Component Based Software Engineering for Embedded Systems

A literature survey

Authors:

Johan Fredriksson
Jerker Hammarberg
Joel Huselius
John Håkansson
Annika Karlsson
Ola Larses
Markus Lindgren
Goran Mustapic
Anders Möller
Mikael Nolin
Thomas Nolte
Jonas Norberg
Dag Nyström
Aleksandra Tešanović
Mikael Åkerholm

June 2003

Editor:
Mikael Nolin

MRTC-Report no. 102
ISSN 1404-3041 ISRN MDH-MRTC-102/2003-1-SE

This work has been supported by:
SSF within the SAVE project,
Volvo and Vinnova wihtin the EAST/EEA project, and
KKS within the HEAVE project.

Contents

1 Introduction

This document describes a subset of the litterature that should be interesting for engineers and scientist that work within the area of software development for embedded systems. Papers that are decribed in this report have been selected since they, in some way, have bearing on the issues of Component Based Software Engineering (CBSE) of embedded systems and/or safety-critical embedded systems.

1.1 Background

The papers reviewed in this report have all been read, presented and discussed by the authors of this report during several meetings. Each paper has after the discussion been given a written review; as presented in this report.

The authors of this document are all participants of the SAVE-project, or of closely related projects. The SAVE-project's goal is to establish CBSE for safety critical systems as an engineering discipline. In addition to the review of each paper a comment on the paper's relevance to the SAVE-project is made. This comment should be valid for any reader that is interested in CBSE for safety critical systems.

1.2 Intended Audience

This report should prove useful for any engineer or scientist that whiches to learn more about the possibilities, limitations, and future of CBSE for embedded systems and CBSE for safety-critical systems. This report should give the reader an overview of the issues of interest and sufficient pointers to relevant litterature within the area.

Note, this report does not cover the technical details of any of the reviewed papers. Rather, the intention is to give enough information so that the reader should be able to judge the relevance of the paper and to find related litterature that could be of interest. Hence, this report could be seen as a starting point for someone who whiches to learn more about the subject at hand.

1.3 Structure of the Report

Sections 2 to 7 contains the paper reviews. The papers has been roughly categorised into the following sections:

- General – These papers treats CBSE, embedded systems or software trust in a general way. They define problems and terminology that are useful to be familiar with. These papers does not propose specific solutions or methods, rather

they deal with the problem of CBSE for embedded systems on a high level of abstraction.

- Techniques – These papers describes techniques that should be useful for CBSE for embedded systems. The described techniques are not geared towards any particular component model or component technology, rather they are generally applicable to the problem domain.

- Technology – These papers describe existing component models or component technologies, or contain evaluations of such. Both research component-models and industrially used component-models are covered here.

- Low Level – This section is somewhat similar to the "Techniques" section, in that the papers reviewed here describes techniques that could be generally useful for CBSE for embedded systems. Unlike the "Techniques" section, however, these techniques does not have direct bearing on CBSE, rather they deal with general problems of embedded systems and trusted systems. The term "Low Level" has been chosen to indicate that these techniques can bee seen as base techniques on top of which component-technologies can be built, or as techniques not related to CBSE but which must be considered in engineering of embedded and/or trusted systems.

- Architecture Description Language – While not directly related to CBSE, Architecture Description Languages (ADLs) are often considered as useful mechanisms to document the design of component-based systems. To establish CBSE for embedded systems as true engineering discipline, some form of higher level description of the systems architecture is needed, hence this brief section on ADLs is included in this report.

- Aspect-Oriented Development – Aspects and aspect oriented design/programming has recently received a lot of attention in the software engineering community. The motivation for studying aspect orientation in the context of CBSE for embedded systems is that aspects promises mechanisms to tune and configure components during compile time. For resource constrained and safety critical systems (which are typical classes of embedded systems) this type of off-line configuration can be very useful.

2 General

In this section we review papers that treats CBSE, embedded systems or software trust. They define problems and terminology that are useful to be familiar with. These papers does not propose specific solutions or methods, rather they deal with the problem of CBSE for embedded systems on a high level of abstraction.

2.1 Specification of Software Components

Lüders, Lau, and Ho [LLH02]

2.1.1 Introduction

Specification of software components is one of the most important research challenges in component-based software engineering. It represents the first step towards true component reuse as the component specification gives all necessary information to the component user on how/why the component can be (re)used.

2.1.2 Summary

The component in its simplest form contains code that can be executed on some platform and an interface that provides the access to the component. The authors of the paper take a traditional view where a component is considered to be a black box, i.e., its internals unaccessible to the component user. Hence, interfaces are the only access points to the component and the specification of the component comes down to the specification of the component interfaces.

Specification of the component interfaces in the current component-based systems, e.g., COM [Mic96] and CORBA [OMG02], is done only on the syntactical level. On this level, the component specification consists of specification of provided and required interfaces. Provided interfaces are the one that contain operations that a component provides to other components or to the component user, while required interfaces are the one that contain operations used by the component. Each interface consists of a number of operations which might have input, output or input/output parameters.

The component specification on a syntactic level does not provide information about the effects of invoking the operations in the component, except the ones that can be guessed by the name of the operations. Authors recognise the lack of semantic information in the component model, such as possible error codes, values that an operation accepts, and constraints on the order in which the operations are invoked. The problem of semantic component specification has been addressed in the paper by presenting the way a semantic specification has been done in an approach that combines UML [OMG99] and the Object Constraint Language (OCL) [WK98]. According to this

approach, additional constructs have been added to the syntactical component specification, such as preconditions and postconditions, as well as component state and invariants. Preconditions are assertions that the component assumes to be fulfilled before an operation is invoked, while postconditions are assertions that the component guarantees will hold just after the operation has been invoked, provided that the operations preconditions where satisfied. Pre- and postconditions depend on the state maintained by the component, and, in the component model, pre- and postconditions are specified over the component state. An invariant is a predicate over the interface's state that will always hold.

Syntactic and semantic component specifications are not enough when considering so called extra-functional component properties, e.g., reliability, temporal properties, and safety. To deal with the specification of the extra-functional component properties, authors propose the notion of credentials. A credential is a tuple: <Attribute, Value, Credibility, IsPostulate>, where:

- Attribute is a description of a properties of the component,
- Value is a measure of the property,
- Credibility is a description of how the measure has been obtained, and
- IsPostulate is a flag that is set if the credibility is replaced by the plan for obtaining the measure.

2.1.3 Reflections

Although the paper gives the description on how the component specification can be done on the syntactic and semantic level and provides a specification of extra-functional properties through the notion of credentials, it only scratches the surface of the very complex problem of component specifications. The notions introduced in the paper are necessary but not sufficient for specifying a component that can be reused in different reuse context. The notion of a black box component that has been introduced by the first component based systems, e.g., COM, CORBA, is adopted but misused in the paper: the authors themselves recognise that the specification of the component provided by the COM and CORBA component model is just a syntactic one, and needs to be expanded to include more details on the component behaviour thus, making the component more open as opposed to the COM/CORBA components. Credentials for handling extra-functional properties are good, but the authors present just a skeleton of how the extra-functional properties might be specified, and can be applied on a very small class of the extra-functional properties that can conform to the credential tuple specification.

The paper does not discuss nor does it recognises the problem of composing different components into a component-based system. The way to specify how a component

can be composed with other components and how it relates to other components is completely left out from component specifications, even though it represents a major part in determining the reuse contexts of components.

2.1.4 Relevance for the SAVE-Project

The problem of component specification has to be addressed by SAVE. This paper can be viewed as a vague guideline how the component specification could be done, and can be used by SAVE as a possible "skeleton" for the SAVE component specification, which must be enriched with more specification details, and real-time and embedded-specific issues. The notion of credentials should, within the SAVE component specification, be replaced with a stronger constructs that would provide a precise way of dealing with temporal requirements and analysis of the system behaviour. Moreover, composition requirements of each component should be defined in SAVE, since that is the prerequisite for enabling efficient system analysis with respect to temporal behaviour and safety of the composed system.

2.1.5 Interesting pointers

The paper references work from SEI [WS01] that could be interesting to follow up.

References

[LLH02] Frank Lüders, Kung-Kiu Lau, and Shui-Ming Ho. *Building Reliable Component-Based Software Systems*, chapter Specification of Software Components. Artech House, 2002.

[Mic96] Microsoft. The Component Object Model Specification, 1996. http://-www.microsoft.com/com/.

[OMG99] OMG. OMG Unified Modeling Language Specification, v1.3, June 1999.

[OMG02] OMG. CORBA/IIOP Specification, v3.0.2, 2002. formal/02-12-06.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML.* Addison-Wesley, 1998.

[WS01] K.C. Wallnau and J.S. Stafford. Ensembles: Abstractions for a New Class of Design Problem. In *Proc. of the IEEE 27th Euromicro Conference (Euromicro 2001)*, Warsaw, Poland, September 2001. IEEE Computer Society Press. http://www.sei.cmu.edu/pacc/WallnauStafford-ECBSE.pdf.

2.2 Embedded Software

Lee [Lee02]

2.2.1 Summary

The author notes that, the science of computing has systematically abstracted away the physical world. This is not feasible for embedded software because it normally interacts with the real world. A component architecture based on a principle called *actor oriented design* is suggested. The different actors interact according to a *model of computation*, and dynamic aspects, such as temporal properties, are expressed in the interfaces.

Embedded software is not just software executing on small computers. It is very much different because of properties intrinsic to embedded systems, for example the time a computation takes have to be known, it cannot be abstracted away. Embedded systems are normally reactive; highly concurrent, must not terminate, and mixes computational styles and implementation technologies.

In software engineering OO (Object Oriented) design has had great impact. It is the base of component technologies and it abstracts away the complicated implementation and exposes only interfaces through which the components can be called; but OO design captures only the static structures of the software. For an embedded component technology that is not sufficient, because the dynamic properties of the components also have to be described.

The author suggests "actor-oriented design", which is a new software-architecture approach to build systems from smaller parts (components). The components are parameterised actors with ports. Ports and parameters define the interface of an actor. A port represents an interaction with other actors. Its precise semantics depends on the chosen model of computation.

A model of computation [Lee01] governs the interaction of components in a design. In the real world the laws of physics would be the model of computation, i.e. the way things affect each other. In a model of computation for embedded software, concurrency and time have to be numbers one entities. Data-flow, synchronous/reactive, discrete events, process networks, rendezvous, publish and subscribe, continuous time, finite state machines, are all examples of models of computation. They all deal with concurrency and time in different ways and they all have their advantages and disadvantages, there is no "the model of computation". To design interesting embedded systems, different models of computation have to be mixed heterogeneously and hierarchically. What a composition of components with different models of computation should mean have to be dealt with carefully. In Ptolemy II [IHK+01] the composition is accomplished by a notion called *domain polymorphism*. It means that an actor (component) that can be executed by any of a number of models of computation is domain polymorphic, i.e. it has a well-defined behaviour in several domains. An aggregation

of components under the control of a domain should itself be a domain polymorphic component, and then the aggregate can be used as a component within a different model of computation. All domain polymorphic components in Ptolemy II have to implement a Java interface called the Executable. It defines three phases of execution: initialisation, iteration and termination. The iteration phase is also divided into three phases: prefire, fire and postfire. To get hierarchical mixtures of domains, a domain must itself implement the Executable interface.

The interface definition in Ptolemy has been developed in an ad-hoc manner; the author believes that type system concepts can be extended to make interface definitions more systematic. Then, all that is needed is that the properties of an interface be given as elements of partial order, e.g. the dynamic properties is given by automata and the partial order is the simulation relation between different automata. The result is called system-level type systems.

In OO the type system captures only the static structures of the software, in embedded systems that is not sufficient; the state trajectory and the concurrency is of outmost importance. Reflection on the system state allows for fault detection, isolation and recovery, i.e. if the system state is outside the safe region, the run-time type checking can detect it and take appropriate measures. This reflection could be done with reflection automata; an automata that simulate the system state. To develop such reflection automata, the author suggests state-oriented languages; a syntactic overlay on an OO language.

A framework is a set of constraints on components and their interactions. With this broad definition there exists a huge number of frameworks. As a rule frameworks tend to be very specialised with strong constraints in order to get strong benefits. Therefore frameworks have to be heterogeneously combined to solve all of the system aspects. Ptolemy is such a project.

2.2.2 Reflections

In the paper the concept *model of computation* is introduced. A model of computation governs the interaction of software components.

OO design is not very useful when developing embedded systems. It merely captures the static structures while the dynamic behaviour, which is of outmost importance when designing embedded systems, is not captured. The author describes a new term for describing refactored software architecture: *actor-oriented design*. Actor-oriented design can be seen as something quite similar to an ADL. Actor-oriented design is described as actors with ports through which interaction with other actors are done. The precise semantic of a port is given by a model of computation. In an ADL the components has ports through which they interact, and a connector represent the interaction between components, e.g. a model of computation. The dynamic behaviour

of components could be made accessible by using reflection automata. The automata model the dynamic behaviour of the system and which is comparable to the observer concept in control theory. Questions are however: Is this the way to do it? Will this not add a lot to the processing demands of the component?

Heterogeneity is a key concept in this paper. The authors state that different models of computation have to be mixed because they all have their advantages and disadvantages. This means that frameworks supporting different models of computation have to be mixed.

A way to compose components systematically by using type system concepts is discussed. This is probably a good idea and it is important to follow the progress. Some work on how to check compatibility between components with type checking has been done in [LX01]. It is with these results as a base that the authors suggest that it might be possible to use the same approach when composing components.

2.2.3 Relevance for the SAVE-Project

The relevance for SAVE is high. A discussion on how to compose components is raised in this paper. The Ptolemy project has been working with this problem and has come up with an ad-hoc solution. With that experience as a base a more systematic approach is suggested for future research.

References

[IHK+01] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Heterogeneous Concurrent Modeling and Design in Java. Technical Memorandum UCB/ERL M01/12 University of California, Berkeley, March 2001. http://ptolemy.eecs.berkeley.edu/publications.

[Lee01] E.A. Lee. Computing for Embedded Systems. In *Proc. of the 18th IEEE Instrumentation and Measurement Technology Conference (IMTC)*, volume 3, 2001.

[Lee02] E.A. Lee. *Advances in Computers*, volume 56, chapter Embedded Software. Academic Press, 2002. ISBN 0120121565.

[LX01] E. A. Lee and Y. Xiong. System-Level Types for Component-Based Design. In *Proc. of EMSOFT 2001*, LNCS 2211. Springer-Verlag, October 2001.

2.3 Component-Based Embedded Systems

Müller, Stich, and Zeidler [MSZ02]

2.3.1 Summary

The authors of the article Component-Based Embedded Systems have developed a comprehensive systematic approach to embedded software development. They have divided the development into five different categories by looking at different aspects of the development cycle through a series of case studies. Their prerequisites for an effective development are based on the Component Model, Component Architecture, Component Repository, Component Environment and the run-time system.

The authors claim that a central concept of components is the interface. Interfaces are often implemented as a pure collection of methods. Also events and attributes are finally modelled as interface methods. However embedded systems, especially embedded Real-Time systems, need to specify non-functional properties such as worst case execution time, power consumption, memory consumption, etc. Many component interfaces are implemented as object interfaces supporting polymorphism. Polymorphism is often supported through late binding, which often give performance penalties. Interfaces without polymorphism should be used for interfaces that can be determined at design time.

It is noted that, pre and post conditions are of great value for software quality, especially if they're checked during runt-time. However, embedded systems often have limited resources, and it may be much to resource demanding to do so during run-time. Alternatively design-time checking using a composition environment, which either simulates or calculates the correctness of connected components with given pre and post conditions, could be a viable method.

Deign time composition should be enough for the relatively static configurations of embedded systems because the resources are often limited. Design-time composition allows for optimisations, such as that interfaces could be translated into direct function calls. This requires some form of introspection or source-code components.

2.3.2 Comments and Relevance for the SAVE-project

The paper may have some relevance to the SAVE project because the authors emphasise some important issues when developing embedded component based systems. However, the paper has no clear specific relevance for the SAVE project. The paper could perhaps function as a guidline for the development process.

References

[MSZ02] P.O. Müller, C.M. Stich, and C. Zeidler. *Building Reliable Component-Based Software Systems*, chapter Component-Based Embedded Systems. Artech House, 2002.

2.4 Components in Real-Time Systems

Isovic and Norström [IN02]

2.4.1 Summary

The paper presents issues related to the use of component technology in the development of real-time systems. When applying CBSE (Component-Based Software-Engineering) methodologies in the development of real-time systems, important issues include reusability, flexibility, predictability and reliability.

Requirements regarding infrastructure-components (e.g. real-time operating systems and databases) and application-specific component models are introduced and discussed. Challenges using real-time components, such as guaranteeing temporal behaviour not only of the components but also of the entire system, are considered. Problems concerning composability are discussed with respect to timing properties. These timing properties, referred to as timing analysis, is divided into (1) verifying the timing properties of each component in the composed system, and (2) schedulability analysis.

A method for developing reliable real-time systems using a component-based approach is presented. It is a standard top-down development process to which timing and other real-time specific constraints have been added and precisely defined at design time. The method emphasises the temporal constraints that are estimated in the early design phase of the system and are matched with the characteristics of existing real-time components.

Finally, the paper provides guidelines about what one should be aware of when reusing and updating on-line real-time components.

2.4.2 Reflections

The paper deals with important aspects of CBSE for real-time systems. A lot of essential issues regarding components for real-time systems are emphasised, e.g., temporal requirements. The paper serve as a good foundation for understanding issues related to CBSE in the development of real-time systems.

The method proposed for designing component-based real-time systems seems to be reasonable but does not provide any actual news, except, perhaps, for the parts dealing

with timing analysis.

2.4.3 Relevance for the SAVE-Project

The paper can serve as a foundation for understanding issues related to CBSE in the development of real-time systems, but the paper is too brief to have practical relevance to SAVE.

References

[IN02] D. Isovic and C. Norström. Components in Real-Time Systems. Technical report, Departement of Computer Science and Engineering, MdH, 2002.

2.5 Specification, Implementation, and Deployment of Components

Crnkovic, Hnich, Jonsson, and Kiziltan [CHJK02]

2.5.1 Summary

In this article, Crnkovic et al. give an overview of CBSE (Component Based Software Engineering) as a relatively new sub-area of Software Engineering. The article points out that concept of CBSE originate from multiple other areas of research in software. Most important of those areas are: OO (Object Oriented) programming, reuse, software architecture, modelling languages and formal specifications. However, only the relationship to OO programming and (very short) reuse is clarified in the article.

The article starts by discussing some definitions of components. The three main discussion topics thought the rest of the article are specifications of components, patterns and frameworks.

In the specification of components, properties of components are divided to functional and extra-functional properties. Current component technologies only successfully address the syntactic part (syntactic interfaces) of the functional property specification, while the semantic aspects are still open area of research. The area of extra-functional properties is still rather unexplored.

The paper concludes by clarifying relationships among the following concepts:

- Frameworks and components,
- frameworks and contracts,
- frameworks and patterns,

- components and patterns, and
- contracts and interfaces.

2.5.2 Reflections

The three main topics in the article are specifications of components, patterns and frameworks; even though the title suggests: Specification, Implementation and Deployment. It is only mentioned that Implementation and Deployment are related to Frameworks and Patterns. The article is rather short (6 pages) and in this limited space, it can only mention the most important concepts. The most of the definitions are taken from the book [CL02], so the article can be seen as a compilation of the first chapters of the book.

2.5.3 Relevance for the SAVE-Project

The article is an easy to read introduction text to a newcomer to CBSE area. A better reading than this article is the initial chapters of the book [CL02]. The article also provides some relevant references for further reading [AA01].

References

[AA01] A. Arsanjaniand and J. Alpigini. Using grammar-oriented object design to seamlessly map business models to component-based software architectures. In *Proc. of the International Association of Science and Technology for Development*, May 2001.

[CHJK02] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Specification, Implementation and Deployment of Components. *Communications of the ACM*, 45(10), 2002.

[CL02] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

2.6 Trusted Componets for the Software Industry

Meyer, Mingins, and Schmidt [MMS98]

2.6.1 Summary

This paper talks about providing a solid foundation for the software industry by providing extensive libraries of trustable and reusable software-components. The main

issues discussed in the paper are concerning how trust is built in using a combination of approaches including:

- Use of "design by contract",
- formal validation and mathematical proof of correctness,
- extensive testing,
- wide public scrutiny (e.g. open source),
- evaluation through metrics,
- exhaustive validation in practical projects, and
- rigorous change management.

How are we supposed to build correct and reliable software, talking into account that the quality of our result depends on the quality of so much else, including hardware, operating systems, compilers and run-time libraries? The answer is in the combination of techniques chosen. By performing expensive testing together with academic formal methods and the recent push for reuse and "component ware" the quality of our software is increased; or is it? The major areas where effort is needed, according to the authors, are:

- Choice of areas for component development, starting humble, getting more advanced,
- base component development,
- language specific component adaptation,
- verification technology,
- testing technology for reusable components,
- assertion languages and other tools for providing practical components,
- development and application of property metrics,
- development of courses on reuse-based technology and applications, and
- identification of application areas and future technology along with tools for building quality component-ware.

2.6.2 Reflections

The paper seems to be a project description at an early stage, when there still not are any real results. Many interesting issues regarding motivating and building trusted components for the industry are discussed.

Since the document was published 1998 they might have produced some results that would be interesting to look deeper into.

2.6.3 Relevance for the SAVE-Project

This paper has no practical relevance to SAVE. However, as for gathering some directions of what are the important issues when pushing for components in industry, this paper has several points.

References

[MMS98] B. Meyer, C. Mingins, and H. Schmidt. Trusted Components for the Software Industry. *IEEE Computer*, May 1998.

2.7 Investigating the Influence of Formal Methods

Plegger and Hatton [PH97]

2.7.1 Summary

Formal methods promise much, but can they deliver? In the project described in the article, results are inconclusive but careful data gathering and analysis helped establish influences on product quality.

The project was the air-traffic control information system built by Praxis in the early 1990's for the UK Civil Aviation Authority.

The result of the analysis was that the whole system was highly testable and contained fewer high-complexity functions than the average system. This included both the formally and informally designed parts. The authors believe that this may be the result of a formal specification of the system. It is easier to find problems in highly testable code, so this is believed to be a good property of software.

The authors did not find any compelling quantitative evidence that formal design techniques *alone* produce code of higher quality, but in a combination of other techniques it can act as a catalyst.

2.7.2 Relevance for the SAVE-Project

The article itself did not address safety critical or real-time system, but the concept of formal methods is very relevant for SAVE. Formal methods can be used as a mean to prove that the implementation is conformant to its specification. This is very important when working with components, if you should be able to trust a component someone else has developed.

References

[PH97] S.L. Pfleeger and L. Hatton. Investigating the Influence of Formal Methods. *IEEE Computing*, 30(2):33–43, Feb 1997.

2.8 Separation of Concerns through Unification of Concepts

Nierstrasz and Achermann [NA00]

2.8.1 Summary

The idea of having unified concepts within a given domain is obviously an important issue for research as concepts are necessary for discussion and communication of problems and solutions. The idea of this paper is to establish some common concepts for modelling components and scripts. The specification language suggested by the authors is called Piccola and is briefly described in the introduction of the paper.

Change is identified as the biggest problem for software and it is the concerns for different types of change that the authors want to separate out and target with the modelling. In Piccola applications are built from components and scripts, where components are the stable parts of the system and the scripts are the non-stable, changing, part of the application. In the second section of the paper different types of change and measures to cope with change is discussed. From the discussion a set of requirements on component based systems are extracted.

Further, a set of techniques to meet the requirements of change is listed and it is proposed that the Piccola language support the necessary techniques. Each technique is described and the concepts of the Piccola language are related to the techniques.

In the concluding part it is recognised that Piccola is not yet a definite language, at the time of the paper (2000) three different implementations each with its own syntax existed. The work with Piccola is depicted as an ongoing research.

2.8.2 Reflections

One aspect of the Piccola approach is that the aim for the language seems to be on low-level implementations of IT systems and not on complete system design of embedded systems. The requirements brought forth in the paper includes issues like run-time composition which is deemed very hard to support in the highly resource constrained embedded environment.

Further, the definitions and concepts and how they are related are not complete and very vague in the paper. For a real evaluation of the Piccola language it would be necessary to use other references with more details. This might be problematic as it seems that there are three different implementations available. However, it is possible that by now there is a defined language since it has passed some years since the paper was written

2.8.3 Relevance for SAVE

Unified concepts and valid models is a highly important issue for SAVE. However, the Piccola language is not applicable for SAVE as the language is defined based on requirements for information systems and not for embedded applications. In total this article is of less relevance for the SAVE project.

References

[NA00] O. Nierstrasz and F. Achermann. Separation of Concerns through Unification of Concepts. In *14^{th} European Conference on Object-Oriented Programming*, 2000.

2.9 Component Testability and Component Testing Challenges *Gao [Gao00]*

2.9.1 Summary

In this article, Gao tries to explain the importance of testing and testability of components, and what things to be considered in the design of components in order to improve testability properties. Also, some challenges of the component testing are mentioned.

The testability of components is important because:

- It simplifies the test operations,

- it reduces test cost, and
- it increases software quality.

The following factors are mentioned as important for understanding component testability:

- Observability and traceability,
- controllability, and
- understandability

The following component based testing challenges are mentioned:

- Reusing component based tests,
- constructing testable components,
- constructing component test drivers and stubs in a systematic way, and
- building a generic and reusable (across languages and technologies) component test-bed.

2.9.2 Reflections

The overall structure of the document is rather poor and as well as the language used. The article misses introduction about testability in general. In summary, it can bee seen as two lists – important issues for component testability and a list of challenges. There is no summary in the article and neither reference for the future work. However it may serve its purpose, which is to bring the attention to testability of components issues to the rest of the CBSE (Components Based Software Engineering) community.

2.9.3 Relevance for the SAVE-Project

Even though the article can be seen as a rather poorly written article, it may serve as an initial input for the component testability work in SAVE. It is worth mentioning that component testability work has been identified as an open and important area in the road-map for testing area of software engineering for this decade [Har00].

2.9.4 Interesting Pointers

Interesting pointers on the topic include [Fre91, Bin94, VM95].

References

[Bin94] R. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9), 1994.

[Fre91] R.S. Freedman. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6), June 1991.

[Gao00] J. Gao. Component Testability and Component Testing Challenges. In *International Workshop on Component-Based Software Engineering*, May 2000.

[Har00] M.J. Harrold. Testing: A Roadmap. In *Proc. of the conference on The future of Software engineering*. ACM Press, 2000.

[VM95] J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 1995.

2.10 Technical Concepts of Component-Based Software Engineering

Bachmann et. al. [BBB+00]

2.10.1 Summary

The purpose of the paper is to establish a foundation for further research. The chosen course of action is to pronounce a vision, and from that derive key technical concepts of CBSE (Component Based Software Engineering) that should be emphasized. The vision is in brief that the properties of a system should be predicted from the properties of the components themselves, in analogy with other engineering disciplines. The key technical concepts needed to support this vision are identified in the paper and summarized below.

Component
The definition of a component is quite unclear since it exist too many attempts, and the interpretation of a single attempt may differ even within the community. In this paper they found the variations to be quite subtle, and tries to come up with an own definition which is consistent with the majority and own purposes. The brief definition is three points dealing with (1) a component as a black box (2) target for third party composition (3) supported by a component model.

Interface
CBSE relies heavily on interfaces. They must handle all those properties that lead to inter-component dependences, since the rest of the component is hidden for a developer. The hint is that although interfaces are familiar and has existed for several years, CBSE require more of an interface than earlier applications. Properties such as

behaviour, synchronization and quality of service (maximum response time, average response time and precision) might be good to express in the interface.

Contract
A reason for introducing contracts in the CBSE community is that a pronounced goal with components is substitutability. Interactions that define the context for which within the substitution occur can be very complex, utilizing contracts can be a good candidate to handle these substitutions, the paper suggest a metaphor interface contract which gathers those notations that are useful for CBSE. The use of contracts may also relive the burden of expressiveness in the interface, since a part of the properties may be expressed in the interface and another handled by the contract. The summarizing characteristics of an interface contract are: between two parties, possibility for parties to negotiate the details, prescribes measurable behaviour on all signatories and they cannot be changed without agreement from all signatories.

Component Model
The authors claim that the only way that a component can be distinguished from other forms of packaged software is through its compliance with a component model. Furthermore, no agreement on what should be included in a component model exists, but a component model should specify the standards and conventions imposed on developers of components. The proposal is that component models should deal with different component types, interaction schemes between components and clarify how different resources are bound to components.

Component Framework
A component framework can be imagined as a small operating system which components require. It is often implemented above a standard operating system, forming a middleware. A component framework typically supports one single component model, but based on arguments on what is needed to create a robust market for CBSE the authors argue for standardized frameworks for different application areas. They realize that different application areas have different requirements of a supporting component framework, but that as much as possible should be standardized. Two other possibilities are discussed, and all of them could be used in conjunction, one is making components easy to port between different frameworks, and another is making frameworks programmable to support different application areas.

Composition
The possibilities for composition should be defined by the component model, the base classes of possible interaction patterns should be component to component (C-C), component to framework (C-F, F-C) and framework to framework (F-F). Specializations can be applied to obtain more classes. Under composition, resource binding are also treated, in terms of early or late. The authors have interesting opinions on binding issues, they state the fact that early binding leads to efficient runtime interaction while late binding with dynamic naming services is resource demanding; but that the design constraints needed to support late binding leads to systems with properties that

are more readily analyzed and predicted prior to system assembly.

Certification
Certifications to the authors mean (1) the certified object fulfils a technical specification (2) a trusted agent or authority issues the certification. Objects to certify have traditionally been whole systems, but in CBSE we could also certify frameworks and components. The point would be that we could use a certification of a component as a guarantee that it fulfils some properties, and use that knowledge to predict properties of an assembled application.

2.10.2 Reflections

The paper is a good tutorial on basic technical concepts of CBSE, it contains several references on each topic and is easy to read. Regarding the authors own definitions, my impression is that they are clearly based on earlier research and are introduced only when necessary to make concepts clearer.

2.10.3 Relevance for SAVE

The paper is relevant for the SAVE project. It is an identification and definition of essential concepts, with descriptions derived from a vision addressing predictability, which is essential for SAVE. But what is missing for the SAVE project is the depth.

2.10.4 Interesting Pointers

The depth is missing, but perhaps the references on each topic could be used as a starting point for finding more information. The paper is quite long and so is the reference list, so all important topics and references are not mentioned. One intersting topic is interfaces, the paper have subtopics as extending APIs with extra functional properties [BJPW99] and specifying behaviour [Kra98], [GLD97]. The topic of utilizing contracts in CBSE is treated in [Mey92], [Ber98]. Another interseting topic is composition and the paper refer to [Szy98], [Pel99].

References

[BBB+00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, May 2000.

[Ber98] S. Berry. Programming the Middleware Machine with Finesse. In *Proc. of OMG-DARPA-MCC Workshop on Compositional Software Architecture*, Jan 1998.

[BJPW99] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999. Special Issue on Components.

[GLD97] S. J. Goldsack, K. Lano, and E. Durr. Invariants as Design Templates in Object-Based Systems. In *Proc. of the 1st Workshop on Component-Based System*, Dec 1997.

[Kra98] R. Kramer. The Java design by Contract Tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, 1998.

[Mey92] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.

[Pel99] C. Peltz. A Hierarchical Technique for Composing COM based Components. In *Proc. of the 2nd International Workshop on Component-Based Software Engineering (CBSE)*, May 1999.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

2.11 Component-Based Software Engineering for Resource-Constraint Systems: What are the Needs?

Hammer and Chaudron [HC01]

2.11.1 Summary

The paper identifies five requirements on a component model, which according to the authors must be met in order to establish a CBSE discipline for resource constrained systems. For each requirement the paper explains its relevance and discusses suitable solution directions. The discussed requirements are listed below.

- Loose coupling between components — Components decoupled in time and space, an engineer has more freedom when composing an application.
- Specification and verification of end-to-end constraints — Instead of expressing constraints on individual components (or part of end-to-end transactions), the engineer should be able to express constraints on end-to-end transactions.

- Specification of resource requirements for each component — A set of components is composable not only if their interfaces fit together, their total resource requirements should also match what the system offer.

- Dependable platform — Without dependable platform we cannot build a dependable application.

- Visual development environment — A component model is as useful as its supporting tools.

2.11.2 Reflections

The paper is short and well written; the authors early make clear that the purpose is to suggest directions for possible solutions, not to give the best solution to each requirement. Regarding the requirements themselves, resource constrained systems are a very broad categorisation. It could be a good idea to refine these requirements for more distinguished purposes and achieve requirements coupled with a more specific type of system. Looking upon the requirements with real-time systems in mind, the discussed requirements are quite obvious, however some the list is of requirements is not complete.

2.11.3 Relevance for SAVE

The paper is relevant for the SAVE project, although some of the discussed requirements are discussable for our purposes. When dealing with hard real-time environments it is more important to build deterministic applications than to be able to build them fast or easy, so for instance the loose coupling and visual development environment requirements are targets for discussion. The paper actually concludes by explaining that it is a coarse categorisation and that the requirements need to be discussed and refined.

2.11.4 Interesting Pointers

The paper starts to discuss composability problems that may arise when components do not support the same architecture style and refers to [GAO95]. Another potentially interesting reference is [Ham00], which is another paper written by one of the authors; the paper describes a method for verifying end-to-end transactions in the context of UML [OMG99].

References

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In *Proc. of the 17th International Conference on Software Engineering*, April 1995.

[Ham00] D.K. Hammer. Component-Based Architecting for Distributed Real-Time Systems: How to achieve composability? In *Proc. of Int. Symposium on Software Architectures and Component Technology (SACT)*, Jan 2000.

[HC01] D.K. Hammer and M.R.V. Chaudron. Component-Based Software Engineering for Resource-Constraint Systems: What are The Needs? In *In proc. Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, Jan 2001. position paper.

[OMG99] OMG. OMG Unified Modeling Language Specification, v1.3, June 1999.

3 Techniques

These papers revied in this section describes techniques that should be useful for CBSE for embedded systems. The described techniques are not geared towards any particular component model or component technology, rather they are generally applicable to the problem domain.

3.1 Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects

Stewart, Volpe, and Khosla [SVK97]

3.1.1 Summary

This paper is a case study showing how the software abstraction Port-Based Objects (PBO) can be used for implementation of a reconfigurable real-time system. The case is based on a project with reconfigurable robots at CMU, the Chimera project [SSK92].

A PBO is defined as an independent concurrent process. It communicates with other modules only through input and output ports, there is no synchronisation with other modules. The PBO have input and output ports for variables and also ports to exchange configuration constants and ports for communication with external resources that are not PBOs. A PBO with a well-defined set of ports can be used as a component in a library. Components can be combined if for each variable there is only one output port supplying the data, and for each input port there is an output port supplying the variable. A valid combination is called a configuration.

The intention is too achieve as loose coupling as possible, similar to the blackboard coupling of Hammer & Chaudron [HC01] (Section 2.11). The method uses a global table in shared memory that is replicated to a local table in each PBO by a predefined schedule in the RTOS services. The objects are integrated by a predefined framework

The purpose of the PBOs is to make it easier for control engineers to go from control design to implementation. Using the framework process for implementation it is possible to only insert control algorithms into code templates and then use the RTOS services to manage the components.

The approach enables rapid development, evolutionary design, tele-configuration of services and allows flexibility in implementation and configurations. But there are still questionable issues from a control point of view with dynamic reconfiguration that are not addressed in the paper.

3.1.2 Reflections

The paper covers an interesting case and has relevant discussions on both reasons for and properties of components for real-time systems. Still, as it is a case-study, no new central concepts are defined, instead the case helps to create an understanding of the meaning of some existing abstract concepts of component based software design. An unproportionally large section of the paper is used for analysis of the mechanisms regarding the inter-object communication through shared memory. This analysis is case specific and therefore few conclusions for the general case can be made.

3.1.3 Relevance for SAVE

The paper is interesting for SAVE as it is a case-study of an implemented real-time component based architecture, it does however not introduce new, broad and paradigm shifting concepts. The decomposition of the system using shared global variables managed by the RTOS is interesting. The notion of PBO is also a concept worth evaluating and comparing to other software abstractions for component based design.

The article is rather old (1997) and further research down the same track should be available and could be investigated.

3.1.4 Interesting Pointers

The paper list some publications on real-time object oriented systems design and development [BG92, SGW94, BJ96]. The concept of interface specification is outlined as critical to design of component based systems [PCW85], and the related issue of interface adaptation is also pointed to [Bea92, PA91].

References

[Bea92] B.W. Beach. Connecting Software Components with Declarative Glue. In *Proc. International Conference on Software Engineering*, pages 11–15. IEEE, IEEE Press, 1992.

[BG92] T.E. Bihari and P. Gopinath. Object-Oriented Real-Time Systems: Concepts and Examples. *IEEE Computer*, 25(12):25–32, Dec 1992.

[BJ96] B.A. Blake and P. Jalics. An Assessment of Object-Oriented Methods and C++. *Journal for Object-Oriented Programming*, 9(1):42–48, March-April 1996.

[HC01] D.K. Hammer and M.R.V. Chaudron. Component-Based Software Engineering for Resource-Constraint Systems: What are The Needs? In *In proc. Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, Jan 2001. position paper.

[PA91] J.M. Purtilo and J.M. Atlee. Module Reuse by Interface Adaptation. *Journal on Software: Practice and Experience*, 21(6):539–556, 1991.

[PCW85] D.L. Parnas, P.C. Clements, and D.M. Weiss. The Modular Structure of Complex Systems. *IEEE Trans. Software Eng.*, 11(3):259–266, March 1985.

[SGW94] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons, 1994.

[SSK92] D.B. Stewart, D.E. Schmitz, and P.K. Khosla. The Chimera II Real-Time Operating System for Advanced Sensor-based Control Applications. *IEEE Trans. Systems, Man, and Cybernetics,*, 22(6):1282–1295, November-December 1992.

[SVK97] D. Steward, R. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, December 1997.

3.2 The Specification of System Components by State Transition Diagrams

Broy [Bro97]

3.2.1 Summary

Broy introduces a new state transition formalism to describe and specify component behaviour. He claims that older formalisms are suffering either from being too heavy (e.g. Lamport [AL88] and Lynch [LT89]) and cumbersome to use in practice, or from being too simple and lacking precise formal semantics (e.g. Statecharts [Har87]). This new variant of state transition diagrams, being both powerful and easy to use as well as being backed up by rigorous semantics, combines the advantages of existing techniques.

Broy's state transition diagrams (STDs) consist of labelled nodes that represent control states and labelled arcs that represent transitions. A node is labelled with an identifier and a predicate over data variables that must be true in that control state. An arc is labelled with a transition rule having the following format:

{precondition} input pattern / output pattern {postcondition}

The precondition specifies a predicate over the data variables for the transition to be enabled. The input pattern specifies, for each input channel, a sequence of messages that must have been received for the transition to take place. The output pattern specifies, for each input channel, a sequence of messages to be produced when the transition takes place. Finally, the postcondition specifies the assignments to the data state by a predicate over the data variables at the time step following the transition.

In section 5, the state transition machines (STMs) that constitute the underlying formalism for the STDs are converted into stream processing functions, introduced in section 4, with unambiguous semantics. This is done by expressing the stream processing function in terms of elements of state transition machines. The conversion resolves the remaining ambiguities of the STM formalism.

In the end of the paper, the diagrams are informally extended with mechanisms for multithreading and hierarchical definitions.

3.2.2 Reflections

The state transition diagrams introduced by Broy are very powerful, and it should not be too difficult for engineers to make full use of them to define component behaviour. There is a tool called AutoFOCUS [Aut] that makes use of Broy's state transition diagrams. However, there seems to be much to do before they can make their way into the industry, especially considering the rapid development of other more commercially oriented tools and formalisms such as UML [OMG99]. In particular, there is nothing to be found in the paper concerning model checking and other formal analysis of STDs, and without contributions to this field, there seems to be little reason to introduce a new and intricate formalism.

References

[AL88] M. Abadi and L. Lamport. The Existence of Refinement Mappings. Technical Report 29, Digital Systems Research Center, August 1988.

[Aut] The AutoFocus Homepage. http://autofocus.in.tum.de/index-e.html.

[Bro97] Manfred Broy. The Specification of System Components by State Transition Diagrams. Technical report, Technische Universität München, Institut für Informatik, May 1997.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[OMG99] OMG. OMG Unified Modeling Language Specification, v1.3, June 1999.

3.3 Quality of Service Specification in Distributed Object Systems Design

Frølund and Koistinen [FK98]

3.3.1 Summary

The paper presents a language for specification of quality of service properties. The language is called QML, for Quality of Service Modelling Language. After describing the language, the paper uses an example to demonstrate that QML is useful in the design of distributed object systems.

The QML uses three constructs, *contract types* that define how properties are specified for some quality of service category, *contracts* that define required or provided values for properties, and *profiles* that bind contracts to interface elements.

By *refinement* a contract can be specified as extra requirements added to some other contract, much like how classes can be refined in an object-oriented language. Automatic *conformance checking* can be used to verify that provided values match those required by a contract. All domain knowledge needed for this is incorporated in the contract types, by specifying what values should be considered better than others.

3.3.2 Discussion

The method used in the paper for specification of values of extra-functional properties has its advantages, since it supports refinement and conformance checking. This makes it easy to create such specifications, and to verify that they conform to some requirements.

However, the scope of the paper is design of distributed object systems, so it deals mainly with the specification of interfaces and their implementations. The composition of properties is an entirely different subject matter, perhaps more specific to component based software engineering.

3.3.3 Relevance for SAVE

The specification of extra-functional properties is a definite issue for the SAVE project, and this method has its advantages - mostly in refinement, conformance checking, and independence of the property specified.

There are however standards that incorporate QoS specification, such as the *General*

Resource Modelling framework provided in the UML Profile for Schedulability, Performance, and Time Specification [OMG02]

3.3.4 Interesting Pointers

The paper mentions some related work that could be interesting to follow up [ZBS97].

References

[FK98] Svend Frølund and Jari Koistinen. Quality-of-Service Specification in Distributed Object Systems. *IOP Distributed Systems Engineering Journal*, pages 179–202, December 1998.

[OMG02] OMG. UML Profile for Schedulability, Performance and Time Specification, March 2002. OMG document number ptc/02-03-02.

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

3.4 Interface Automata

de Alfaro and Henzinger [dAH01]

3.4.1 Summary

The article presents a light-weight formalism for capturing the temporal aspects of software component interfaces that are beyond the scope of traditional type systems. Those type systems specifies interfaces only in terms of values and domains. The modeling language presented in this article is well suited for both design, documentation and validation. This last purpose makes it possible to make automatic checking of compatability between interfaces.

The traditional approach to composition is *pessimistic*. In this approach the environment may behave as it pleases, and the components are composable only if there is *no* environment that can lead the composition into an erroneous state. The authors opinion is that the *optimistic* approach is more natural since components are often designed under assumptions about the environment they will work in. Here the component interfaces are complemented with restrictions on the environment, and a composition is valid if there is *some* environment that can make them work together. When performing a composition both the the interface automata's and their environment restrictions are composed. The article shows that the algorithm for composing and has game-playing foundations that lead to an efficient algorithm for checking composability.

3.4.2 Relevance for the SAVE-Project

The ability to make automated compatability checks are interesting for SAVE, but the article never considers reusable components. If they would do that the optimistic approach is no longer the most natural way of looking at composition.

References

[dAH01] L. de Alfaro and T.A. Henzinger. Interface Automata. In *In 9^{th} Symp. Foundations of Software Engineering*, pages 109–120, 2001.

4 Technology

In this section we review papers describing existing component models or component technologies, or contain evaluations of such. A sample of both research component-models and industrially used component-models is covered here.

4.1 Component models and technology

Estublier and Favre [EF02]

4.1.1 Summary

In this paper the authors tries to assess the concepts and principles of different component technologies currently available in the industry.

Component technologies are focused on the last phases of the development process, and are therefore aimed at the implementation, deployment and execution. This makes it hard to understand the concept and principle of the underlying component model. The authors try to explain a set of component technologies with the terms of an ADL (Architecture Description Language); therefore they start by looking at ACME [GMW00], which is a typical ADL according to the authors. The goal is to raise the abstraction level above the implementation specific matters in order to assess the underlying component model.

The investigated component technologies are: Java Beans [SUN], COM+ [Mic96], CCM [OMG02], .NET [Mic], and OSGI [OSG]. Each one is investigated from five different aspects:

- interface,
- assembly,
- implementation,
- life cycle, and
- framework (run-time support).

4.1.2 Reflections

There seems to have been two goals with the paper: assesing the underlying component model of a number of investigated component technologies, and a comparison between ADLs and component technologies. The authors state that it might not be relevant and hard to compare component technologies and ADLs since the aim of the two approaches is quite different: ADLs focus on the early phases of the development

process and component technologies on the late. It is only in the conclusion part where a comparison between ADLs and component technologies is done. Note that it should be an ADL since they only investigate the ACME ADL.

The main goal of the paper was to assess the underlying component model of different component technologies. If found it will be possible to identify the basic building blocks of components and maybe develop a general component model.

Unfortunately the authors have not been successful in describing what they have found. They have identified five aspects (see the overview for a list) that will be investigated for each one of the component technologies. Where these aspects come from and the precise meaning of them are not explained; instead the reader have to figure out what the authors mean by reading and comparing the results of the investigated technologies.

4.1.3 Relevance for the SAVE-Project

Even though the paper does not reach the goal of identifying the underlying component models of the investigated component technologies, it is a good overview of component technologies used in the industry today.

References

[EF02] Jacky Estublier and Jean-Marie Favre. *Building Reliable Component-Based Software Systems*, chapter Component Models and Technology. Artech House, 2002.

[GMW00] D. Garlan, R. T. Monroe, and D. Wile. *Foundations of Component-Based Systems*, chapter ACME: Architectural Description of Component-Based Systems. Cambridge University Press, 2000.

[Mic] Microsoft. .NET Home Page. http://www.microsoft.com/net/.

[Mic96] Microsoft. The Component Object Model Specification, 1996. http://-www.microsoft.com/com/.

[OMG02] OMG. CORBA Component Model 3.0, June 2002. http://www.omg.org/-technology/documents/formal/components.htm.

[OSG] OSGI. OSGI Service Gateway Specification, Release 1.0. http://www.-osgi.org.

[SUN] SUN Microsystems. Introducing Java Beans. http://developer.java.sun.-com/developer/onlineTraining/Beans/Beans1/index.html.

4.2 The Koala Component Model

van Ommering [vO02]

4.2.1 Summary

The paper presents the Koala component-model developed by Philips. The component model is intended to be for software for consumer electronics, e.g., DVD-players, audio equipment, and television sets. Typically, such systems are resource constrained, using cheap hardware components, in order to keep the development costs low.

The paper describes how components are constructed, connected and deployed. The different interfaces and component binding technologies are presented. Furthermore the product-line architecture used at Philips is described. Finally, component management and versioning is discussed.

The Koala component is a lightweight component, realised in the C programming language. It has three different kinds of interfaces: requires, provides and diversity interfaces. The diversity interface is used to (i) parameterise the component to suit a particular system, and (ii) to allow optional interfaces, that is interfaces that may or may not be connected during run-time. The Koala component supports both link-time binding and late (dynamic) binding. A tool will automatically insert bind mechanisms if late binding is used.

4.2.2 Reflections

The Koala component model is interesting since it is developed and used in industry. The concepts used are straightforward. However, Koala is not directly applicable in safety-critical applications. Also, the paper is rather shallow and does not directly go into detail in any aspect of the component model.

4.2.3 Relevance for SAVE

The component model has relevance to SAVE since it is intended for a resource-constrained environment. The targeted systems have requirements in common with vehicle control systems, especially with respect to execution time and memory usage. Unfortunately, Koala does not cover extra functional properties like reliability and safety whatsoever. The paper reviewed in section 4.4 describes how such properties are considered for the Koala model.

However, concepts used in Koala should definitely be considered when designing the SAVE component-model.

References

[vO02] Rob van Ommering. *Building Reliable Component-Based Software Systems*, chapter The Koala Component Model, pages 223–236. Artech House Publishers, July 2002. ISBN 1-58053-327-2.

4.3 EAST/EEA relevant consideration of CORBA

Voget, Ditze et. al [VD+02]

4.3.1 Summary

The paper is a project internal report summarising the considerations done by the Body and Telematic groups of EAST/EEA about CORBA [OMG02a] and CORBA's possible relationships to EAST/EEA middleware. The considerations have shown that RTCORBA [OMG02c] and minimumCORBA [OMG02b] are two interesting concepts for middleware in real-time environments, and that it may be of interest for EAST/EEA to develop a domain specific CORBA specification.

CORBA is the acronym for Common Object Request Broker Architecture, OMG's [OMG] open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP [OMG02a], a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

RTCORBA is an optional set of extensions to CORBA tailored to equip ORBs (Object Request Brokers) to be used as a component of a real-time system. MinimumCORBA defines a standard, fully interoperable subset of CORBA functionality that is intended for resource-constrained systems.

4.3.2 Reflections

The paper gives a short and descriptive presentation of CORBA, RTCORBA and minimumCORBA but is not at all complete, leaving a lot of questions unanswered.

4.3.3 Relevance for the SAVE-Project

CORBA, in its original shape, is in most cases too memory and CPU consuming to be used in embedded real-time systems. MinimumCORBA could be of some interest to SAVE, but it would be even more interesting to see a domain specific CORBA specification, e.g., as proposed by the authors of this paper.

References

[OMG] OMG. OMG Home Page. www.omg.org.

[OMG02a] OMG. CORBA/IIOP Specification, v3.0.2, 2002. formal/02-12-06.

[OMG02b] OMG. Minimum CORBA, v1.0, 2002. formal/02-08-01.

[OMG02c] OMG. Real-Time CORBA Specification, v1.1, 2002. formal/02-08-02.

[VD+02] Sefan Voget, Michael Ditze, et al. EAST/EEA relevant considerations of CORBA. Technical report, EAST-EEA Internal Report, 2002.

4.4 Evaluation of Static Properties for Component-Based Architectures

Fioukov, Eskenazi, Hammer, and Chaudron [FEHC02]

4.4.1 Summary of Paper

The paper presents an extension of the Koala component-model [vO02]. This extension includes methods to specify and evaluate static properties of architectures. The example method presented in the paper calculates static memory consumption. In this approach, an extra interface, the reflective interfaces, is added to the component-model. In this interface, static properties can be specified (predicted properties can be used if the component is not yet implemented).

A tool to calculate the overall requirements of the system is used. It takes the components, the composition and compositional rules as input and derives an estimation of the composed property. Two methods can be used, the exhaustive method and the selective method. The selective method is useful when many properties are to be calculated on a complex architecture. It will produce a good estimation in a short period of time. The selective method calculates selected properties for a selected evaluation depth.

4.4.2 Reflections

The paper is rather short. Unfortunately there are many gaps in the logical thread, which leaves the reader with a number of unanswered questions. Heavy use of “home made” terminology does not make the paper clearer. The impression of the methods are that they are rather superficial. The tool to make static property evaluations seems rather trivial.

The fact that the selective method calculates properties for a selected evaluation depth is questionable. Intuitively, much of the memory used by the system could be located

in the basic components, since the compound component merely combines different subcomponents and use their services. The selective method excludes the "bottom" parts of the architecture, where possibly most basic components are located.

4.4.3 Relevance for SAVE

The ability to calculate and bound resource requirements of software for embedded safety-critical applications is crucial. In that respect is the approach taken in the paper relevant, however it seems, as stated earlier, like a rather trivial and superficial approach. Further work in this area is needed.

References

[FEHC02] A.V Fioukov, E.M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of Static Properties for Component-Based Architectures. In *Proc. of the 28th Euromicro Conference*, pages 33–39. IEEE Computer Society, September 2002.

[vO02] Rob van Ommering. *Building Reliable Component-Based Software Systems*, chapter The Koala Component Model, pages 223–236. Artech House Publishers, July 2002. ISBN 1-58053-327-2.

4.5 Packaging Predictable Assembly with PECT

Hissam, Moreno, Stafford, and Wallnau [HMSW02]

4.5.1 Summary

In this article, a technology called PECT (Prediction Enabled Certification Technology) by the Software Engineering Institute (SEI) is presented. PECT is designed to enable predictable assembly of certifiable components (correlation between these two). PECT can be viewed as a technology and a method for creating instances of the technology. Research objectives of the SEI-group working with PECT are to provide guidelines for constructing PECT instances in different domains and properties of interest.

A PECT instance is created by integration of Component Technologies (CT) and (one or more) Analysis Models (AM). CT is viewed as an already available distribution channel for created PECT instances. Relationship of CT to AM is 1:N. There are two different types of audiences for PECT instances – those who create a PECT instance and those that use the created instance.

Some more material on these and related topics, from the same authors, can be found on their web site and in the book [CL02].

4.5.2 Reflections

Some differences in the recent articles from the same research group point out that this is still on-going and unstable work with some unclear concepts and relationships. In absence of a standardised terminology in CBSE (Component Based Software Engineering), authors try to introduce some of the terminology themselves. Weaknesses of the PECT technology are mentioned at some places in the text but are not very easy to find – an example is that creating PECT instances by co-refinement is one of the most difficult tasks in creation and requires "judgement, experience and taste" from the designer. Relationship to component certification was mentioned but not discussed in more details in the analysed paper.

4.5.3 Relevance for the SAVE-Project

Work and concepts presented in this article are interesting and it may be very relevant to SAVE. A language called CL for writing glue code for assembling components is mentioned in the recent work from the same research group. It will be interesting to follow the advancements of this work at ICSE Components workshop in May 2003.

4.5.4 Interesting Pointers

An article by the same authors was published on the discussed topic [HMSW03], also some info is available on-line [PEC]. Other interesting references include [VP00].

References

[CL02] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[HMSW02] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Packaging Predictable Assembly. In *First International IFIP/ACM Working Conference on Component Deployment*. Springer-Verlag LNCS, June 2002.

[HMSW03] S.A. Hissam, G.A. Moreno, J.A. Stafford, and K.C. Wallnau. Packaging Predictable Assembly. *Journal of Systems and Software*, 3, 2003.

[PEC] http://www.sei.cmu.edu/pacc/.

[VP00] J. Voas and J. Payne. Dependability certification of software components. *Journal of Systems and Software*, 52, 2000.

4.6 Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems

Hatclif, Deng, Dwyer, Jung, and Ranganath [HDD+03]

4.6.1 Summary

Cadena is an integrated development environment for systems based on the Bold Stroke architecture. It provides development, analysis, and verification tools. An identified insufficiency that motivates the Cadena-tool is the current lack of scalable analysis techniques that allow automated analysis tools for component frameworks. The intention is that the tool should be used to investigate a variety of behavioural analysis techniques for component-based systems.

Bold Stroke is a reusable product-line software architecture used as the basis for avionics program on a range of Boeing production and experimental aircraft programs. The architecture target distributed real-time systems, and is based on the Corba Component Model (CCM) [OMG02].

The Cadena-tool provides:

1. Property specification for intra-component dependencies,
2. assembly description,
3. deployment facilities,
4. dependency analysis, and
5. model-checking infrastructure.

Output from the dependency analysis is usable as input to component execution-rate assignment, and node distribution. Execution-rate assignment affects the execution pattern for the components in the system, and is therefore a required input to the schedulability analysis. The model-checking infrastructure consists of a feature that can translate components specifications to a form accepted as input by a known model-checker.

4.6.2 Reflections

The paper is difficult to read and its contribution is questionable.

References to Cadena, Bold Stroke, and the Corba Component Model (CCM) are mixed without discrimination or separation. Further, technicalities are mixed with insights, seemingly at random, or at least without pedagogic concern for the reader.

The contributions of the paper are to our opinion two: (1) The insight that component-based systems are suitable for verification as some preparatory work is performed any way, and (2) the CCM-tool for specifying relations between CCM-components. The feasibility of the CCM-tool has also been proven by construction.

However, the Cadena-tool, presented as a contribution in the paper, is more of an umbrella for a number of existing products. Also, the presented motivation for Cadena, that it could be used for to investigate the effectiveness of a variety of behavioural analysis techniques for component-based systems, has been left for future work. Therefore, the remaining contributions of the paper are the two listed above.

4.6.3 Relevance for the SAVE-Project

More interesting then the Cadena-tool it self is the environment at which it is aimed: the Bold Stroke initiative has been successfully used in military aircrafts produced by Boeing. Unfortunately, the paper contains no references to Bold Stroke.

References

[HDD+03] John Hatclif, William Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-Based Systems. In *Proc. of the International Conference on Software Engineering*, May 2003.

[OMG02] OMG. CORBA Component Model 3.0, June 2002. http://www.omg.org/-technology/documents/formal/components.htm.

4.7 VEST: A Toolset For Constructing and Analyzing Component Based Operating Systems For Embedded and Real-Time Systems

Stankovic [Sta01]

4.7.1 Summary

In this paper the author propose to use component based techniques for developing embedded systems software, i.e., software for resource constrained systems. There are several motivations for why this should be done:

- Much work has been done on developing component technologies, such as COM [Mic96], CORBA [OMG], and Jini [AOS+99], but relatively little has been done on developing components that are suitable for embedded systems. Components in COM, CORBA, etc., do usually have too high memory and computing needs to be suitable for embedded systems.

- VxWorks [Win], Pebble [GSB+99], and MetaH [Ves97] do provide the entire micro-kernel as the infrastructure. And the choices for what to include/exclude in the kernel are rather coarse grained.

  It is the author's hypothesis that these infrastructures are too large and not specialised enough for many embedded systems.

- Little support is currently available for the configuration and composition process for component based systems. This support should include analysis techniques for composed components to verify that the composition fulfils some desired properties.

  Embedded system development do usually have strict demands on being able to predict properties that will exist in the system, e.g., end-to-end real-time requirements.

The goal of their work is to develop techniques that can be used for constructing 1) static and fixed embedded systems at low cost, 2) networked and long-lived embedded systems, and 3) have a reflective runtime environment to support a high degree of adaptability.

The selected approach is to start by constructing the infrastructure first, which should enable the construction of application-specific operating-systems. To support the construction of this software they have developed a tool called VEST (Virginia Embedded Systems Toolset).

4.7.2 VEST Overview

VEST is a tool that supports infrastructure creation, embedded system composition, and mapping of passive software components to active run-time structures (such as tasks). Reflective information, dependencies and aspects are stored in a library (more on this below). VEST does also have a set of analysis tools that can, for example, be used for real-time and reliability analysis.

To enable this VEST has a library of micro-components, components, and complete infrastructures. Micro-components are passive software such as interrupt handlers, dispatchers, and plug and unplug primitives. The tool does support analysis, dependency checks, and composition.

The dependency checks have been divided into four categories:

Factual are dependencies are those dependencies that are directly related to the component itself. Examples are WCET (Worst-Case Execution-Time), memory requirements, deadline, the code for the component, etc.

Inter-Component are dependencies between components. Examples are required interfaces, call graphs, precedence requirements, mutual exclusion, etc.

Aspects are dependencies that exist more or less on system level. Examples are end-to-end deadlines, performance, reliability, etc.

General . The authors themselves are not sure whether this group is required or not. They also note that the actual grouping of dependencies is not that important, but it is important that you do not forget about any of the important dependencies.

VEST extracts reflective information about the components and pass them to external tools for performing the actual analysis. It does this by formatting selected parts of the reflective information such that it conforms with what the external tool requires.

In the paper there are a couple of examples that show how these ideas can be applied for analysing real-time properties of composed components.

4.7.3 Relevance for the SAVE-Project

The paper does definitely have relevance to SAVE. However, they provide few direct answers on how to solve problems specific for embedded systems[1]. On the other hand there are some nice ideas and concepts in the paper.

The paper is more or less focused on real-time properties of composed components, but there should not be anything that hinders other kinds of analysis as well.

There are several questions that can be asked after reading this paper:

Component model . What component model are the authors using? How do we connect or compose these components?

Tailorability . Is there a need for OSs with such a tailorability as described in the paper? What is the target ROM or RAM size? What is the typical application area (except the wrist-watch example)?

Interface and implementation . How do they separate between interface and implementation? They provide an example where “create task”, “kill task”, and “set priority” are micro-components which we can select to include in the infrastructure or not. What is the interface towards the data structures that these operate on? (Typically each task does have a data structure where its data is stored.)

[1] [Sta01] is the single source of information we have used when writing this summary. Consequently we do not know if further progress have been made by the author.

References

[AOS+99] K. Arnold, B. O'Sullivan, R.W. Scheifler, J. Waldo, and A. Wollrath A. *The Jini Specification*. Addison-Wesley, 1999.

[GSB+99] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble Component-Based Operating System. In *USENIX Technical Conference*, pages 267–282, June 1999.

[Mic96] Microsoft. The Component Object Model Specification, 1996. http://-www.microsoft.com/com/.

[OMG] OMG. CORBA Home Page. http://www.omg.org/corba/.

[Sta01] John A. Stankovic. VEST — A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science*, 2211:390–??, 2001.

[Ves97] S. Vestal. Support for Real-Time Multi-Processor Avionics. In *Proc. 18^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 11–21, December 1997.

[Win] Wind River Systems Inc. VxWorks Programmer's Guide. http://www.-windriver.com/.

5 Low Level

This section is somewhat similar to the "Techniques" section (section 3), in that the papers reviewed here describes techniques that could be generally useful for CBSE for embedded systems. Unlike the "Techniques" section, however, these techniques does not have direct bearing on CBSE, rather they deal with general problems of embedded systems and trusted systems. The term "Low Level" has been chosen to indicate that these techniques can bee seen as base techniques on top of which component-technologies can be built, or as techniques not related to CBSE but which must be considered in engineering of embedded and/or trusted systems.

5.1 Safe and Reliable Computer Control Systems Concepts and Methods

Thane [Tha96]

5.1.1 Summary

The paper presents existing concepts and methods used in the design of safe and reliable computer control systems. The author also gives his opinions on what realistic expectations can be formed regarding the reliability and safety of a system when using some of these methods.

After and introduction and some definitions of terms, the paper describes the concepts regarding reliability. The methods described to achieve reliability are categorised into fault removal, fault avoidance and fault tolerance, depending on the strategy used. If the strategy for example is to find faults in a system (e.g. by testing) and then remove the fault, this is referred to as fault removal.

The next subject of the paper is that of safety. Safety is essentially reliability with a notion of consequences, so that failures leading to hazardous conditions can be ranked. This ranking can then be used to address the types of faults that could lead to potentially hazardous failures.

5.1.2 Discussion

The paper presents a good overview of the problems and techniques concerning safety and reliability, at least for someone new to the field. Overall the quality of the paper is good, but the subject is rather generic. This makes it hard to draw any conclusions on how safety and reliability could be achieved for systems assembled from components.

5.1.3 Relevance for SAVE

The concepts of safe and reliable computer control systems are for sure relevant for the SAVE project, but the methods that are briefly outlined in the paper would need to be evaluated in the context of components and composition.

References

[Tha96] Henrik Thane. Safe and Reliable Computer Control Systems – Concepts and Methods. Technical Report TRITA-MMK 1996:13, ISSN 1400-1179, ISRN KTH/MMK/R-96/13-SE, Mechatronics Laboratory, KTH, 1996.

5.2 A Ravenscar-Compliant Run-Time Kernel for Safety-Critical Systems

Lundqvist and Asplund [LA99]

5.2.1 Summary

Modelling a full Ada 95 runtime system using formal methods is not possible today. This work shows that it is possible to model the complete runtime system of the Ravenscar subset of Ada 95. In this work there are no restrictions on the functionality of either the protected objects or the delay queue. The authors have shown that the model of the runtime system fully behaves according to the semantics of Ada 95.

Ada is one of the most popular languages used when developing safety critical systems. However, until today it has only been nontasking subsets derived from Ada 83 available for safety critical systems. At the International Real-Time Workshop'97 a subset of Ada 95 was defined, the Ravenscar Tasking Profile [DB98, WB97], which offers a reliable tasking model and gives greater predictability to eventdriven systems. The objectives of the article is to show the correctness of the Ravenscar tasking profile for Ada 95.

Ravenscar is a welldefined subset of Ada 95 that includes tasking. In the following paragraphs some of the new features in Ada 95 are mentioned, but it is beyond the scope of this report to go into a detailed discussion of the semantics of these features. In Ada the core of the language is mandatory for all implementations. In addition a set of annexes for special needs is defined. Most annexes are optional extensions to the Ada 95 language and/or the runtime system (RTS). One of these annexes, the realtime annex, is mandatory for Ravenscar. Compared to Ada 83, Ada tasking has been extended with features such as requeue and asynchronous transfer of control in Ada 95. None of these new features are allowed in Ravenscar, but some others are e.g. protected objects (POs).

The key realtime features included in Ravenscar are:

- A fixed set of processes (Ada tasks), using fixed priority scheduling. Termination of tasks is not allowed in Ravenscar, and furthermore all tasks have to be declared at library level. Hierarchies of tasks are not permitted.
- Protected objects which provide asynchronous interprocess communication by a monitorlike mechanism, and supports priority inheritance to avoid priority inversion. The rendezvous mechanism is not allowed. All synchronisation and exchange of data between tasks is done by means of protected objects.
- Delay until operations, i.e. delay to an absolute time, allowing the implementation of periodic processes.
- Basic provision for releasing sporadic processes in response to interrupts.

The authors argue that, formal methods are necessary when it comes to achieving correct software, i.e. software that can be proven to fulfill its requirements. The tool used in the Manaproject for modeling and ver ification, of both the kernel and the application, is UPPAAL a tool box for modeling, simulation and verification of real time systems. The tool is developed jointly by Uppsala University and Aalborg University [LPY97]. A system model in UPPAAL consists of a collection of timed automata modeling the finite control structures of the system. A timed automaton is a finite automaton, extended with time by adding realvalued clocks and transition guards.

5.2.2 Reflections and Relevance for SAVE

The relevance for the SAVE project is quite low. The paper is for people familiar with Ada and Ravenscar. The paper more or less describes a tool, developed at Uppsala University, which has little or low relevance to SAVE.

References

[DB98] B. Dobbing and A. Burns. The Ravenscar Tasking Profile for High Integrety Real-Time Systems. In *SIGAda'98*, November 1998.

[LA99] K. Lundqvist and L. Asplund. A Formal Model of a Run-Time Kernel for Ravenscar. In *Proc. Sixth International Conference on real-Time Computing Systems and Applications*, pages 504–507, 1999.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UppAal in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[WB97] A. Wellings and A. Burns. Workshop Report, The Eighth International Real-Time Ada Workshop (IRTAW8). *Ada User Journal*, 18(2), June 1997.

5.3 A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology

Lang and Stewart [LS98]

5.3.1 Summary

This paper gives a comprehensive overview of existing exception handling techniques and what properties of these techniques are required in order to be useful in component-based real-time software systems (CB-RTS). Most modern programming languages and some operating systems today have dedicated mechanisms for exception handling, but none of them addresses the specific needs of CB-RTS. This fact serves as a motivation for development of new and better-suited techniques. The paper does not present any new mechanisms, but is nevertheless a good starting point and points out the directions for further research.

The main contribution is a list of criteria for evaluating whether a given mechanism is well suited for CB-RTS. Every criterion is well motivated. The list is divided into three groups of criteria: essential requirements, desirable requirements and performance goals. The essential requirements follow:

Reusability. The exception handler supplied with a component must be reusable in the same way as the component is reusable, and the user should not have to rewrite it.

Encapsulation. It must be possible to hide exceptions that are handled locally. This is essential in achieving a black-box view of components.

Predictability. Exception handling must be time-bounded since the components may be used in hard real-time systems.

Exception handler binding. The exception handler must be replaceable during runtime. This is important when the system is critical and cannot be shut down, but exception handling needs may differ for different environmental circumstances.

Distributed processing. The exception handling mechanism must support handling in another process than the one that raised the exception.

Totality. The mechanism must be able to handle all types of errors. Safety-critical components must not fail regardless of what error occurs.

Criticality Management. The software must be in control of the priority of the exception handling, so that important processes will not be blocked by less important handlers.

Consistency. Handlers must be able to clean up and restore the state if errors occur, since real-time systems normally continue to run in the presence of errors.

The paper also evaluates existing exception handling mechanisms, but surprisingly not directly with regard to the stated criteria. Instead, it is explained how the mechanisms realize a number of elements of exception handling. However, each of these elements is closely related to one or a few of the criteria.

5.3.2 Reflections

This paper will be useful in any work involving the development of a component model or framework where exceptions will be a part of the interaction between the components. The list of evaluation criteria can be used as a checklist when designing the exception handling mechanisms.

The paper is well written, easy to read and well structured so that it easily can be used as a look-up reference.

References

[LS98] Jun Lang and David B. Stewart. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274–301, March 1998.

5.4 On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems

El Sobaki [Sho02]

As a result of the ever-increasing chip sizes, a current trend in embedded systems is System-on-Chip (SoC); complete hardware systems are integrated to a single silicon dye. As much communication is performed between sub-systems on-chip, that information will never leave the chip. Hence, the observeability of the system decreases outside the chip. However, testing and debugging techniques rely on the observeability of the system [Hus02, TH99, Sch94].

The Multiprocess Application Monitor (MAMon) is a hardware monitor (i.e. a form of a probe) for SoC's. The solution assumes the use of Real-Time Unit (RTU) [L$^+$98], which is a hardware-enhanced real-time kernel. The probes of MAMon are implemented in a hybrid between hardware and software, as the probe is partly integrated on the dye as a sub-system, the required level of observeability can be maintained.

Monitored information is passed off-chip by MAMon, and can be logged in a database

on a remote node. Customised applications can then make of the extracted data to evaluate the system.

The RTU allows MAMon to record the occurrence of events such as task-switches, execution of service-calls, and interrupts. Via a software interface, the executing code can use MAMon to store information regarding the data-state of the system.

5.4.1 Reflections

The paper is well written, the contribution is clear. However, the detailed system requirements (e.g. the RTU-platform) makes comparison to other approaches difficult, and applicability questionable.

5.4.2 Relevance for SAVE

SAVE-systems must provide some level of observeability; this is necessary for to facilitate debugging and testing of SAVE-systems.

Probes can be implemented in hardware, software, or some hybrid of those two [Hus02]. Trade-offs to consider are perturbation and cost: Hardware probes do not have to perturb the system and can provide a more detailed view of the system. Software probes, on the other hand, can be portable between platforms, they are cheaper, and they have a larger level of abstraction. Thus, it can be discussed how probes are best implemented. However, in the case of component-based systems, it seems more suitable to introduce the concept of components to software probes as these can have a concept of the components in the system - hardware probes cannot. We note that it, as far as we know, is feasible to use software probes even in SoC's.

As it will imply a certain (type of) operating system and hardware architecture, and has no concept of the (potentially) complex inter-component dependencies of the software it monitors, it is our belief that MAMon itself is not usable for SAVE.

References

[Hus02] Joel Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[L+98] Lennart Lindh et al. Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems. In *the Seventh Swedish Workshop on Computer Systems Architecture*, June 1998.

[Sch94] Werner Schütz. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems*, 7(2):129 – 157, September 1994.

[Sho02] Mohammed El Shobaki. On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems. In *Proc. of the 8th International Conference on Real-Time Computing Systems and Applications*, 2002.

[TH99] Henrik Thane and Hans Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proc. of the 20th Real-Time System Symposium*, pages 360 – 369. IEEE, December 1999.

6 Architechture Description Language

While not directly related to CBSE, Architecture Description Languages (ADLs) are often considered as useful mechanisms to document the design and architecture of component-based systems. To establish CBSE for embedded systems as true engineering discipline, some form of higher level description of the systems architecture is needed, hence this brief section on ADLs is included in this report.

6.1 A Classification and Comparison Framework for Software Architecture Description Languages

Medvidovic and Taylor [MT00]

6.1.1 Summary

As software complexity increases there is a stronger need for a "higher level" design, or architecture, that captures the coarser grained architectural elements. Such high-level designs can, if used appropriately, aid in performing analysis of the system being built at an early stage, and hence also help in reducing the costs. (It is common knowledge that correcting problems late is more expensive than correcting problems early.)

A number of Architecture Description Languages (ADLs) have been proposed to support this "higher level" design, and these are being subject to quite intense research. However, there seems to be little consensus in the research community on what shall be supported by an ADL; what is an ADL and what parts should be modelled by an ADL. This paper attempts to clarify what an ADL is by performing an extensive survey on other peoples and their own work, and finding commonalities among the different approaches.

After having looked at what has been performed in the area the authors propose a classification and comparison framework for ADLs. This framework focuses on the following four elements of ADLs:

- Components,
- connectors,
- configurations, and
- tool support.

Within these categories they specify a number of items that shall be studied and compared. The items for components are interface, types, semantics, constraints, evolution, and non-functional properties. For connectors we have interface, types, semantics, constraints, evolution, and non-functional properties. For architectural configurations

they are understandability, compositionality, refinement and traceability, heterogeneity, scalability, evolution, dynamism, constraints, non-functional properties. Finally, for tool support there is active specification, multiple views, analysis, refinement, implementation generation, and dynamism.

For the above items they describe what the terms mean, but in this short summary we omit this description.

Using the just defined comparison and classification framework the authors move on and apply the framework to 10 different existing ADLs. The result is a number of tables that briefly describes the support that exist in the languages for all the items mentioned above.

6.1.2 Relevance for the SAVE-Project

Most systems do benefit from having some form of higher-level description of it, and if analysis can be performed on this description it is even better. In this sense the work in this paper is highly relevant for SAVE as well. SAVE does need some way of describing systems if we are to be able to perform analysis that can predict (parts of) the behaviour of the system.

This paper performs an extensive study of existing work within the area and it is clear that the authors know the area well. However, as a reader, there is some difficulty in understanding the grander picture since no overview of the different ADLs is presented. There are only short snippets from each language that focus on a single item. It seems to be more or less assumed that the reader is already familiar with some of the compared ADLs.

Therefore the contributions of the paper may be hard to apply directly in SAVE. On the other hand, there are several interesting references that might be worth to follow up (there are 74 references in total in the paper). One approach could be to first follow up the individual languages, e.g., [All97, AG94, GAO94, MDEK95, BEJV96, LKA+95, LV95, Mor, SDK+95, SDZ96, GR91] (to get an overview of the different languages), and then look at the cited previous work.

References

[AG94] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. Submitted for publication, May 1994.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[BEJV96] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, June 1996.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proc. of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

[GR91] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *Proc. of the 13th international conference on Software engineering*, pages 23–34. IEEE Computer Society Press, 1991.

[LKA+95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[Mor] Mark Moriconi. Architecture Refinement.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[SDZ96] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and Implementations for Architectural Connections. In *3^{rd} International Conference on Configurable Distributed Systems*, 1996.

6.2 Holistic Object-Oriented Modelling of Distributed Automotive Real-Time Control Applications

Axelsson [Axe99]

6.2.1 Summary

The automotive industry was late in introducing object-oriented (OO) techniques such as analysis, design, and programming because it was traditionally too expensive (in terms of computational resources) and there was also lacking support for some features specific to automotive applications.

However, due to the increasing complexity of automotive systems, OO techniques are needed and the author is suggesting an OO approach that is holistic in the following respects:

- It treats functionality in the vehicle level rather than on individual control units,
- it includes both the embedded computer-system and its environment, and
- it considers both hardware and software.

A new development process is presented in the paper and a supporting tool. This new process is compared to the traditionally used one and points of improvement are discussed.

The new tool acts as a repository (design database). Using this repository it is easy to edit the data using different kinds of graphical and textual editors. Analysis can be performed, e.g., execution time, cost (resource usage), and schedulability analysis. Furthermore it is possible to simulate the behaviour of the models of both the system and the environment. Synthesis is also supported, e.g., code generation.

6.2.2 Reflections

The paper is nice reading, presenting a holistic approach for the design of automotive systems using OO techniques. The tool presented in the paper is promising, however it is not yet implemented. Gathering some experiences using this tool would be interesting.

6.2.3 Relevance for the SAVE-Project

The paper might have some relevance for SAVE, although there should be some more recent work by the author.

References

[Axe99] J. Axelsson. Holistic Object-Oriented Modelling of Distributed Automotive Real-Time Control Applications. In *Proc. of the* 2^{nd} *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pages 85–92, Saint-Malo, France, May 1999. IEEE Computer Society.

7 Aspect-Oriented Development

Aspects and aspect oriented design/programming has recently received a lot of attention in the software engineering community. The motivation for studying aspect orientation in the context of CBSE for embedded systems is that aspects promises mechanisms to tune and configure components during compile time. For resource constrained and safety critical systems (which are typical classes of embedded systems) this type of off-line configuration can be very useful.

7.1 Component-based software engineering for distributed real-timesystems

Rastofer and Bellosa [RB01]

7.1.1 Summary

The paper describes a component model that separates the component functionality from platform-specific issues like concurrency, synchronisation, and distribution. The authors also try to show how to adapt the components to an execution platform and how to create real-time applications with predictable properties.

Applications are built from components in a platform-independent model, then mapped to an execution environment that introduce platform-specific features. The components are passive, and have a number of input and output ports. They communicate with other components by sending an event on one of its output ports.

The applications are assembled from components by connecting its output ports to input ports of other components. When a complete application is created, it still consists of platform independent components. All components have to be adapted to the target execution environment. This is done in multiple steps:

- Step 1
  - The graph of connected components is partitioned into sets of related components.
  - Components that communicate with each other should reside on the same node.
  - Additional location constraints arise because some of the components only work on the node that have the physical devices attached.
  - All paths that span multiple nodes are cut up, so that only local paths remain.
- Step 2

  - On each node, there is a set of local paths. These paths are mapped to tasks of the local execution platform.

- Step 3
  - Find a schedule for the local set of tasks, by finding the tasks WCET and by assigning each task a priority.
  - The global schedule, including the schedule for the communication between nodes has to be checked.

- Step 4
  - When a valid schedule is found, all components have to be adapted to the platform and to the application.

7.1.2 Reflections

The paper deals with interesting questions about building a platform-independent component model by separating component functionality from concurrency, synchronisation, and distribution. The paper is a bit short, leaving many questions unanswered.

7.1.3 Relevance for the SAVE-Project

The paper does definitely have SAVE relevance. Building applications from components in a platform-independent component model, that separates the components functionality from the platform specific issues of concurrency, synchronisation, and distribution, provides flexibility and reusability.

References

[RB01] U. Rastofer and F. Bellosa. Component-based software engineering for distributed embedded real-time systems. *IEE Proc. online no 20010536*, 2001.

7.2 From Contracts to Aspects in UML Designs

Jézéquel, Plouzeau, Weis, and Geihs [JPWG]

7.2.1 Summary

The paper deals with the effort to apply aspect-oriented software development [KLM+97] to the design level, by modularizing systems design crosscutting concerns into aspects.

The paper contributes with the idea of extending aspect-orientation to the software modeling level. In order to do that, the authors have chosen the UML modeling language [OMG99], and the notion of quality of service contracts as an infrastructure to build these ideas upon. Quality of service contracts represent the specifications of non-functional constraints expressed in the Quality of service Modeling Language (QML) [FK98]. The quality of service contracts, when applied on the design of components, typically crosscut component designs as each of the contracts has to be applied to a component interface. The authors of this paper find it useful to abstract these cross-cutting design issues expressed by contracts on the interfaces of the components, and implement them in terms of aspects. Here, aspects are designs that are independent of a programming language and fairly independent of the component specifications, i.e., aspects can be reused with different components or component configurations. It is claimed in the paper that encapsulating quality of service contracts in aspects improves reuse and maintainability of designs.

7.2.2 Reflections

Although the idea of extending aspect-orientation to the design level is appealing, the paper lacks a strong motivation for doing so. It is a short paper very focused on solving a big problem, but using very limited examples and short explanations to lead the reader into understanding the importance of their contributions.

7.2.3 Relevance for the SAVE-Project

Unfortunately, this paper does not have any SAVE relevance. However, the authors reference some of their own earlier work [BJPW99] that could be interesting for SAVE to pursue.

References

[BJPW99] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999. Special Issue on Components.

[FK98] S. Frølund and J. Koistinen. QML: A Language for Quality-of-Service Specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories, February 1998.

[JPWG] J-M. Jézéquel, N. Plouzeau, T. Weis, and K. Geihs. From Contracts to Aspects in UML Designs. http://www.qccs.org/aodcontracts.pdf.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira L., J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *11^{th} European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[OMG99] OMG. OMG Unified Modeling Language Specification, v1.3, June 1999.

References

[AA01] A. Arsanjaniand and J. Alpigini. Using grammar-oriented object design to seamlessly map business models to component-based software architectures. In *Proc. of the International Association of Science and Technology for Development*, May 2001.

[AG94] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. Submitted for publication, May 1994.

[AL88] M. Abadi and L. Lamport. The Existence of Refinement Mappings. Technical Report 29, Digital Systems Research Center, August 1988.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[AOS+99] K. Arnold, B. O'Sullivan, R.W. Scheifler, J. Waldo, and A. Wollrath A. *The Jini Specification*. Addison-Wesley, 1999.

[Aut] The AutoFocus Homepage. http://autofocus.in.tum.de/index-e.html.

[Axe99] J. Axelsson. Holistic Object-Oriented Modelling of Distributed Automotive Real-Time Control Applications. In *Proc. of the* 2^{nd} *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pages 85–92, Saint-Malo, France, May 1999. IEEE Computer Society.

[BBB+00] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, May 2000.

[Bea92] B.W. Beach. Connecting Software Components with Declarative Glue. In *Proc. International Conference on Software Engineering*, pages 11–15. IEEE, IEEE Press, 1992.

[BEJV96] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227, June 1996.

[Ber98] S. Berry. Programming the Middleware Machine with Finesse. In *Proc. of OMG-DARPA-MCC Workshop on Compositional Software Architecture*, Jan 1998.

[BG92] T.E. Bihari and P. Gopinath. Object-Oriented Real-Time Systems: Concepts and Examples. *IEEE Computer*, 25(12):25–32, Dec 1992.

[Bin94] R. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9), 1994.

[BJ96] B.A. Blake and P. Jalics. An Assessment of Object-Oriented Methods and C++. *Journal for Object-Oriented Programming*, 9(1):42–48, March-April 1996.

[BJPW99] A. Beugnard, J-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, July 1999. Special Issue on Components.

[Bro97] Manfred Broy. The Specification of System Components by State Transition Diagrams. Technical report, Technische Universität München, Institut für Informatik, May 1997.

[CHJK02] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan. Specification, Implementation and Deployment of Components. *Communications of the ACM*, 45(10), 2002.

[CL02] I. Crnkovic and M. Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. ISBN 1-58053-327-2.

[dAH01] L. de Alfaro and T.A. Henzinger. Interface Automata. In *In 9^{th} Symp. Foundations of Software Engineering*, pages 109–120, 2001.

[DB98] B. Dobbing and A. Burns. The Ravenscar Tasking Profile for High Integrety Real-Time Systems. In *SIGAda'98*, November 1998.

[EF02] Jacky Estublier and Jean-Marie Favre. *Building Reliable Component-Based Software Systems*, chapter Component Models and Technology. Artech House, 2002.

[FEHC02] A.V Fioukov, E.M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of Static Properties for Component-Based Architectures. In *Proc. of the 28th Euromicro Conference*, pages 33–39. IEEE Computer Society, September 2002.

[FK98a] S. Frølund and J. Koistinen. QML: A Language for Quality-of-Service Specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories, February 1998.

[FK98b] Svend Frølund and Jari Koistinen. Quality-of-Service Specification in Distributed Object Systems. *IOP Distributed Systems Engineering Journal*, pages 179–202, December 1998.

[Fre91] R.S. Freedman. Testability of Software Components. *IEEE Transactions on Software Engineering*, 17(6), June 1991.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proc. of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch, or, Why it's hard to build systems out of existing parts. In *Proc. of the 17th International Conference on Software Engineering*, April 1995.

[Gao00] J. Gao. Component Testability and Component Testing Challenges. In *International Workshop on Component-Based Software Engineering*, May 2000.

[GLD97] S. J. Goldsack, K. Lano, and E. Durr. Invariants as Design Templates in Object-Based Systems. In *Proc. of the 1st Workshop on Component-Based System*, Dec 1997.

[GMW00] D. Garlan, R. T. Monroe, and D. Wile. *Foundations of Component-Based Systems*, chapter ACME: Architectural Description of Component-Based Systems. Cambridge University Press, 2000.

[GR91] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *Proc. of the 13th international conference on Software engineering*, pages 23–34. IEEE Computer Society Press, 1991.

[GSB+99] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble Component-Based Operating System. In *USENIX Technical Conference*, pages 267–282, June 1999.

[Ham00] D.K. Hammer. Component-Based Architecting for Distributed Real-Time Systems: How to achieve composability? In *Proc. of Int. Symposium on Software Architectures and Component Technology (SACT)*, Jan 2000.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[Har00] M.J. Harrold. Testing: A Roadmap. In *Proc. of the conference on The future of Software engineering*. ACM Press, 2000.

[HC01] D.K. Hammer and M.R.V. Chaudron. Component-Based Software Engineering for Resource-Constraint Systems: What are The Needs? In

In proc. Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01), Jan 2001. position paper.

[HDD+03] John Hatclif, William Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-Based Systems. In *Proc. of the International Conference on Software Engineering*, May 2003.

[HMSW02] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Packaging Predictable Assembly. In *First International IFIP/ACM Working Conference on Component Deployment*. Springer-Verlag LNCS, June 2002.

[HMSW03] S.A. Hissam, G.A. Moreno, J.A. Stafford, and K.C. Wallnau. Packaging Predictable Assembly. *Journal of Systems and Software*, 3, 2003.

[Hus02] Joel Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[IHK+01] J. Davis II, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Tsay, B. Vogel, and Y. Xiong. Heterogeneous Concurrent Modeling and Design in Java. Technical Memorandum UCB/ERL M01/12 University of California, Berkeley, March 2001. http://ptolemy.eecs.berkeley.edu/publications.

[IN02] D. Isovic and C. Norström. Components in Real-Time Systems. Technical report, Departement of Computer Science and Engineering, MdH, 2002.

[JPWG] J-M. Jézéquel, N. Plouzeau, T. Weis, and K. Geihs. From Contracts to Aspects in UML Designs. http://www.qccs.org/aodcontracts.pdf.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira L., J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *11^{th} European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[Kra98] R. Kramer. The Java design by Contract Tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, 1998.

[L+98] Lennart Lindh et al. Hardware Accelerator for Single and Multiprocessor Real-Time Operating Systems. In *the Seventh Swedish Workshop on Computer Systems Architecture*, June 1998.

[LA99] K. Lundqvist and L. Asplund. A Formal Model of a Run-Time Kernel for Ravenscar. In *Proc. Sixth International Conference on real-Time Computing Systems and Applications*, pages 504–507, 1999.

[Lee01] E.A. Lee. Computing for Embedded Systems. In *Proc. of the 18th IEEE Instrumentation and Measurement Technology Conference (IMTC)*, volume 3, 2001.

[Lee02] E.A. Lee. *Advances in Computers*, volume 56, chapter Embedded Software. Academic Press, 2002. ISBN 0120121565.

[LKA+95] David C. Luckham, John L. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[LLH02] Frank Lüders, Kung-Kiu Lau, and Shui-Ming Ho. *Building Reliable Component-Based Software Systems*, chapter Specification of Software Components. Artech House, 2002.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UppAal in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.

[LS98] Jun Lang and David B. Stewart. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274–301, March 1998.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[LV95] David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[LX01] E. A. Lee and Y. Xiong. System-Level Types for Component-Based Design. In *Proc. of EMSOFT 2001*, LNCS 2211. Springer-Verlag, October 2001.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[Mey92] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, 1992.

[Mic] Microsoft. .NET Home Page. http://www.microsoft.com/net/.

[Mic96] Microsoft. The Component Object Model Specification, 1996. http://-www.microsoft.com/com/.

[MMS98] B. Meyer, C. Mingins, and H. Schmidt. Trusted Components for the Software Industry. *IEEE Computer*, May 1998.

[Mor] Mark Moriconi. Architecture Refinement.

[MSZ02] P.O. Müller, C.M. Stich, and C. Zeidler. *Building Reliable Component-Based Software Systems*, chapter Component-Based Embedded Systems. Artech House, 2002.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *Software Engineering*, 26(1):70–93, 2000.

[NA00] O. Nierstrasz and F. Achermann. Separation of Concerns through Unification of Concepts. In *14^{th} European Conference on Object-Oriented Programming*, 2000.

[OMGa] OMG. CORBA Home Page. http://www.omg.org/corba/.

[OMGb] OMG. OMG Home Page. www.omg.org.

[OMG99] OMG. OMG Unified Modeling Language Specification, v1.3, June 1999.

[OMG02a] OMG. CORBA Component Model 3.0, June 2002. http://www.omg.-org/technology/documents/formal/components.htm.

[OMG02b] OMG. CORBA/IIOP Specification, v3.0.2, 2002. formal/02-12-06.

[OMG02c] OMG. Minimum CORBA, v1.0, 2002. formal/02-08-01.

[OMG02d] OMG. Real-Time CORBA Specification, v1.1, 2002. formal/02-08-02.

[OMG02e] OMG. UML Profile for Schedulability, Performance and Time Specification, March 2002. OMG document number ptc/02-03-02.

[OSG] OSGI. OSGI Service Gateway Specification, Release 1.0. http://www.-osgi.org.

[PA91] J.M. Purtilo and J.M. Atlee. Module Reuse by Interface Adaptation. *Journal on Software: Practice and Experience*, 21(6):539–556, 1991.

[PCW85] D.L. Parnas, P.C. Clements, and D.M. Weiss. The Modular Structure of Complex Systems. *IEEE Trans. Software Eng.*, 11(3):259–266, March 1985.

[PEC] http://www.sei.cmu.edu/pacc/.

[Pel99] C. Peltz. A Hierarchical Technique for Composing COM based Components. In *Proc. of the 2nd International Workshop on Component-Based Software Engineering (CBSE)*, May 1999.

[PH97] S.L. Pfleeger and L. Hatton. Investigating the Influence of Formal Methods. *IEEE Computing*, 30(2):33–43, Feb 1997.

[RB01] U. Rastofer and F. Bellosa. Component-based software engineering for distributed embedded real-time systems. *IEE Proc. online no 20010536*, 2001.

[Sch94] Werner Schütz. Fundamental Issues in Testing Distributed Real-Time Systems. *Real-Time Systems*, 7(2):129 – 157, September 1994.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[SDZ96] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and Implementations for Architectural Connections. In *3^{rd} International Conference on Configurable Distributed Systems*, 1996.

[SGW94] B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons, 1994.

[Sho02] Mohammed El Shobaki. On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems. In *Proc. of the 8th International Conference on Real-Time Computing Systems and Applications*, 2002.

[SSK92] D.B. Stewart, D.E. Schmitz, and P.K. Khosla. The Chimera II Real-Time Operating System for Advanced Sensor-based Control Applications. *IEEE Trans. Systems, Man, and Cybernetics,*, 22(6):1282–1295, November-December 1992.

[Sta01] John A. Stankovic. VEST — A toolset for constructing and analyzing component based embedded systems. *Lecture Notes in Computer Science*, 2211:390–??, 2001.

[SUN] SUN Microsystems. Introducing Java Beans. http://developer.java.sun.-com/developer/onlineTraining/Beans/Beans1/index.html.

[SVK97] D. Steward, R. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, December 1997.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TH99] Henrik Thane and Hans Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proc. of the 20th Real-Time System Symposium*, pages 360 – 369. IEEE, December 1999.

[Tha96] Henrik Thane. Safe and Reliable Computer Control Systems – Concepts and Methods. Technical Report TRITA-MMK 1996:13, ISSN 1400-1179, ISRN KTH/MMK/R-96/13-SE, Mechatronics Laboratory, KTH, 1996.

[VD+02] Sefan Voget, Michael Ditze, et al. EAST/EEA relevant considerations of CORBA. Technical report, EAST-EEA Internal Report, 2002.

[Ves97] S. Vestal. Support for Real-Time Multi-Processor Avionics. In *Proc. 18^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 11–21, December 1997.

[VM95] J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 1995.

[vO02] Rob van Ommering. *Building Reliable Component-Based Software Systems*, chapter The Koala Component Model, pages 223–236. Artech House Publishers, July 2002. ISBN 1-58053-327-2.

[VP00] J. Voas and J. Payne. Dependability certification of software components. *Journal of Systems and Software*, 52, 2000.

[WB97] A. Wellings and A. Burns. Workshop Report, The Eighth International Real-Time Ada Workshop (IRTAW8). *Ada User Journal*, 18(2), June 1997.

[Win] Wind River Systems Inc. VxWorks Programmer's Guide. http://www.-windriver.com/.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley, 1998.

[WS01] K.C. Wallnau and J.S. Stafford. Ensembles: Abstractions for a New Class of Design Problem. In *Proc. of the IEEE 27th Euromicro Conference (Euromicro 2001)*, Warsaw, Poland, September 2001. IEEE Computer Society Press. http://www.sei.cmu.edu/pacc/WallnauStafford-ECBSE.-pdf.

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.