu, and S. Shim. Tracking component-based software. In *Proceedings of the International Conference on Software Engineering, 2000's COTS Workshop: Continuing Collaborations for Successful COTS Development*, 2000.
- [9] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proceedings of the 7<sup>th</sup> International Symposium on Component-Based Software Engineering*. 2004 Proceedings Series: Lecture Notes in Computer Science, Vol. 3054, May 2004. Edinburgh, Scotland.

- [10] CiA. CANopen Communication Profile for Industrial Systems, Based on CAL, October 1996. CiA Draft Standard 301, rev 3.0, <http://www.canopen.org>.
- [11] J. Gait. A Probe Effect in Concurrent Programs. *Software – Practice and Experience*, 16(3):225 – 233, March 1986.
- [12] A. Jhumka, M. Hiller, and N. Suri. An Approach to Specify and Test Component-Based Dependable Software. In *Proceedings of the 7<sup>th</sup> IEEE International Symposium on High Assurance Systems Engineering*, pages 211 – 218, 2002.
- [13] J. Hörnstein and H. Edler. Test Reuse in CBSE Using Built-in Tests. In *Proceedings of Workshop on Component-based Software Engineering*, April 2002.
- [14] EC IST-1999-20162. Component+. [www.component-plus.org](http://www.component-plus.org), February 2004.
- [15] M. El Shobaki and L. Lindh. A Hardware and Software Monitor for High-Level System-on-Chip Verification. In *Proceedings of IEEE International Symposium on Quality Electronic Design*, pages 56 – 61, March 2001.
- [16] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *Software Tools for Technology Transfer*, 14, 2001.
- [17] K.L. Lundbäck, J. Lundbäck and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. In *Real-Time in Sweden – Presentation of Component-Based Software Development Based on the Rubus concept, Arcticus Systems: <http://www.arcticus.se>*. Västerås, Sweden.
- [18] T. Genssler, A. Christoph, B. Schuls, M. Winter, et al. PECOS in a Nutshell. PECOS project <http://www.pecos-project.org>.
- [19] T. Nolte, A. Möller, and M. Nolin. Using Components to Facilitate Stochastic Schedulability. In *Proceedings of the 24<sup>th</sup> Real-Time System Symposium – Work-in-Progress Session*. IEEE Computer Society, December 2003. Cancun, Mexico.

- [20] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471 – 482, April 1987.
- [21] K.-C. Tai, R.H. Carver, and E.E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering*, 17(1):45 – 63, January 1991.
- [22] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - a Component Model for Safety-Critical Real-Time Systems. In *Proceedings of 30<sup>th</sup> Euromicro Conference, Special Session Component Models for Dependable Systems*, September 2004.



