

Mälardalen University Press Dissertations

No. 35

**AN EVOLUTIONARY APPROACH TO SOFTWARE
COMPONENTS IN EMBEDDED REAL-TIME SYSTEMS**

Frank Lüders

2006



MÄLARDALEN UNIVERSITY

Department of Computer Science and Electronics
Mälardalen University

Copyright © Frank Lüders, 2006

ISSN 1651-4238

ISBN 91-85485-29-2

Printed by Arkitektkopia, Västerås, Sweden

Contents

Abstract	v
Acknowledgements	vii
List of Included Papers	ix
List of Other Papers	xi
Chapter 1 Introduction	1
1.1 Software Components in Embedded Real-Time Systems.....	2
1.2 An Evolutionary Approach	2
1.3 Research Questions.....	4
1.4 Research Methods	6
1.5 Contributions and Organization of the Dissertation	10
1.6 References.....	13
Chapter 2 State of the Art and Related Work	15
2.1 Software Architecture.....	15
2.1.1 Definitions of Software Architecture	15
2.1.2 Architectural Design	17
2.1.3 Architectural Analysis and Evaluation	21
2.1.4 Architectural Description and Documentation	23
2.2 Component-Based Software Engineering.....	28
2.2.1 Definitions of Software Components	28
2.2.2 Software Component Models and Technologies.....	31
2.2.3 Software Component Services	38
2.2.4 Component-Based Software Engineering Practices	42
2.3 Embedded Real-Time Systems	44
2.3.1 Definitions of Embedded Real-Time Systems.....	45
2.3.2 Industrial Control Systems.....	47
2.3.3 Software Components in Embedded Real-Time Systems	54
2.4 References.....	61

Chapter 3	Specification of Software Components.....	67
3.1	Introduction	67
3.2	Current Component Specification Techniques	68
3.3	Specifying the Semantics of Components	72
3.4	Specifying Extra-Functional Properties of Components	79
3.5	Summary	81
3.6	Corrections to the Original Version	82
3.7	References.....	82
Chapter 4	Adopting a Software Component Model in Real-Time Systems Development	85
4.1	Introduction	85
4.2	Motivation.....	86
4.3	Adopting Microsoft Models.....	87
4.3.1	COM Interfaces	87
4.3.2	Instantiation and Dynamic Linking.....	89
4.3.3	Location Transparency with DCOM.....	91
4.3.4	The Next Generation: .NET	93
4.4	Related Work	94
4.5	Conclusion and Future Work	94
4.6	References.....	95
Chapter 5	Adopting a Component-Based Software Architecture for an Industrial Control Systems – A Case Study.....	97
5.1	Introduction	97
5.1.1	Questions Addressed by the Case Study.....	98
5.1.2	Case Study Method	99
5.2	Context of the Case Study	100
5.3	Componentization	105
5.3.1	Reverse Engineering of the Existing Software Architecture.....	105
5.3.2	Component-Based Software Architecture	107
5.3.3	Interaction between Components.....	109

5.4	Experiences	113
5.5	Related Work	116
5.6	Conclusions and Future Work.....	117
5.7	References.....	118
Chapter 6	A Prototype Tool for Software Component Services in Embedded Real-Time Systems.....	121
6.1	Introduction	121
6.2	Component Services.....	123
6.2.1	Logging.....	123
6.2.2	Execution Time Measurement.....	124
6.2.3	Synchronization	125
6.2.4	Execution Timeout.....	126
6.2.5	Vertical Services.....	127
6.3	Prototype Tool.....	127
6.3.1	Design Consideration	128
6.3.2	Supported Services	129
6.3.3	Example Application	130
6.4	Related Work	136
6.5	Conclusion and Future Work	138
6.6	References.....	139
Chapter 7	Use of Software Component Models and Services in Embedded Real-Time Systems.....	143
7.1	Introduction	143
7.2	Background.....	145
7.2.1	The Component Object Model (COM).....	145
7.2.2	Software Component Services for Embedded Real-Time Systems.....	147
7.3	Example Application.....	151
7.4	Tests	157
7.4.1	Test Setup	157
7.4.2	Results.....	158
7.5	Discussion.....	161

7.6	Related Work	162
7.7	Conclusion and Future Work	163
7.8	References.....	164
Chapter 8	Evaluation of a Tool for Supporting Software Component Services in Embedded Real-Time Systems	167
8.1	Introduction	167
8.2	Background	168
8.3	Case Study Design	172
8.4	Data Collection.....	174
8.5	Analysis.....	176
8.6	Related Work	178
8.7	Conclusion and Future Work	179
8.8	References.....	180
Chapter 9	Conclusion	183
9.1	Summary of Results.....	183
9.2	Research Questions Revisited.....	188
9.3	Future Work	193

Abstract

Component-based software engineering denotes the practice of building software from pre-existing smaller products, in particular when this is done using standardized software component models. The main expected benefits of this practice over traditional software engineering approaches are increased productivity and timeliness of development projects. While the use of software component models has become common for desktop and server-side software, this is not the case in the domain of embedded real-time systems, presumably due to the special requirements such systems have to meet with respect to predictable timing and limited use of resources. Much research exists that aims to define new component models for this domain, typically focusing on source code components, static system configuration, and relatively narrow application domains.

This dissertation explores the alternative approach of using components based on binary code, allowing dynamic configuration, and targeting a broader domain. A study of a general purpose component model shows that the model is compatible with real-time requirements, although putting some restrictions on its use may be necessary to ensure predictability. A case study demonstrates how the model has been beneficially used in an industrial control system. The dissertation furthermore proposes an approach for extending the component model with run-time services for embedded real-time systems. It presents a prototype tool for supporting such services, along with two empirical studies to evaluate the approach and the tool as well as the component model itself. One study shows that both the component model and the services provided by the tool result in very modest and predictable run-time overheads. The other study, evaluating the effects on productivity and quality of using the approach in a software development project, did not produce quantitative evidence, but concludes that the approach is promising and identifies possible adjustments to it and opportunities for further studies.

Acknowledgements

I would like to thank my supervisor Ivica Crnkovic for all his help and support during my work with this dissertation. The first half of this work was conducted within the project *Standard Technologies in Industrial Applications*, run jointly by Mälardalen University's Department of Computer Engineering and ABB Automation Products. It was funded by the company and the Swedish Knowledge Foundation. I was at that time employed part-time by ABB and I am grateful to Ivica and Erik Gyllenswärd (formerly of ABB) for hiring me and giving me the opportunity to pursue my research interests. I greatly appreciate the helpful cooperation of former colleagues at ABB in Malmö and Västerås, and I would particularly like to thank Staffan Andersson for his valuable input.

The second half of the research was conducted within the *Industrial Software Engineering* project, funded by ABB Sweden and the Swedish Knowledge Foundation, at Mälardalen University's Department of Computer Science and Electronics. I am grateful to the many students that have contributed to the research. A prototype software tool was initially developed by participants of the course on Software Engineering in 2005 and evaluated by the help of participants of the same course in 2006. The tool was developed further and evaluated by several students as part of their Master thesis projects. I want to thank everybody at the department for making it such a pleasant and inspiring place to work. Special thanks to my current and former colleagues in the Industrial Software Engineering group. Thanks also to my assistant supervisors Per Runeson and Björn Lisper for their help and to Per for fruitful collaboration on several papers.

Finally, I wish to thank my family and friends who have been there for me over these years and ask their forgiveness for the times I have been "too busy" to be there for them. Most of all, I am grateful beyond words to Elise for her invaluable help and support and for the brightness she brings to my life.

Frank Lüders
Västerås, November 2006

List of Included Papers

- F. Lüders, K.-K. Lau, and S.-M. Ho, "Specification of Software Components." In I. Crnkovic and M. Larsson (editors), *Building Reliable Component-Based Software Systems*, Artech House Books, 2000.
- F. Lüders, "Adopting a Software Component Model in Real-Time Systems Development." In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop*, 2004.
- F. Lüders, I. Crnkovic, and P. Runeson, "Adopting a Component-Based Software Architecture for an Industrial Control System - A Case Study." In C. Atkinson, C. Bunse, H. Gross, and C. Peper (editors), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- F. Lüders, D. Flemström, A. Wall, and I. Crnkovic, "A Prototype Tool for Software Component Services in Embedded Real-Time Systems." In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, 2006.
- F. Lüders, S. Ahmad, F. Khizer, and G. Singh-Dhillon, "Use of Software Component Models and Services in Embedded Real-Time Systems." In *Proceedings of the 40th Hawaii International Conference on System Sciences*, 2007.
- F. Lüders, I. Crnkovic, and P. Runeson, "Evaluation of a Tool for Supporting Software Component Services in Embedded Real-Time Systems." In *Proceedings of the 6th Conference on Software Engineering Research and Practice in Sweden*, 2006.

List of Other Papers

- I. Crnkovic, M. Larsson, and F. Lüders, "State of the Practice: Component-based Software Engineering Course." In *Proceedings of the 3rd International Workshop on Component-Based Software Engineering*, 2000.
- I. Crnkovic, M. Larsson, and F. Lüders, "The Different Aspects of Component Based Software Engineering." In *Proceedings of the Microprocessor Systems, Process Control and Information Systems Conference*, 2000.
- I. Crnkovic, M. Larsson, and F. Lüders, "Software Process Measurements using Software Configuration Management." In *Proceedings of the 11th European Software Control and Metrics Conference*, 2000.
- I. Crnkovic, M. Larsson, and F. Lüders, "Implementation of a Software Engineering Course for Computer Science Students." In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, 2000.
- F. Lüders and I. Crnkovic, "Experiences with Component-Based Software Development in Industrial Control." In *Proceedings of the 1st Swedish Conference on Software Engineering Research and Practice*, 2001.
- F. Lüders, I. Crnkovic, and A. Sjögren, "A Component-Based Software Architecture for Industrial Control." In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, 2002.
- F. Lüders, I. Crnkovic, and A. Sjögren, "Case Study: Componentization of an Industrial Control System." In *Proceedings of the 26th Annual Computer Software and Application Conference*, 2002.
- F. Lüders, *Use of Component-Based Software Architectures in Industrial Control Systems*. Technology Licentiate Thesis, Mälardalen University, 2003.
- F. Lüders, D. Flemström, and A. Wall, "Software Component Services for Embedded Real-Time Systems." In *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*, 2005.
- F. Lüders, D. Flemström, and A. Wall, "Software Component Services for Embedded Real-Time Systems." In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, 2005.

Chapter 1

Introduction

Component-based software engineering denotes the disciplined practice of building software from pre-existing smaller products, generally called *software components*, in particular when this is done using standard or de-facto standard *component models* [1, 2]. A component model generally defines a concept of components and rules for their design-time composition and/or run-time interaction, and is usually accompanied by one or more component technologies, implementing support for composition and/or interoperation. The main expected benefits of component-based software engineering are increased productivity and timeliness of software development projects.

The use of software component models has become increasingly popular during the last decade, in particular in the development of software for desktop applications and distributed information systems. Popular component models include *JavaBeans* [3] and *ActiveX* [4] for desktop applications and *Enterprise Java Beans* (EJB) [5] and *COM+* [6] for distributed information systems. In addition to basic standards for naming, interfacing, binding, and other concepts related to composition and interoperation, these models also define standardized sets of run-time services oriented towards the application domains they target. This concept is termed *software component services* [2].

The remainder of this chapter first addresses the use of software components and component models in embedded real-time systems in Section 1.1 and presents an evolutionary approach to this challenge in Section 1.2. Next, a number of research questions are formulated in Section 1.3, followed by a discussion of the research methods used to address these questions in Section 1.4. Finally, Section 1.5 discusses the research contributions of this dissertation and presents an outline of the rest of the dissertation, including the published papers and their individual contributions. Particular care is taken to distinguish my own contributions to each paper from those of the coauthors.

1.1 Software Components in Embedded Real-Time Systems

Unlike for desktop applications and distributed information systems, there has been no widespread use of software component models in the domain of embedded real-time systems. It is generally assumed that this is due to the special requirements such systems have to meet, in particular with respect to timing predictability and limited use of resources such as memory and CPU time. Much research has therefore been directed towards defining new component models for real-time and embedded systems, typically focusing on relatively small and statically configured systems. Most of the published research proposes models based on source code components. Typically, these models target relatively narrow application domains. Examples of such models include the *Koala* component model for consumer electronics [7], *PECOS* for industrial field devices [8], and *SaveCCM* for vehicle control systems [9].

The focus on statically configured systems of source code components is motivated by efficiency as well as by the possibility of ensuring predictable behavior through source code analysis and white-box testing. A potential liability of using source code components is that application developers rely on component properties that may be inferred from the source code but are not guaranteed by component specifications. Thus, a system may break if a component is updated with a new version that does not have the same inferred properties, although the component specifications are compatible. Other possible problems related to source code components are exposure of intellectual property and complication of deployment as such components must be compiled and linked with the rest of the system. The restriction to static configuration is increasingly at odds with requirements for flexibility, adaptiveness, etc. The development of component models for relatively narrow application domains is motivated by the desire to optimize systems for attributes considered particularly important for those domains. Typically, such narrow models, as well as supporting tools and run-time infrastructures, have to be developed by the application developing organizations themselves.

1.2 An Evolutionary Approach

An alternative approach is to strive for a component model for embedded real-time systems based on binary components and targeting a broader domain of applications, similarly to the domain targeted by a typical real-time operating system. Support for such a model could suitably be provided by platform

vendors, as is the norm for component models used for desktop applications and information systems. Although any model based on binary components is likely to incur some overhead, efficient use of resources should be a primary concern in the design of such a model for embedded real-time systems. When it comes to ensuring predictable behavior, the vision of this approach is that analysis of systems should be based on specifications (i.e. models) of the included components rather than relying on access to source code. Realizing this vision requires methods for ensuring that components comply with specifications as well as for predicting the properties of a system based on properties of its constituent components [10]. Further investigation of such methods is outside the scope of this dissertation, however.

The possibility explored in this dissertation is to use a mainstream component model as the starting point for developing a component model for embedded real-time system. Benefits of adopting an existing component model include that it may be possible to use existing development environments, existing components can be re-used or adapted for the real-time domain, and integration with application from other domains becomes significantly simpler. As a concrete example of a component model, Microsoft's *Component Object Model* (COM) [11] has been selected. Some reasons that COM is an attractive starting point are that the model is relatively simple, commercial COM implementations are already available for a few real-time operating systems, and the model is well-known and accepted in industry. While COM is increasingly being replaced by the newer .NET technologies [12] in the desktop and information systems domains, .NET is not easily adapted to the domain of embedded real-time systems. In particular, the loss of predictability resulting from automatic memory management (i.e. garbage collection) is a serious barrier.

The assumption that COM is a suitable starting point for the effort outlined above was strengthened by positive results from the earlier phases of the work presented in this dissertation. A study of COM and its extension *Distributed COM* (DCOM) [13] has shown that these models are not inherently incompatible with real-time requirements, although some restrictions on how the models are used may be necessary to ensure predictability, and an industrial case study has demonstrated that the key concepts of COM can be beneficially used in the development of an embedded real-time system. The latter study furthermore demonstrates the possibility of adopting COM for parts of a system without requiring the rest of the system to be changed, thus allowing a gradual adoption of the model.

Putting restrictions on the use of the component model to improve predictability is not enough to make it an attractive option for industrial-scale applications. In addition, there must be efficient support for creating software (i.e. building new components and applications) and reusing software (i.e. reusing components across applications). The approach suggested is a combination of restrictions and extensions of the existing component model to adopt it to the target domain. The goal is to lay the groundwork for a software component model for embedded real-time systems, using the basic concepts of COM as the starting point and extending this basic model with standardized services of general use for this application domain, much like COM+ extends COM with services for distributed information systems. This is termed an *evolutionary* approach, partly because it is based on adaptation of an existing model and partly because it is intended to support a gradual evolution of existing monolithic systems into component-based systems. In contrast, approaches proposing new domain-specific component models may be termed *revolutionary*. Furthermore, such models often require all the software in a system to be in the form of compliant components, thereby hindering an evolutionary adoption of the model in existing systems.

1.3 Research Questions

The overall question addressed by this dissertation is whether a software component model can be beneficially used in the development of software for embedded real-time systems; more specifically, whether a model based on binary components can be beneficially used and whether an extension of such a model with support for run-time services of general use for the application domain can bring additional benefits. By “beneficially” is meant that using the model results in savings in software development effort while not having unacceptable effects on important quality attributes of the developed software. The most obvious of these quality attributes is the software’s ability to exhibit the predictable timing and use of resources required for embedded real-time systems.

This overall question can be decomposed into more detailed questions. The first to be addressed in this dissertation is formulated as follows:

Research Question 1

What are the advantages and liabilities of using a software component model based on binary components in the development of embedded real-time systems?

More specifically, the use of COM and DCOM is investigated in Chapter 4 of this dissertation. In addition to the question of whether it is possible to use these models for systems with real-time requirements at all, the question of how they should be used to ensure real-time predictability is addressed. This leads to the following two sub-questions:

Research Question 1-1

Is it possible to use COM/DCOM in the development of software for systems with real-time constraints?

Research Question 1-2

What restrictions (if any) should be placed on the use of COM/DCOM in software for systems with real-time constraints to ensure predictability?

The next question addresses the use of COM in a concrete system where a part of the system's software architecture is redesigned to allow functionality to be implemented in independently developed components:

Research Question 2

What are the effects of adopting a component-based software architecture for an embedded real-time system?

This question has been addressed by an industrial case study, described in Chapter 5. Based on the challenges of the studied project, the following two sub-questions were formulated:

Research Question 2-1

What are the effects on the effort required to make extension to the system?

Research Question 2-2

What are the effects on the real-time predictability of the system?

Since the aim of the project was to make it easier to make extensions to the system and adopting the new software architecture required a development effort in itself, it is interesting to compare this effort to the reduction in efforts required for extensions to determine if, and after how many extensions, the effort invested in adopting the new architecture is regained.

Another question is related to the extension of a basic component model with automatically generated support for run-time services of general use for embedded real-time systems:

Research Question 3

What are the effects of using automatically generated support for software component services in the development of an embedded real-time system?

This question is addressed by two empirical studies using a prototype tool for proxy-based software component services, introduced in Chapter 6. The first of these studies, described in Chapter 7, addresses the following sub-question:

Research Question 3-1

What are the effects on the software's size, resource usage, and predictability?

The second study, described in Chapter 8 of this dissertation, addresses the following two sub-questions:

Research Question 3-2

What are the effects on the quality of the produced software?

Research Question 3-3

What are the effects on the software development effort?

1.4 Research Methods

This dissertation, like most software engineering research, belongs to the domain of *empirical research*. As such, it differs from much computer science research, which is mathematical or logical in nature and strive to present formal proofs. In their treatment of software metrics, Fenton and Pfleeger [14] discuss empirical investigation in software engineering. Although they focus on investigations in software developing organizations as a tool for making scientific and objective assessments or decisions, the applicability to research is also stated. *Formal experiments*, *case studies*, and *surveys* are identified as three different ways of conducting empirical investigations.

Formal experiments are used to investigate causal relationships in controlled settings. An example might be the effect of two different programming languages on productivity. An experiment would vary the language and measure the productivity in the development of two equivalent pieces of software. It would furthermore be necessary to control that other parameters, such as programmer skill and motivation, that may affect the productivity is kept constant. In addition, formal experiments are, by definition, replicable. Due to these requirements on control and replicability, experimentation is most suitable

bly performed with fairly limited activities. In fact, most formal experiments reported in the software engineering literature have been performed in academic settings with students as subjects. Thus, the validity of their results to industrial scale software development is often questioned, although some such experiments are accompanied by arguments for wider validity [15, 16].

In settings such as industrial projects, where the researcher does not have the level of control required for formal experiments, case studies or surveys can be used. A survey is retrospective in nature and samples the results of activities after they are completed. This is often performed on a large set of information, for instance obtained from a set of projects from one or more organizations. A case study is usually not retrospective, and the researcher will decide in advance what to study and plan how to capture the necessary data. A typical software engineering case study follows a development project, using direct observation as an important source of data. The projects selected for such studies are often those that are believed to be typical for an organization or an application area. Thus, there is a difference in scale between the different techniques where formal experiments can be viewed as *research in the small*, case studies as *research in the typical*, and surveys as *research in the large*. Based on the description by Fenton and Pfleeger [14], Table 1-1 summarizes some of the aspects in which the three forms of empirical investigation differ.

Table 1-1 Differences between three empirical investigation techniques

Aspect	Experiments	Case studies	Surveys
Level of control	High	Low	Low
Replicable?	Yes	No	No
Retrospective?	No	Usually not	Yes
Scale	Small	Typical	Large

The goal of the research described in this dissertation has been to use empirical methods to answer the research questions presented in the previous section. Research Question 1 has been addressed by studying the specifications of the component models in question. Although this is not an empirical investigation in itself, the results of the study have been instrumental when planning the subsequent empirical studies.

Research Question 2 has been addressed by a case study conducted in an industrial setting. This technique is discussed in more detail by Robson [17], who provide the following definition:

Case study is a strategy for doing research which involves an empirical investigation of a particular contemporary phenomenon within its real life context using multiple sources of evidence.

Thus, rather than a single method, a case study represent a strategy that can include several methods, such as observation and interviews. In this particular study, the investigated phenomenon was the use of a component-based software architecture and the context an industrial development project. This is a typical example in that the phenomenon is not easily separated from the context. The sources of evidence have included direct observation through project participation, interviews with project members, documentation, and software artifacts. Clearly, this kind of strategy cannot be expected to lead to a definitive answer to the research question supported by anything like a formal proof. Instead, an overall analysis of the collected data can be expected to provide evidence in support of one or more possible answers to the question.

More specifically, the employed strategy can be called a participatory case study, since I have been an active member of the project under investigation. This is similar to what Robson calls *action research* [17]. An advantage of such a participatory study is that the researcher has opportunities to make observations that yield information that might be hard to obtain in other ways. There is also a risk, however, that the researcher may loose the required distance and objectivity. A possible way to mitigate this risk is to analyze and report the study in cooperation with other researchers that can contribute with an outsider's view. This approach was taken in the preparation of this dissertation.

Research Question 3 has been addressed by two different empirical studies, which may also be viewed as case studies. The first of these consisted of implementing an application both with and without using the approaches under investigation. The study is similar to an experiment in some ways, as the application development is repeated while varying some parameter and the only source of evidence is measurement of static and dynamic aspects of the developed applications. The study could have been turned into a formal experiment by implementing a sufficiently high number of different real-time applications, which could then have been viewed as a sample of all possible applications within the domain. Since this would have required more effort than was possible, a case study strategy was adopted by selecting an application be-

lieved to be typical for the domain. Thus, the study provides a direct answer to the research question for the particular application and, more importantly, provides evidence to support a hypothesis for the application domain.

In the second study addressing this research question, four teams of students were given the same development task. Two of the teams were instructed to use the approach under investigation. Thus, this study is an example of a *multiple-case* study, which Yin argues is preferable to the classical single-case study [18]. The reason that this study cannot be viewed as a formal experiment is that the number of teams (two using the approach and two not using the approach) is too small to rule out that any observed differences between the teams are caused by spurious effects, i.e. other factors than whether the approach is used or not. Thus, a more elaborate analysis of the teams' performance rather than merely observation of dependent variables is employed to investigate the causal relationships between the use of the approach and the project outcomes. The sources of evidence were documentation (including reported working hours), software artifacts, and observation of and communication with project members.

The phenomenon investigated in the studies described in the preceding two paragraphs is the use of proxy-based software component services in embedded real-time systems. The studies are viewed as case studies, although it may be argued that neither of the two studies the phenomenon in its real-life context. In both cases, software is developed by students (as part of a term project and Master thesis project, respectively), which may be considered a form of laboratory environment. Different strategies for varying parameters were used in the studies. In the first study, the same group of students developed an application both with and without using the approach while, in the second study, half of the teams were instructed to use the approach and the other half were not. In more realistic contexts, such as industrial projects, repeating the development effort is usually prohibited for cost reasons. Thus, the studies may also be viewed as hybrids between case studies and quasi-experiments, i.e. a design similar to experiments where the researcher lacks the proper control over parameters [19].

Another commonality of the two studies addressing the last research question is that they involved the use a prototype software tool that has been developed in the course of the research described in this dissertation. The term *constructive research* is sometimes used to describe research that involves building an artifact to solve a domain problem [20]. While such an artifact is not a scientific

result in itself, knowledge obtained by using the artifact may be. In this work, the constructive research strategy has been employed by first implementing the prototype tool and then conducting two empirical studies where the tool is used by students. The domain problem in this case is to make software components an attractive alternative in the development of embedded real-time systems.

1.5 Contributions and Organization of the Dissertation

The research described in this dissertation uses empirical methods to investigate the use of a particular type of software component model in the development of embedded real-time systems. Thus, its primary contribution is increased knowledge of the advantages and liabilities of using this type of component model in this application domain. Most other research on component-based development for embedded real-time system focuses on rather different component models, as described in the introduction. In addition, the collection of empirical evidence on the effects of using component-based development is a contribution in itself. While it is generally assumed that the component-based paradigm leads to benefits related to both productivity and quality there is a shortage of empirical evidence for this. In addition to these epistemic contributions, there are more practical contributions, in the form of a proposed approach to software component services for embedded real-time systems and a prototype tool that demonstrates how automatic code generation can support such services.

The dissertation is a collection of published papers, with some additional introductory and concluding chapters. Chapter 2 describes the state of the art within component-based software engineering in addition to selected topics within software architecture and embedded real-time systems. The treatment of software architecture covers definitions of software architecture, architectural design/styles, analysis/evaluation of software architectures, and architectural description/documentation. Within component-based software engineering, definitions of software components, software component models/technologies, component-based software engineering practices, and software component services are described. The section on embedded real-time systems covers definitions of embedded real-time systems, industrial control systems, and software components in embedded real-time systems. The description of industrial control systems provides useful background information for the case study presented in Chapter 5 and the example applications used in Chap-

ters 6–8, while the discussion of software components in embedded real-time system is the dissertation’s main treatment of related work, along with the included papers’ more specific discussions of related work.

Chapter 3, coauthored with Kung-Kiu Lau and Shui-Ming Ho, was originally published in the book *Building Reliable Component-Based Software Systems* (Artech House Books, 2000). The chapter discusses the state of the practice and research of software component specification. Thus, it is an extension of the dissertation’s coverage of the state of the art. In addition, it contains a contribution in the form of UML metamodels of the concepts involved in software component specification. The bulk of the paper is the description of three levels of software component specification, which are denoted syntactic, semantic and extra-functional specification. This includes a description of interface and component specifications in COM, some knowledge of which is assumed in subsequent chapters. Most of this work, including the UML metamodeling, is my individual contribution. The co-authors contributed mainly to the introduction and summary of the paper and to the description of realization specifications at the end of Section 3.3. This version of the paper contains some corrections to the original version, which are described in Section 3.6.

Chapter 4 was originally published in *Proceedings of the 28th Annual NASA/IEEE Software Engineering Workshop* (IEEE Computer Society Press, 2004). The paper presents a motivation for applying component-based software engineering to real-time systems and discusses the consequences of adopting a software component model in the development of such systems. Specifically, the consequences of adopting Microsoft’s COM, DCOM, and .NET models are analyzed. The most important aspects of these models are discussed in an incremental fashion. The analysis considers both real-time systems in general and the industrial control system described in more detail in Chapter 5, where some aspects the COM model have been adopted. The study concludes that COM and DCOM are not inherently incompatible with real-time requirements, but suggests restrictions on the use of the models to improve predictability. The paper is my individual contribution.

Chapter 5, coauthored with Ivica Crnkovic and Per Runeson, was originally published in the book *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends* (Springer, 2005). It describes an industrial case study demonstrating that a component-based software architecture can be beneficially used in the development of an embedded real-time system. The investigated case is an example of an evolutionary compo-

nentization of an existing system. The description of the project, the system, and its architectural changes is my contribution. The analysis of the experiences was initiated by me and refined in collaboration with the coauthors, who provided the desired outsider's views.

Chapter 6, coauthored with Daniel Flemström, Anders Wall, and Ivica Crnkovic, was originally published in *Proceedings of the 9th International Symposium on Component-Based Software Engineering* (Springer, 2006). The paper suggests a proxy-based approach for software component services in embedded real-time systems and describes a prototype tool for COM and Windows CE along with an empirical evaluation of it by a control system example, implemented on the Windows CE emulator. The empirical studies described in the following two chapters build on this work. The underlying principle of proxy-based services for embedded real-time systems was originally my idea and was refined through discussions with Daniel Flemström. The prototype was initially developed by students under his and my supervision and subsequently extended by me. The other coauthors helped with critical reviews and contributed to the description of related work.

Chapter 7, coauthored with Shoaib Ahmad, Faisal Khizer, and Gurjodh Singh-Dhillon, has been accepted for publication in *Proceedings of the 40th Hawaii International Conference on System Sciences* (IEEE Computer Society Press, 2007). The paper describes empirical evaluations of the run-time effects of using COM and proxy-based software component services on Windows CE. This is based on measurements using applications that have been developed using these models as well as reference applications implementing the same functionality without using the models. These measurements show that the overheads associated with both COM and proxy-based services are modest and quite predictable. The conception, planning, and design of the study are my individual contribution. The coauthors contributed by doing some of the software implementation, performing the measurements, and documenting the results. This was conducted as part of their Master thesis project, which I supervised.

Chapter 8, coauthored with Ivica Crnkovic and Per Runeson, was published in *Proceedings of the 6th Conference on Software Engineering Research and Practice in Sweden* (Umeå University, 2006). The paper describes an empirical study of the development-time effects of using proxy-based software component services. This was achieved by giving four teams of students the same development task and instructing two of the teams to use the prototype tool introduced in

Chapter 6. While the study did not show any significant relationships between the use of the tool and the performance of the teams, it was helpful in identifying possible modifications to the tool that would have improved the quality of the developed software. Several options for further investigation were also identified. This paper is mainly my individual contribution. The coauthors helped by taking part in discussions during the study and contributed to the discussion of the research design and methodology in Section 8.3.

Chapter 9 concludes the dissertation by summarizing the results and conclusions of the included papers. These are furthermore augmented by more recent results, obtained through input from the company where the industrial case study was conducted, additional analysis of data collected in the empirical studies with student participation, and some supplementary tests with the prototype tool. Based on this information, answers to the research questions are formulated and the validity of these answers discussed. Finally, different opportunities for future work are discussed, with particular focus on further empirical studies to test and possibly strengthen the conclusions of those already conducted or to address remaining questions. In addition, other research challenges related to the use of software component models and services are identified, including its possible impact on specification and compositional reasoning, and the possibilities of commercializing the research results are briefly discussed.

1.6 References

- [1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edition. Addison-Wesley, 2002.
- [2] G. T. Heineman and W. T. Councill (editors), *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [3] R. Englander, *Developing Java Beans*. O'Reilly, 1997.
- [4] D. Chappell, *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [5] B. Burke and R. Monson-Haefel, *Enterprise JavaBeans 3.0*, 5th edition. O'Reilly, 2006.
- [6] D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.

- [7] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [8] T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, and P. Müller, "Components for Embedded Software – The PECOS Approach." In *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.
- [9] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren, "SaveCCM – A Component Model for Safety-Critical Real-Time Systems." In *Proceedings of the 30th EROMICRO Conference*, 2004.
- [10] S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau, "Enabling Predictable Assembly." In *Journal of Systems and Software*, volume 65, issue 3, 2003.
- [11] D. Box, *Essential COM*. Addison-Wesley, 1997.
- [12] D. S. Platt, *Introducing Microsoft .NET*, 3rd edition. Microsoft Press, 2003.
- [13] F. E. Redmond III, *DCOM: Microsoft Distributed Component Object Model*. Wiley, 1997.
- [14] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd edition. PWS, 1997.
- [15] P. Runeson, "Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Data." In *Proceedings of the 7th International Conference on Empirical Assessment & Evaluation in Software Engineering*, 2003.
- [16] M. Höst, B. Regnell, and C. Wohlin, "Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment." In *Empirical Software Engineering*, volume 5, issue 3, 2000.
- [17] C. Robson, *Real World Research: A Resource for Social Scientists and Practitioner-Researchers*, 2nd edition. Blackwell, 2002.
- [18] R. K. Yin, *Case Study Research: Design and Methods*, 3rd edition. Sage, 2002.
- [19] C. Wohlin, P. Runeson, M Höst, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*. Kluwer, 1999.
- [20] K. Lukka, "The Constructive Research Approach." In L. Ojala and O.-P. Hilomola (editors), *Case Study Research in Logistics*. Publications of the Turku School of Economics and Business Administration, 2003.

Chapter 2

State of the Art and Related Work

2.1 Software Architecture

The structure and organization of software systems have been discussed, to a certain degree, since the late 1960s. Well-known examples from the early literature on this topic include influential papers by Dijkstra [1] and Parnas [2]. The last decade, however, has seen an unprecedented interest in this area, both within the research community and among software practitioners. In one of the first papers in the recent wave of software architecture literature [3], Perry and Wolf claim that software design, while receiving much attention in the 1970s, was largely overlooked during the 1980s. The authors use the term *software architecture* instead of design to evoke notions of a professional discipline and to make analogies with other fields, such as building and computer architecture.

2.1.1 Definitions of Software Architecture

The term software architecture denotes both a discipline – that of software architects – and a type of artifact – the architecture of a software system. The recent interest in the field has resulted in an abundance of definitions of software architecture in the latter sense of the term. This section presents and discusses some of the most influential of these definitions.

The above-mentioned paper by Perry and Wolf presents the following model:

$$\text{Software Architecture} = \{\text{Elements, Form, Rationale}\}.$$

The *elements* of an architecture can be processing elements, data elements, or connecting elements (which may themselves be processing elements or data

elements or both). The *form* specifies constraints on elements and their interaction with each other. The *rationale* provides motivations on the choice of elements and the form. Although nobody seems to question the value of documenting the rationale for a software architecture, more recent definitions tend to view rationale as not being part of the architecture itself.

In the first book on the topic [4], Shaw and Garlan define the software architecture of a system as:

a collection of computational components – or simply *components* – together with a description of the interactions among these components – the *connectors*.

This definition is inspired by the way practitioners tend to represent software architectures informally in the form of box and line diagrams. For such diagrams to be useful for others than their creators, it is important that the meanings of both the boxes (components) and the lines (connectors) are described.

The terminology of Shaw and Garlan's definition has become widely adopted within the field. It has also been somewhat criticized, however, for instance in a book by staff members from the Software Engineering Institute (SEI) [5]. The authors argue that the term connector is unfortunate since it indicates a run-time mechanism, while software architecture also covers structures that are not observable at run-time. In the second edition of the book, the term component is also avoided since it has become so closely associated with the topic of component-based software engineering, where components are usually viewed as run-time entities. The latest edition of the SEI book uses the following working definition:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

This definition has some interesting aspects. The notion that a system may have multiple structures is closely related to the concept of architectural views, which is now widely accepted in the research community. Views are further discussed in this chapter in connection with architecture description and documentation. The definition furthermore states that an architecture includes the externally visible properties of components, implying that other component properties are not part of the architecture.

Finally, a recommended practice for architectural documentation from the Institute of Electrical and Electronics Engineers (IEEE) [6] defines architecture as:

the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

The main novelty of this definition is its mention of the system's environment. This is also an example of a process-oriented definition that includes design and evolution principles. As is the case with rationale, the majority of the literature seems to consider such principles to be important but distinct from the architecture itself.

2.1.2 Architectural Design

It was described earlier how Perry and Wolf selected to use the term software architecture instead of the more traditional term software design. The question still arises, however, as to the precise relationship between architecture and design. A common view is expressed in [7]:

Architecture is design, but not all design is architecture.

In other words, a system's software architecture comprises some, but not all, the decisions made in the design of the system. The definitions presented in the previous section do, to varying degrees, specify which types of design decisions an architecture should include. It can generally be said that software architecture is concerned with high-level design decisions that are made at an early stage of the design process. The term *architectural design* is often used for the design activities of this early stage. In this thesis, the term *architectural decision* will furthermore be used to denote design decisions made during this stage, and a software architecture will at times be viewed as a set of architectural decisions.

Shaw and Garlan characterize architectural design as being concerned with structural issues, such as:

global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives.

The SEI book presents guidelines for making architectural decisions that help to ensure a system's quality properties. Decisions that target particular properties are called *architectural tactics*. For example, fault-tolerance is an avail-

ability tactic and information hiding is a modifiability tactic. A set of related tactics is called an *architectural strategy*. Bosch [8] suggests a method of architectural design where an initial architecture is designed based on the system's functional requirements. The architecture is then evaluated against the extra-functional requirements for the systems and transformed if necessary. This process of evaluation and transformation is applied iteratively until the architecture is believed to meet all functional and extra-functional requirements. Evaluation of software architectures is discussed later in this chapter. An approach developed by Siemens Corporate Research [9] focuses on identifying factors that influence architectural issues, which are classified into technical, organizational, and product factors. Based on analyses of these factors, strategies are determined to resolve the issues. The early design of a system's architecture is also a central concept in the Rational Unified Process (RUP) [10]. In this influential process model, a stable architecture is the main milestone of the elaboration phase, which precedes the labor-intensive construction phase.

In all engineering disciplines, successful solutions to past problems are often used as models when new problems are to be solved. This is also true for software architecture, where architects have primarily drawn on their own experiences or that of their development organization. The research community has realized the benefit of having a collection of well-documented prototype solutions. The term *architectural style* was introduced in Perry and Wolf's paper to denote such a prototype solution.

This term is also used by Shaw and Garlan in their book. Drawing on their definition of software architecture, they present the following definition:

An architectural style defines a *vocabulary* of component and connector types, and a set of *constraints* on how they may be combined. There may also exist one or more *semantic models* that specifies how to determine a system's overall properties from the properties of its parts.

The use of the word vocabulary emphasizes that styles are intended for communicating software architecture solutions. The book also identifies a number of commonly occurring styles. Some of these are briefly discussed below.

- *Pipes and filters*. The components in this style are called filters and each have a set of inputs and a set of outputs. The outputs of a filter can be attached to inputs of other filters via simple connectors called pipes. Typically, the filters transform streams of input data to streams of output data in an incremental fashion. An important constraint is that filters

should be independent in the sense that they do not share state and each filter is unaware of the identities of the other filters it is connected to.

- *Object-oriented systems.* In this style, the components are objects that encapsulate abstract data types and their associated operations. An object can be “connected” to other objects by holding references to them and invoke their operations. Typically, the sets of components and connectors are dynamic, since objects can create and delete other objects and object references can be passed as parameters to operations. Although this style is sometimes considered relatively recent, it is rooted in object-oriented programming, which was first developed by Dahl and Nygaard in the 1960s [11].
- *Event-based systems.* The components in this style have interfaces that provide both operations and events. A component’s operations may be invoked directly by other components as in object-oriented systems. In addition, a component may register an interest in an event that another component provides by associating one of its own operations with it. When the second component subsequently announces the event, the registered operation is invoked, along with any operations that other components have registered. Thus, there are two distinct types of connectors in this style.
- *Layered systems.* The components in this style are called layers and are commonly thought of as being stacked on top of each other. Each layer provides services to the layer above it and is a client of the layer below it. The connectors are defined by the protocols used between the layers. A variation of the style is systems where a layer may use the services provided by all lower layers.
- *Repositories.* In this style there are two distinct types of components: a central data store that represents the state of the system and a set of independent components that operate on the data store. An interesting sub-style is systems where computation is entirely controlled by the state of the data store and the independent components react to changes to this state in an opportunistic fashion.

A valuable property of these and other common styles is that the consequences of using them as the basis for a system’s software architecture are fairly well understood. The pipes and filters style, for instance, results in sys-

tems of highly independent components, where filters can suitably be developed and tested separately and possibly reused in different configurations. A possible disadvantage is that all filters have to comply with the data format required by the pipes, which may not be optimally suited for their computation and result in loss of performance and increased internal complexity. An advantage of object-oriented systems is that algorithms and data representation are encapsulated and can be maintained locally. On the other hand, system wide modifications, such as adding new objects, can be difficult since objects need to know the identity of other objects in order to invoke their operations. Event-based systems represent a possible solution to this problem, although the components are not as independent as in the pipes and filters style.

A common occurrence in practice is systems that incorporate several architectural styles. For instance, a system may have components and connectors that match the types defined by several styles. An example is a layered event-based system where each layer provides both operations and events to the layer(s) above it. Another way to combine styles is to mix different components and connectors in the same system, which is sometimes called heterogeneous architectures. For instance, a part of a system could be organized as a repository where one or more of the independent components exchange data with another part of the system that consists of pipes and filters. Hierarchical heterogeneity occurs when a component in a system of one style is internally organized using another style. A typical example is a layer, internally structured using the object-oriented style, which may even be reflected in the layer's services.

An influential direction within the software engineering community in the last decade is the widespread interest in *object-oriented design patterns* [12]. Since architecture is commonly viewed as a special case of design, it is not surprising that the patterns paradigm has also been applied to architectural design. The most comprehensive work in this area has been performed by staff at the German company Siemens, who call their approach *pattern-oriented software architecture* [13]. Like the original work on design patterns, this effort focuses on cataloging known *solutions* to known *problems* in given *contexts*. This approach is similar to that of identifying and documenting architectural styles, and there is now a widespread view that patterns and styles are synonymous.

2.1.3 Architectural Analysis and Evaluation

As previously noted, software architecture is concerned with early design decisions. Clearly, it is important to be able to reason about the effects these decisions will have on the properties of the finished system. The research community has developed a number of architecture analysis and evaluation techniques.

One of the most popular techniques is the *architecture trade-off analysis method* (ATAM) [14] developed by the Software Engineering Institute. The aim of this method is to balance the different quality goals of a system under development, which is very often conflicting. For instance, an architectural decision that results in a very maintainable system may result in sub-optimal performance. ATAM is typical in that it is based on the use of scenarios to analyze how well candidate architectures meet a system's quality goals. Depending on what qualities are being analyzed, scenarios may be operational or related to the system's development or evolution, while the evaluation of their effect may be based on quantitative or qualitative analysis.

ATAM provides a way of determining technical measures of a system's quality goals resulting from a proposed architecture, and thus (viewing the architecture as a set of architectural decisions) from proposed architectural decisions. Software development organizations, however, usually need to consider the costs incurred with these decisions and to balance this with the benefits gained. This need is addressed by an extension of ATAM called the *cost benefit analysis method* (CBAM) [4]. The purpose of CBAM is to calculate the return on investment (ROI) for each proposed architectural strategy. The inputs to this calculation are estimated costs of architectural strategies and measures of the corresponding benefits derived from the ATAM. For a specific architectural strategy, the benefit B_i is defined as:

$$B_i = \sum_j (b_{i,j} \times W_j)$$

where $b_{i,j}$ is the benefit of strategy i in scenario j and W_j is a weight assigned to scenario j , reflecting its relative importance. Each $b_{i,j}$ is the estimated effect of strategy i on the quality goal analyzed in scenario j . If U_{expected} is the measure of the quality goal obtained from ATAM in scenario j when strategy i is included in the architecture and U_{current} is the measure when the strategy is excluded, then $b_{i,j} = U_{\text{expected}} - U_{\text{current}}$. The measures of the quality goals are numbers between 0 and 100, corresponding to the worst-case and best-case situa-

tions respectively. For an architectural strategy with cost C_i and benefit B_i , the ROI value is calculated as:

$$R_i = \frac{B_i}{C_i}$$

Techniques for cost estimation have been widely studied and reported, for instance by Boehm and others [15].

Another analysis method is the *architecture-level modifiability analysis method* (ALMA) [16] by Bengtsson and others. As the name indicates, this method focuses particularly on analyzing the modifiability of a system based on a proposed architecture for the system. Like ATAM, ALMA is scenario-based. The only scenarios considered are change scenarios, and the output of running a scenario consists of measures of the impact of the change on the system and the effort required to implement the change. Depending on the purpose of the analysis this can be described qualitatively or quantitatively. Yet another development is reported by Svahnberg [17]. This work extends the state of the art in architecture evaluation with a quantitative method for selecting between candidate architectures. The first step of the method is to define a set of quality goals as the base for the selection and assign numerical values to these goals that determine their relative importance. The next step is to evaluate each of the candidate architectures with respect to each quality goal, which results in a matrix of numerical scores. These scores need not be meaningful absolute measures of each architecture's ability to meet the quality goals, as long as they serve to relate the abilities of the architectures to each other. By weighing the scores with the importance of each quality goals, the best architecture can finally be determined.

Analysis of software architectures is not only useful for selecting between candidate architectures. Land [18] presents various strategies for in-house integration of software systems – i.e. integration of two software systems owned by the same organization. One of these strategies is to develop a new system by merging the existing systems, which may be done rapidly or evolutionary. A technique for analyzing the systems' software architectures with respect to their similarity is suggested as a primary tool for deciding whether merging is a feasible strategy. The analysis is combined with business and other considerations to determine if rapid or evolutionary merging is more suitable. Several empirical studies are presented, demonstrating that failing to take this analysis into account is likely to result in unsuccessful merging efforts.

2.1.4 Architectural Description and Documentation

In practice, software architectures are usually described using informal box and line diagrams accompanied by descriptive prose. The research community has pointed out that such descriptions are often ambiguous and there is extensive work on architectural description and documentation in the literature.

One research direction is the development of *architecture description languages* (ADLs). A bafflingly high number of such languages have been published, differing in such aspects as use of graphics or text, formality of semantics, emphasis on certain domains or styles, available analyses and tool support etc. In [4], Shaw and Garlan discusses the requirements for ADLs and reviews three early languages and their associated tools. A recent and extensive survey is that of Medvidovic and Taylor [19]. Despite the great volume of work on ADLs there are few testimonies of industrial adoption in the literature. The use of the Koala language at Philips [20] is perhaps the only reported example. This language is fairly implementation-oriented and can be seen as something on the borderline between an ADL and a graphical programming language. Koala is furthermore the name of a related software component model, which is discussed in Section 2.2.2 of this dissertation.

A language that has been widely adopted is the *Unified Modeling Language* (UML) [21]. Although UML has become the standard notation for documenting software design, its suitability for describing software architecture has been questioned. The problem is that UML has its roots in object-oriented methods and is mainly intended for modeling a system as a set of interrelated classes, a concept usually considered to be at a lower level of granularity than software architecture. Still, it has been demonstrated how the language can be used for architectural documentation. One example is the aforementioned approach of Siemens Corporate Research [9]. Their architecture descriptions are written using special architecture-level modeling elements, which have been defined using UML's extensibility mechanisms. Although it would be possible for other organization to re-use these architecture-level modeling elements, it is not likely to occur on a large scale until such elements are standardized and supported by major tool vendors.

Fortunately, such standardization has now taken place in UML 2.0 [22]. This new standard defines the following architectural concepts, which are also central in most ADLs:

- *Component.* A component is a modular unit with well-defined interfaces that is replaceable within its environment. The external view of a component is a set of provided and required interfaces, which may be exposed via ports (see below). A component may also have an internal view in the form of a realization, which is a set of instances of classes or smaller components that collaborate to implement the services exposed by the component's provided interfaces while relying on the services of its required interfaces. The concept can be used to specify both logical and physical components.
- *Port.* A port is a named and typed interaction point of a component. A provided port is typed by a provided interface, a required port by a required interface, and a complex port by an arbitrary set of provided and required interfaces. Complex ports enable the localization of complex interaction patterns where calls may occur in both directions. Unlike interfaces, a port may be associated with a behavior, specifying the externally observable behavior of the component when interacting through the port. This allows the specification of semantic contracts, similar to those described in Paper A. A component may have multiple ports typed by the same interface, and is able to distinguish between calls received through different ports.
- *Connector.* A connector is a link that may be of kind delegation or assembly. A delegation connector either links a provided port of a component to a part of the component's realization, signifying that requests received through the port is forwarded to the part, or it links a realization part to a required port, signifying that request sent through the port originates in the part. Several connections may exist between a single port and different realization parts. An assembly connector links a required interface or port of a component to a matching provided interface or port of another component.

Figure 2-1 is a UML 2.0 diagram that illustrates these modeling elements. The diagram shows a component with one port, typed by one required and one provided interface. The component also has a realization, consisting of two component instances. Delegation connectors link the outer component's port to a provided port of one of these instances and a required port of the other instance to the outer port. The two instances furthermore have ports linked by an assembly connector. The diagram does not show port names.

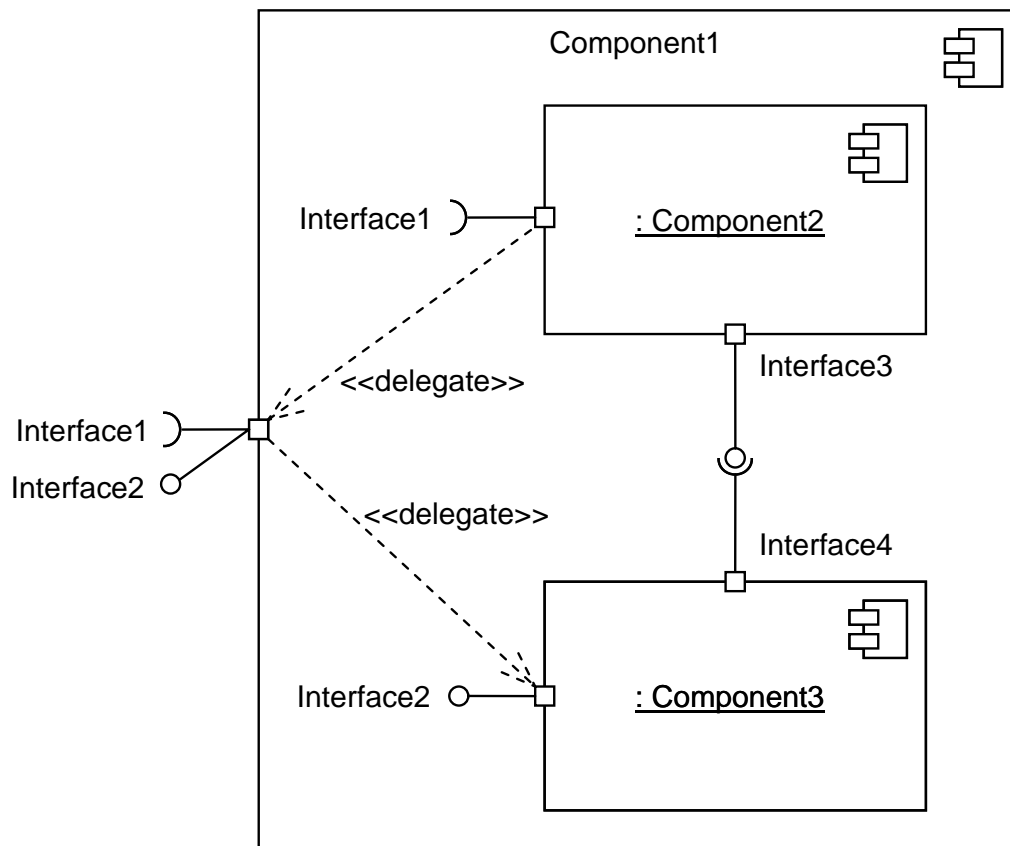


Figure 2-1 Architectural modeling elements in UML 2.0

The production of professional software architecture documentation has been studied at the Software Engineering Institute [7]. This work focuses more on the organization of architecture documents than on particular notations. The central organizing unit for such documents is that of a view, which is defined as follows:

A view is a representation of a set of system elements and the relationships associated with them.

Thus, a view represents a subset of the information contained in an architecture. The use of views is motivated by the fact that software architectures are complex entities that cannot be adequately described in a simple one-dimensional fashion.

One of the most influential publications on architectural views is Kruchten's paper on the *4+1 view model* [23]. His approach, which has been adopted as a central part of the Rational Unified Process, defines the following views:

- The *logical view* primarily supports behavioral requirements: the services the system should provide to its end users.
- The *process view* addresses concurrency and distribution, system integrity, and fault tolerance.
- The *development view* focuses on the organization of the software modules in the software development environment.
- The *physical view* maps the various elements identified in the logical, process, and development views onto the processing nodes.
- The *use case view* contains a small subset of important use cases, intended to show that the elements of the other four views work together seamlessly.

The last view is called the +1 view since it is redundant with, and serves to validate, the other views. Another model that has received considerable attention is sometimes called the *Siemens 4 view architecture model* and is a central part of Siemens Corporate Research's approach [9], mentioned above. It defines the following views:

- The *conceptual view* describes the system in terms of its major design elements and the relationships among them.
- The *module interconnection view* describes functional decomposition and layering.
- The *execution view* describes the dynamic structure of a system.
- The *code view* describes how the source code, binaries, and libraries are organized in the development environment.

The conceptual view has no direct counterpart in the 4+1 view model, while the module interconnection view corresponds roughly to the logical view, the execution view to the process and physical views, and the code view to the development view.

certain subset of stakeholders and concerns. A view is a system specific instance of a viewpoint. The viewpoint specifies the format for describing the view, including languages and notations used as well as any analysis technique that may be applied. The architectural description shall state which viewpoints are used and present the specification of these or refer to other documents where specifications may be found. The standard emphasizes the potential for reuse of viewpoints, and therefore states that a viewpoint may be a library viewpoint. The architectural description is required to include at least one view and (a reference to) a corresponding viewpoint, but there are no pre-defined compulsory views. Consequently, the standard does not prescribe the use of any particular language or notation.

2.2 Component-Based Software Engineering

The field of *component-based software engineering* (CBSE) is concerned with the development of software by assembling pre-existing smaller pieces, which are termed *software components*. Within the field of software architecture there is a widely accepted terminology where the constituent parts of a system's architecture are also called components. This sometimes creates confusion since the architecture and CBSE communities have adopted the term component independently. A widespread view in CBSE is that a software component denotes a physical part (product), while in architecture a component can be any structural entity (file/class, process/thread, module/layer, etc.) and even purely conceptual (e.g. an abstraction invented by a designer). At the risk of adding to the confusion, this dissertation uses the term *component-based software architecture*, in particular in Chapter 5, to mean a software architecture designed to support CBSE.

2.2.1 Definitions of Software Components

The key concept of CBSE is that of software components – e.g. those pieces of software that may be assembled into larger components or final products. Clearly, how the concept of software components is defined has ramifications for the practices of CBSE. This section reviews and discusses some of the different definitions found in the literature.

One of the most influential definitions of software components is that of Szypersky [24]:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

The first part of the definition is technical, and states that software components should be “black-boxes” to be composed without modification (obviously, the definition means that interfaces and context dependencies are the only *visible* parts of a component). Szypersky asserts that source code modules do not qualify as software components since they make it possible for the composer to rely on implementation details, thus violating the principle of black-box composition. The second part of the definition is more market-oriented, effectively stating that it should be possible to market software components as independent products and that buyers should be able to use them as parts in their own products. Naturally, independent deployment also has technical implications, namely that it must be possible to deploy (e.g. upgrade) a single component without any modification, recompilation, or similar of the rest of the systems of which the component is a part.

In what is sometimes called *The CBSE Handbook* [25] Heineman and Councilill present the following definition:

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

According to this definition, all components must conform to a component model, which the authors define as specifying interaction and composition standards. This requirement is quite reasonable, since it is hard to see how CBSE could work without some standards for interaction and composition. It is worth noting that the definition does not require that the component model is defined by a standards body or platform supplier, or that a commercial platform implementation is used. It is furthermore concluded that the two definitions principally agree, since the requirement that components can be modified without modification can only be satisfied if interfaces and context dependencies are well defined and that compliance with a standard naturally supports composition by third parties.

A definition of software components that must be expected also to receive widespread attention is that of UML 2.0, which has already been discussed in connection with architectural description in this dissertation. From the discussion of the previous section, the following definition can be extracted:

A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment. The concept can be used to model both logical and physical components.

In the context of CBSE, a software component corresponds to what UML 2.0 calls physical components. Although some will object to the use of the word physical to describe software components, this is the term used by the UML 2.0 specification to denote deliverables such as COM+, EJB, or CCM components. The definition is somewhat broader than the previous two, as “replaceable within its environment” is a weaker requirement than “subject to independent deployment and composition by third parties”. The term physical component is intended to cover such entities as executables and dynamic link libraries, which do not comply with a component model. The definition is interesting primarily as it helps to establish required and provided interfaces as part of the standard terminology of software components.

This terminology is also used by Crnkovic and Larsson [26], who define a software component as consisting of at least the following elements:

- A set of interfaces provided to, or required from the environment. These interfaces are particularly for interaction with other components, rather than with a component infrastructure or traditional software entities.
- An executable code, which can be coupled to the code of other components via interfaces.

This definition emphasizes that a component’s interfaces are intended to support interaction with other components. Consequently, these components must agree on some format for interfaces and patterns of interaction, which is another way of saying that they must conform to a component model. The definition explicitly states that a component contains executable code. This is not an important difference with the other definitions, however, since these also presume that components contain executable code and focus on properties distinguishing software components from other executable formats. Based on the above definitions, which represent some of the most prominent literary sources on CBSE, it is concluded that there is significant consensus that a software component encapsulates executable code while complying with a component model that supports composition and interoperability.

2.2.2 Software Component Models and Technologies

As already mentioned, a *software component model* specifies standards for composition of and interaction between software components. Among the things that such a standard may specify are data interchange formats, interaction patterns, and run-time services. While incompatible data formats may be a barrier for interoperation between independently developed pieces of software, a more serious problem is that of incompatible assumptions about the overall architecture of the system that these pieces will be parts of. This problem of *architectural mismatch* was identified and characterized by Garlan and others [27]. Software component models may be viewed as a way to avoid architectural mismatch by standardizing certain architectural choices. To facilitate the use of software component models, dedicated software tools and infrastructures are often implemented. These may include run-time environments for component execution and interaction as well as tools for component development, composition, and deployment. This dissertation uses the term *software component technology* to denote a set of dedicated software products supporting the use of a specific software component model. Heineman and Council use the term *component model implementation* to denote the run-time parts of a software component technology.

One of the most widely used component models is Microsoft's *Component Object Model* (COM) [28]. Microsoft first used this model internally, in its *Windows* operating systems [29] as well as in applications available on that platform, before releasing the COM specification. Thus, in this case, a component technology already existed when the component model was published. Today, there are numerous vendors of COM components and COM-based applications for the *Windows* platform. Technologies are also available on several other platforms, but COM has never gained widespread popularity outside the world of *Windows*. Although the model is primarily associated with desktop applications, COM implementations are also available for a few real-time operating systems, such as *Windows CE* [30] and *VxWorks* [31].

On the *Windows* platform, a *COM component* is an executable or dynamic link library (DLL) that implements a set of *COM classes* that each implements a set of *COM interfaces*. Classes may also have optional or required *outgoing interfaces*, i.e. interfaces to be used by the classes and implemented by other components. Both classes and interfaces are identified by *globally unique identifiers* (GUIDs), which are 128-bit numbers that can be generated by an algorithm that virtually ensures their uniqueness. The GUIDs of any classes imple-

mented by the components installed on a system are stored in the Windows registry along with references to the implementing components. The *COM library* is a part of the Windows run-time system and provides an API that an application or component, called a *COM client*, can use to create COM objects by supplying the GUIDs of the desired class and interface. COM does not specify how classes should be implemented. Instead, components are required to provide a *factory interface* that the COM library uses to instruct components to instantiate their own classes.

What COM *does* specify is the binary format of interfaces. A client interacts with a COM object through a pointer to an *interface node*, which includes a pointer to a table of function pointers. Since the interface standard is binary, COM is oblivious to the programming languages use to implement components and clients. Once the COM library has created an object, it returns a pointer to one of the object's interfaces to the client. The client can use an operation of this interface to request pointers to any other interfaces the object supports. This technique is called *interface navigation*. In addition, the COM specification includes a set of predefined interfaces for such purposes as scripting, error handling, and connection-oriented composition. *Distributed COM* (DCOM) [32] is an extension of COM that supports distributing applications across physical machine boundaries. The basic interoperability mechanisms of COM and DCOM are discussed more deeply in Chapter 4 of this dissertation.

A special type of COM components is *ActiveX controls* [33]. These components implement and use predefined interfaces, which are designed to allow interaction with both (visual) composition tools and run-time environments, called containers. A typical application is in graphical user interface (GUI) controls, including controls automatically downloaded from web servers and executed in a web browser. Typically, such controls make use of outgoing interfaces to notify their containing application or web browser of events. A similar component model is Sun's *JavaBeans* [34]. These components are built from Java classes that implement predefined interfaces and use special event objects for notification. JavaBeans share many of the characteristics of ActiveX controls, the main difference being that they must be written in the *Java* programming language [35] and executed on a *Java virtual machine* (JVM) [36]. Many web browsers include a JVM and, as with ActiveX controls, enhancement of web pages is a common use of JavaBeans. Sun provides a solution that makes it possible to use JavaBeans in ActiveX containers. Component technologies related to ActiveX controls and JavaBeans include tools for packaging and deployment of components with associated resources and type information.

COM+ [37] is an extension of COM incorporating support for services, such as transactional processing, message queuing, and security management that are commonly used in distributed information systems. These services are not invoked programmatically from inside the components. Instead, declarative attributes can be associated with components and applications, specifying which services can or must be provided and at which level. This information is stored in a special-purpose repository called the *COM+ catalog* rather than in the Windows registry. The COM+ run-time system uses this information to intercept component interactions and insert system calls as required. This allows existing COM components to be transparently augmented with, for instance, transactional processing and used as part of COM+ applications.

More recent versions of Windows include Microsoft's .NET technology [38], which is perhaps best known as a platform for implementing web services [39] but also defines a new software component model. In this model, components are called *assemblies* (a potential source of confusion as this term is sometimes used in CBSE literature to denote a collection of components). A .NET assembly contains a *manifest* in addition to a collection of types, i.e. classes and interfaces, and/or resources. Strictly speaking, an assembly is only a software component as defined in the previous section if it contains at least one class. The manifest contains the assembly's metadata, including its name, version, contents, dependencies, and so on. Thus, information about classes is stored in the component itself rather than in the Windows registry. Other differences from COM are that a .NET assembly can be distributed across several files and that a hierarchical naming scheme is used, where an assembly is associated with a GUID and the names of its types need only be unique within the assembly.

A more important difference between .NET and COM is that the former's runtime system, called the *.NET Framework*, offers much more functionality than the COM library, relieving .NET components from much of the "housekeeping" tasks of COM components while greatly enhancing the support for such things as versioning, security, and memory management. (This housekeeping is typically not a burden for developers of COM components, however, as it is handled by code automatically generated by development tools.) Unlike in COM, the classes of a .NET assembly are not implemented as native executable code, but as *Microsoft Intermediate Language* (MSIL) code to be executed by the .NET Framework, usually via just-in-time (JIT) compilation. The introduction of .NET does not mean that COM has been removed, and Microsoft offers a solution for interoperation between .NET and COM components. The ser-

vices of COM+ are also available in the .NET Framework, under the name of *Enterprise Services*. The *.NET Compact Framework* [40] is a down-scalable version of the framework for resource constrained embedded systems. This version is not particularly suited for real-time applications, however, as it uses the same automatic memory management (garbage collection) as the standard version, resulting in a loss of predictable timing.

Another model providing services similar to those of COM+ is Sun's *Enterprise JavaBeans* (EJB) [41], which is based on Java but not on the aforementioned JavaBeans model. The required service levels for a set of EJB components are expressed declaratively in a file called a *deployment descriptor*. After deployment, each of the objects implemented by the components, generally called *beans*, live inside an *EJB container*, which also contains objects generated from the deployment descriptor. Clients invoke a bean's operations via these generated objects, which ensure the correct service levels. Unlike JavaBeans, beans in EJB do not communicate through events. There are two principal types of beans. *Entity beans* are used to encapsulate access to database records. An entity bean may implement its own persistence management or let the container manage persistence as specified by the deployment descriptor. *Session beans*, which may be stateful or stateless, represent interaction sessions with clients. *Message-driven beans* can be seen as a special kind of stateless session beans that represent asynchronous interaction session. A session bean may control transactions or leave that to the container. EJB requires the *Java 2 Enterprise Edition* (J2EE) platform [42].

A third model that is similar to COM+ and EJB is the *CORBA Component Model* (CCM) [43]. CCM is standardized by the Object Management Group (OMG) and require that clients and components communicate using an *object request broker* (ORB) as defined by version 3.0 of the OMG's *Common ORB Architecture* (CORBA) [44]. A CCM component is a package, which contains an XML description and possibly binaries for multiple platforms. A CCM application is an assembly of CCM and possibly EJB components, whose configuration is described in an XML document. A CCM component belongs to one of four possible categories. *Service components* correspond to stateless session beans in EJB, and maintain no state. *Session components* correspond to stateful beans and maintain state for the duration of a transaction. *Entity components*, as entity beans, encapsulate database access. *Process components* maintain persistent state throughout the lifetime of a process. Similarly to in EJB, the instances of a CCM component resides within a CCM container, and transaction control as well as persistence may be container managed or self managed. Clients inter-

act with CCM components through *attributes* and *ports*. A port is a *facet*, a *receptacle*, an *event source*, or an *event sink*. Facets and receptacles are provided and required interfaces, respectively. Event sources and sinks are connected via *event channels*. CCM also specify two predefined interfaces that are clearly inspired by COM. All component instances provide the *equivalence interface* for interface navigation and all components implement the *home interface* for instance creation.

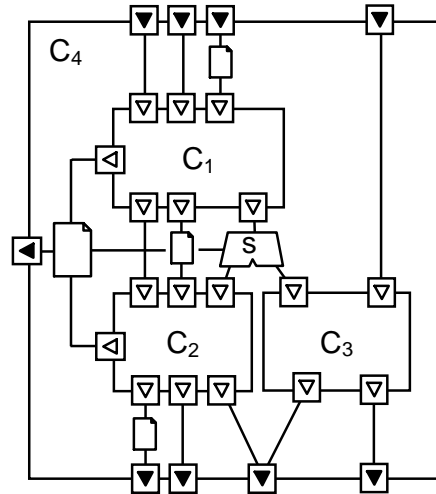


Figure 2-3 A Koala configuration

The *Koala* component model [20] is specifically intended for embedded software in consumer products. In particular, it is being used by Philips in products such as televisions and VCRs. A Koala component has a set of provided and required interfaces, and interacts with its environment through these interfaces only. A Koala configuration specifies a collection of component instances, the *parts list*, and a set of connections between these instances, the *net list*. In the simplest case, a connection links a required interface of one component instance to a matching provided interface of another component instance. Glue code may be associated with connections to provide more complex interactions. Configurations may themselves be used as components in a hierarchical fashion. Koala provides notations for specifying interfaces and components and a graphical language for defining configurations. Figure 2-3 shows such a configuration defining a component consisting of three sub-components. Basic Koala components, i.e. those that are not configurations, are sets of C source code files. As such they do not satisfy the definitions of software components discussed above. However, the motivation for using source code is efficiency,

not exposition of implementation details, and the Koala configuration language encourages black-box composition. The Koala compiler optimizes configurations by inserting into the code of the components static references to connected components wherever possible. Still, puritans may prefer to view Koala as a technology for modular, graphical programming rather than a component model. For instance, it does not support independent component deployment as discussed in the previous.

The component models discussed above are all based on a notion of software components that encapsulate executable code and expose services through interfaces. Thus, they are compatible with the definitions of the previous section. With the exception of Koala, however, they all allow components to interact with their environment without explicitly declaring required interfaces. For instance, COM components can use the services of the Win32 API and JavaBeans those of the standard Java class libraries. Similarly, in all models except Koala, the services exposed through a component's interfaces can be accessed by components that do not declare required interfaces and even by software that are not components. Regarding the standardization of architectural choices, all the models stipulate an object-oriented style. Other architectural aspects are address to a varying degree. All models specify a standard for allocating and releasing the resources used by components: in COM and Koala, this is the responsibility of the components themselves while, in the other models, it is handled more automatically. Koala furthermore defines a standard for concurrent execution, while the other models leave it to the components to use the mechanisms of the underlying platforms for this purpose. Models like COM+ and EJB define standard solutions for such things as transaction handling and persistent data management, which can be viewed as architectural decisions, but give component developers the choice to implement their own solutions instead.

All the models described above are supported by component technologies, including run-time infrastructures providing varying degrees of functionality. One of the smaller and simpler run-time infrastructures is that of COM. As discussed above, one consequence of this is that COM components are required to perform more "housekeeping" than when a richer run-time system is provided, as for .NET components. Developers of COM component are typically not required to write more housekeeping code, however, as this is most often provided by development environments in automatically generated code. This illustrates how one part of a component technology - development tools - can serve as a substitute for another - run-time infrastructure. The ex-

ample of development tools to support COM components is interesting for another reason. While the run-time parts of a component technology can be viewed as the most central parts in some senses, e.g. they are typically the only part explicitly required by the component model specification, other parts are often essential as well. The widespread use of COM would be very unlikely without the comprehensive support for the model provided by tools like *Microsoft Visual Studio* [45]. Other development-time technologies include graphical GUI builders for ActiveX and JavaBeans, highly automated development environments for EJB, and the graphical Koala editor. Examples of deployment-time component technologies include those that support downloading and installing ActiveX controls and JavaBeans used on web pages.

As noted by e.g. Wallnau and others [46], software component models are closely related to the concept of architectural styles. Thus, as discussed in the previous section, one may expect the choice of a component model to affect a system's properties in a predictable way. The component models discussed above each defines one or more types of components as well as different ways in which such components may be connected. Not surprisingly, the object-oriented systems style is evident in all these models. This style corresponds directly to the way that EJB systems and most COM-based systems are organized. ActiveX, JavaBeans, and CCM correspond to an object-oriented, event-based systems style, which may also be used with COM/COM+. Recall that the primary assumed benefit of the object-oriented systems style is encapsulation of implementation details, while the event-based systems style is assumed to result in increased extensibility. Koala differs from the other discussed models in that components are explicitly disallowed to contain references to other components. In a way, this resembles the pipe and filters style, and might be expected to promote reusability.

The definition of architectural style presented in the previous section states that a style might include one or more semantic models that allow a system's properties to be inferred from the properties of its parts. No such models are included in any of the component models discussed above, however, and this seems also to be the case for other models. This is being addressed by the Software Engineering Institute's work on *prediction enabled component technology* (PECT) [47]. A PECT is defined as consisting of a *constructive model*, which, like the component models discussed so far, supports the implementation of systems as assemblies of components, and a set of *analytical models*, which define techniques for predicting different properties of assemblies from the properties of their constituent components.

2.2.3 Software Component Services

Software component services are a way to provide functionality in component-based applications without components having to implement this functionality or invoke operations that provide it. As described briefly in the previous section, component models like COM+, EJB, and CCM specify various such services of general use for distributed information systems. Software component services are a special case of what may be termed software services. This dissertation deals in particular with services that are provided by a run-time system on the basis of declarative attributes as described below.

The basic principle of software services and declarative attributes can be illustrated by the simple example of a console application on the Microsoft Windows operating system. (This example was used by Don Box in a talk on COM+ at the 2000 *Microsoft Tech-Ed Conference*.) A console application is a program with character-based input and output. When such a program is executed in any other way than entering its name on a command line – e.g. by double-clicking the program file in the Windows Explorer – a new console window is first created, and the program then executes while performing its input and output through that window. To create a new console window in Windows, a function of the Win32 API called *AllocConsole* is used [48]. However, when writing a console application, e.g. using Visual Studio, the programmer is not required to include a call to this function. Furthermore, the development environment does not automatically provide any code that calls *AllocConsole* and inspecting the executable file after building will reveal that the program does not call this function at all. Instead, the executable is marked with a flag (i.e. a declarative attribute) that informs the run-time system that it is a console application, and the system uses this information to create a new console window when needed.

The console application example illustrates how declarative attributes can be used to augment software with functionality not implemented or invoked by the software itself. Naturally, this relies on a run-time infrastructure to interpret the attributes and provide the requested services. This need may be filled by the operating systems, as in the example, or by some other software running on top of the operating system, e.g. middleware. An example of middleware intended for embedded real-time systems is MEAD [49], which provides services for fault-tolerant real-time systems, such as application replication. Services are provided transparently by *interception* of messages between applications and the run-time system. This is achieved by dynamically linking ap-

plications with an interceptor providing the same interface as the run-time system. Thus, applications can invoke operations without even being aware of the interceptor. The interceptor forwards invocations to the run-time system and performs any additional processing required to provide the desired services. MEAD has been implemented on top of a CORBA implementation for real-time systems.

In the case of software component services, the services in question are provided by a component model implementation and augment the functionality delivered by software components. Weinreich and Sametinger [50] define a component model implementation as providing a run-time environment and services. They furthermore distinguish between *general services*, which are independent of application domain, *horizontal services*, which targets multiple but not all domains, and *vertical services*, which are specialized for a particular domain. Typically, the vertical services are implemented on top of the horizontal, which are in turn implemented on top of the general. Among the examples of general services that the authors mention are instantiation in object-based models, location transparency in distributed models, and transaction handling in models for information systems. Thus, all the services mentioned in the previous discussion of component models are general. As examples of horizontal and vertical services, Weinreich and Sametinger mention compound documents and the more specialized of the CORBA specification's *CORBAfacilities*, respectively.

A concrete example of software component services that rely on declarative attributes is COM+ services. As already noted, COM+ extends COM and DCOM with services of general use for distributed information systems, and these services are now also available for use with .NET components. Löwy [51] describes the services defined by version 1.0 of COM+. More services have been added in newer versions, but the simpler original version works well as an illustrative example. Unlike EJB and CMM, COM+ does not include any service for data persistence. Instead, COM-based applications can use *ActiveX Data Objects* (ADO) [52] and .NET applications the *ADO .NET* library [53] for accessing data bases. The list below summarizes Löwy's description of the COM+ services.

- *Administration*. Tools and services for configuring components and applications, such as the COM+ catalog.

- *Just-in-time activation (JITA)*. Services for creating and discarding object instances. These play an essential role in supporting run-time services by interception, as described below.
- *Object pooling*. Services that support sharing instances of frequently used and expensive resources between clients. This may often be used to improve performance.
- *Transactions*. Services for treating sequences of operations by distributed components as single atomic operations. This is a common requirement in commercial systems.
- *Synchronization*. Services for controlling concurrent activities.
- *Security*. Services that perform authentication and access control.
- *Queued components*. Services for asynchronous and disconnected communication between components.
- *Events*. Services that provide publish-subscribe event notification.

The administration services are mainly concerned with configuration of services as declarative attributes, which may be specified on the application, component, interface, or operation level. The other services are run-time services, which are mainly provided by interception of operation invocations. This is achieved by the use of proxy objects placed between components and clients. These are created along with other object instances by the JITA services, which inspect the configurations in the COM+ catalog. COM+ uses the term *lightweight* proxies for the objects that implement services and the processing involved in providing a service is called performing a *service switch*. This terminology is probably intended to create an analogy with the proxy objects used in COM/DCOM to support invocation across process and machine boundaries, which involve performing context switches. COM+ proxies are lightweight in the sense that the overheads associated with service switches are only a fraction of those resulting from context switches. Another difference between COM+ and COM/DCOM proxies is that the former is generated at run-time. In the case where multiple services are required, e.g. both synchronization and security, Löwy indicates that multiple proxies are created:

The exact way the lightweight proxies mechanism is implemented is not documented or widely known. However, in this case, COM+ probably does not gen-

erate just one lightweight proxy to do multiple service switches, but rather puts in place as many lightweight proxies as needed, one for every service switch.

Figure 2-4 illustrates the use of two proxies to provide synchronization and security for an object's operations. In this situation, an object C1 provides operations to clients through an interface IC1. Due to the configuration in the COM+ catalog, the JITA services have created the two proxies and placed them between the object and its clients, as shown in the figure. Clients invoke the operations of the IC1 interface on the first proxy, which performs synchronization between concurrent invocations and forwards invocations to the other proxy. This proxy, in turn, forwards invocations to the C1 object, while ensuring that security policies are met, e.g. by authenticating the identity of the calling clients and only forwarding invocations from authorized clients.

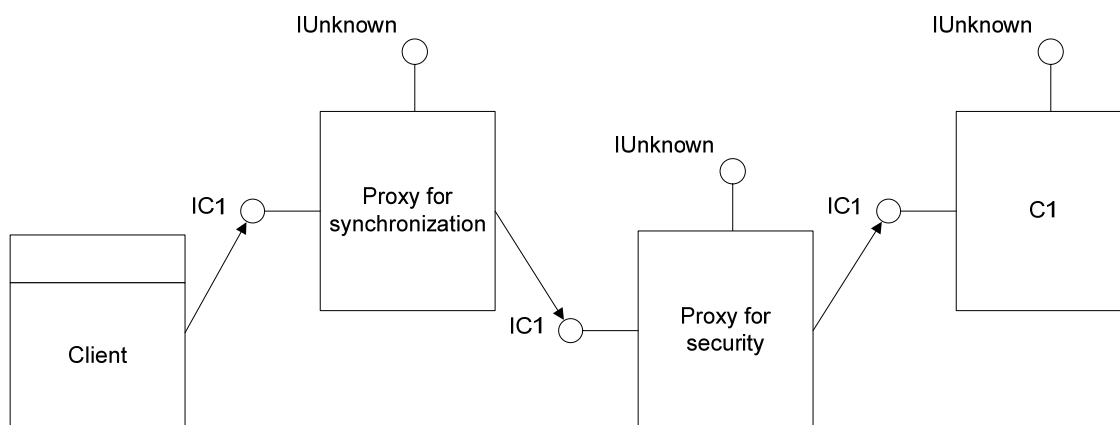


Figure 2-4 Lightweight proxies to provide services in COM+

The use of proxies created at the time of object instantiation is a practical way of implementing interception. Since the operations of COM objects are invoked via interface pointers, which are based on simple function pointers, there is no straightforward way for the run-time system to intercept invocations made directly to a COM object. A proxy always implements the same set of interfaces as the object it encapsulates, thus preserving interface navigation. The fact that proxies are generated by the JITA services at run-time means that instantiation must take some additional time in COM+. An alternative approach that might result in less additional time would be to generate and store proxy object implementations at configuration time.

2.2.4 Component-Based Software Engineering Practices

As already mentioned, CBSE denotes the practice of assembling software from existing components. Thus, in comparison to traditional software engineering, the activity of assembling replaces that of programming. In practice, however, some programming is usually needed to make a set of independently developed component work together. Furthermore, traditional development models, where design and implementation follows strictly from a preceding stage of requirements identification, is less suited for CBSE, where it is usually necessary also to adjust requirements to match what available components can offer. For reference, Figure 2-5 is a simple UML activity diagram illustrating the traditional waterfall model of software development [54]. In more modern models, such as the Rational Unified Process [10], these activities are repeated iteratively.

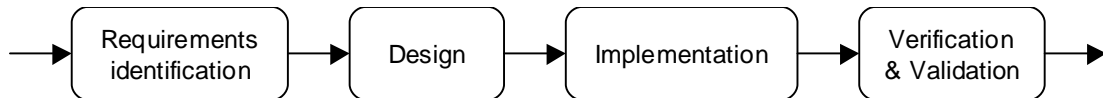


Figure 2-5 Waterfall model of software development

Among the first to address the particular practices required for component-based software in a systematic fashion were Brown and Wallnau [55], who define a reference model for such systems. As illustrated in Figure 2-6, the model focuses on the system as a set of components that progress through various states during development and evolution. Off-the-shelf components are pre-existing components that may have been acquired externally or reused from previous projects within the development organization. They are characterized by having hidden interfaces, where interface is interpreted to include not only a functional description but also all other information that is needed to use a component. Qualification is the process of discovering the hidden parts of the interfaces. The qualified components are subsequently adapted to remove architectural mismatch – i.e. mismatched assumptions about the system’s architecture [26]. Adaptation is usually accomplished by writing wrappers. The adapted components are composed according to a selected architectural style. As discussed in the previous section, selecting a component model in part determines this architectural style. Composition may include writing some additional code, which is often call glue code. The system finally enters a stage of evolution where component may be updated.

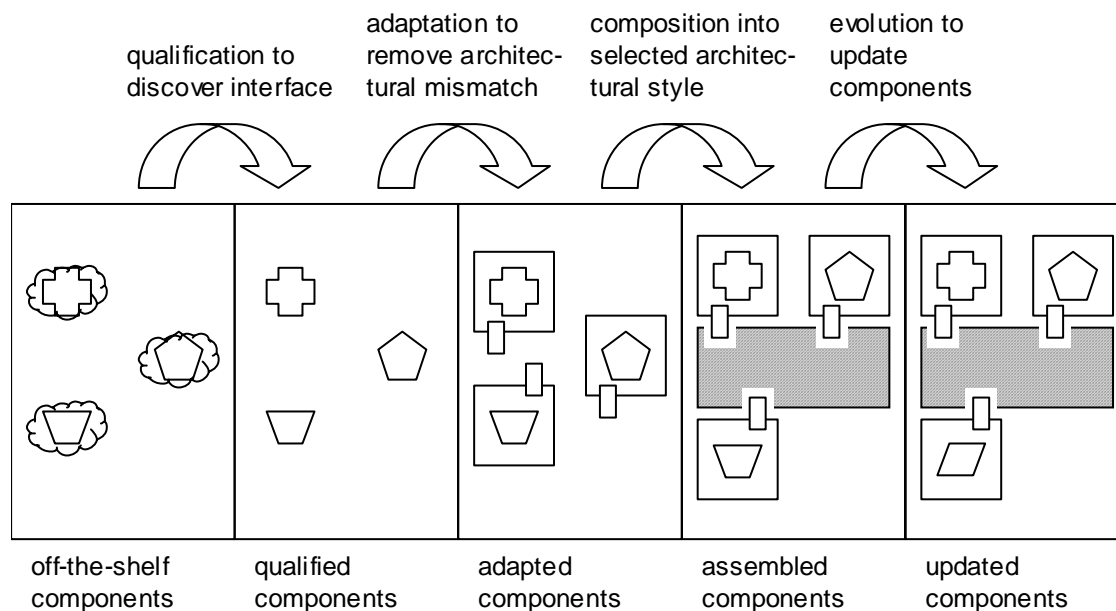


Figure 2-6 Reference model for composition of software components

A central aspect of this model is the assumption that components initially have hidden interfaces, which is particularly important when using commercial components. This work has been extended by Wallnau and others [56], with an even more pronounced focus on commercial components. A central concept of the work is that of an *assembly* – a set of interoperating components that may form part of a system. It is for instance argued that assemblies are more useful as units of evaluation and selection than individual components.

In other component-based systems, as in that of the industrial case study presented in this dissertation, components are implemented to comply with pre-specified interfaces. In these cases, the activities of requirements identification and design will be less different from traditional software engineering, since there is no evaluation, selection, qualification, or adaptation of existing components. However, an essential goal of the design activity is to identify the components to be developed and allocate functionality to them. This can be seen as input for identifying requirements for each component, which can subsequently be independently developed and tested. This leads to a form of nested development process where similar activities are performed on both system and component levels. Based on the waterfall model in Figure 2-5, this can be depicted as in Figure 2-7.

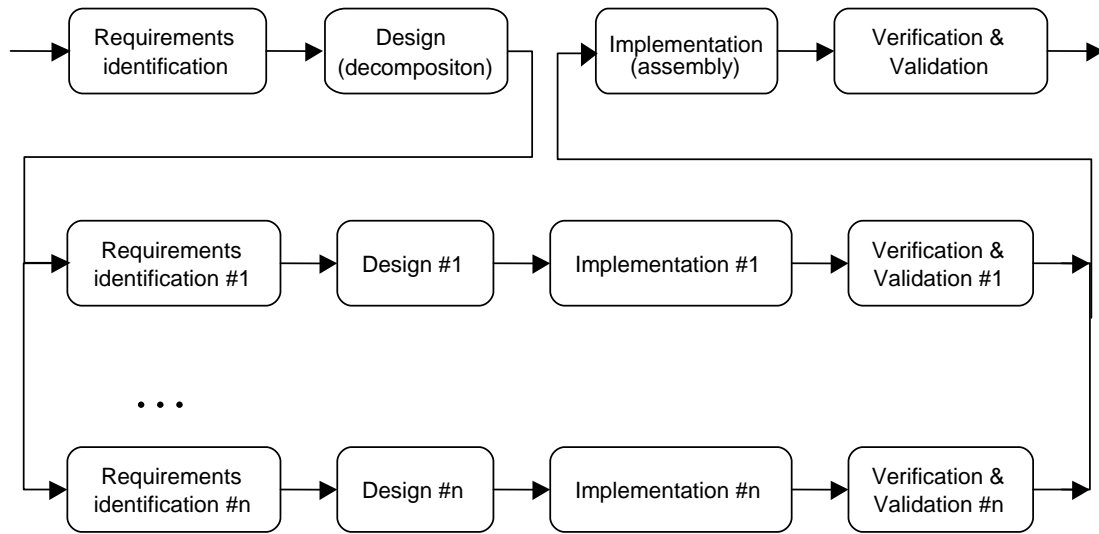


Figure 2-7 Waterfall model adopted for component-based software development

In addition to the practices of developing component-based system, the literature also discusses non-technical aspects of CBSE. For instance, Szyperski [24] points out that a component market of critical size is needed for the development of commercial components to represent a viable business opportunity. Another example is Heineman and Council's book [25], which covers regulatory and legal issues, such as the applicability of commercial law to software components.

2.3 Embedded Real-Time Systems

Embedded real-time computing systems occur as part of many different products and systems, including cell phones, refrigerators, cars, airplanes, and industrial plants. A persistent trend in the development of such systems is the increasing amount of functionality implemented in software. For instance, as noted by Atkinson and others their book [57], the amount of software in cars has grown from about 100 kilobytes 15 years ago to a projected 1 gigabyte in the latest high-end models. In the foreword to the book, Gemund notes that the development of such software tend to be costly and cites estimated costs of US \$15-30 per line of code in consumer products, \$100 in defense applications, and \$1000 in highly critical systems like space shuttles. Thus, a primary challenge related to embedded real-time systems is improved methods for cost efficient development of software for such systems.

2.3.1 Definitions of Embedded Real-Time Systems

There are many definitions of real-time systems in the literature, most of which states that the correctness of such system depends not only on computed outputs, but also on the times at which these outputs are delivered. It has been argued that the term “depending on time” is quite vague and may be used to argue that any computer system is a real-time system – e.g. any useful system must produce output in finite time. In one of the more influential books on real-time systems [58], Laplante attempts to formulate a more precise definition of real-time software systems by first defining the concept of *response time*:

The time between the presentation of a set of inputs to a software system and the appearance of all the associated outputs is called *response time* of the software systems.

Based on this, the following definition is given:

A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.

This is furthermore augmented with a definition of what it means for a system to fail:

A failed system is a system that cannot satisfy one or more of the requirements stipulated in the formal systems specification.

Thus, in order to avoid failure, the design of a real-time software system must ensure that the system can meet its response time requirements. A consequence is that the software may only be run on top of a platform that allows such assertions to be made. Today, this is most often achieved by using a *real-time operating system* (RTOS) based on *pre-emptive priority-based scheduling* [59].

Laplante furthermore notes that systems that must meet explicit response time constraints to avoid failure (as in his definition) are sometimes called *hard real-time systems*. Conversely, systems where failing to meet response time constraint results in degraded performance, but not outright failure, are called *soft real-time systems*. Again, it can be argued that this is the case for all computer systems, as some bound on response times must be set for acceptable performance. The term *firm real-time systems* is sometimes used to distinguish those systems with absolute response time constraints where some low probability of failing to meet constraints is acceptable.

The IEEE's glossary of software engineering terminology (IEEE Std 610.12-1990) [60] defines the term real-time as follows:

Pertaining a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor or respond in a timely manner to the external process.

An interesting aspect of this definition is that it focuses on a system's purpose of controlling or monitoring an external process. Thus, the meaning of the term "timely manner" depends on the external process and the system's purpose and need not be generally defined more precisely. A reasonable assumption is that the term "external process" is not used with a human user (interacting with the system through a keyboard, mouse, and monitor) in mind, but rather some equipment whose timing is determined by the laws of physics.

This glossary also provides a definition of an *embedded computer system*:

A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.

Thus, this definition states that an embedded system is part of a larger system and, apart from that, relies on a couple of examples to convey the meaning of the term. Li and Yao [61] note that no single comprehensive definition of the term exists, but still manage the following, which is somewhat more informative than that of the glossary:

Embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function. The word embedded reflects the fact that these systems are usually an integral part of a larger system, known as the embedding system.

This definition includes the additional information that software is tightly coupled with hardware and that both are designed to perform a dedicated function. Laplante has a similar view of an embedded software system as being completely encapsulated by the hardware it controls. This is contrasted with an *organic system*, not highly dependant on the hardware on which it runs. A *semi-detached system* is a software system that displays characteristics of both embedded and organic systems. Since an embedded system is designed to perform a dedicated function, an RTOS intended for such systems should support tailoring to different hardware configurations [62].

2.3.2 Industrial Control Systems

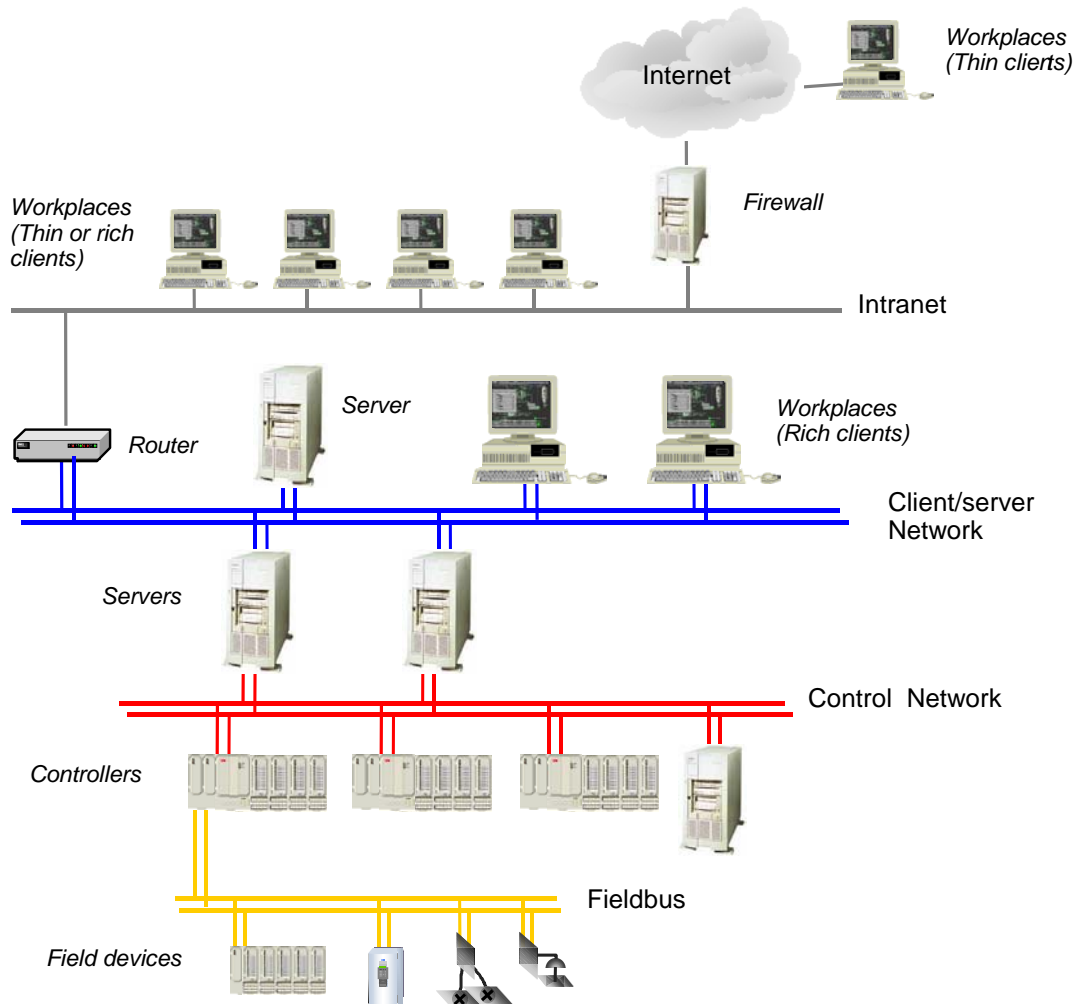


Figure 2-8 Typical configuration of industrial information and control systems

A specific application domain within the broader domain of embedded real-time systems is *industrial control systems*, which are computer systems that control physical processes and equipment. More specifically, they are used in the control of industrial plants. In practically all cases, these are distributed systems in which control functions are performed by several nodes that communicate via different types of networks. Typically, these nodes also communicate with other computer systems, such as different types of servers and workstations. The controllers and field devices are furthermore connected to physi-

cal processes and equipment to be controlled. Figure 2-8 illustrates a typical configuration of interconnected information processing and control nodes in an industrial system. Industrial control systems is given particular attention in this dissertation, as the development of such a system is the topic of the industrial cases study and the example systems used in the other empirical studies are also taken from this domain.

This system comprises different types of computers and other devices that communicate over different networks. The client/server network is used for communication between servers and between servers and workplaces. In some cases, a computer may be used as both a server and a workplace. The network may be connected to a corporate intranet via a router and further to the Internet via a firewall. The control network connects servers and controllers. In small systems, the control and client/server networks may be combined in one physical network. Different types of fieldbuses are used to interconnect field devices and to connect them to the rest of the system, either via controllers as the figure shows or directly to servers. In some cases, fieldbuses and the control network may share the same physical medium.

It is customary to divide the functionality of this kind of systems into different levels, where the functions of each level depend on those of the lower levels.

- The *workplace level* comprises different types of user interaction. A typical example is the software used by operators in control rooms to view and possibly alter the state of the controlled processes. This level also includes applications for such task as analysis of process data and configuration of process equipment. Applications usually run on PCs or other types of workstations, which may be attached to the client/server network, an intranet, or the Internet.
- A central function of the *server level* is to collect and store process data, which is used by different types of applications. These are typically client-server applications where data presentation is implemented on the workplace level and the majority of computation and storage on the server level. In addition, data and commands, possibly originating in the Workplace level, may be sent to process equipment. The server level may also include functions, such as optimization, that determine long-term control strategies. Servers that provide this functionality are connected to the client/server network and possibly the control network.

- The main function of the *control level* is the execution of control software by dedicated controllers. Typically, these repeatedly read values from sensors and computes values to be written to actuators, thus implementing sampled *control loops*, as discussed further at the end of this section. Control applications may be much more complex, however, for instance including sophisticated communication with other devices. Controllers are attached to the control network and possibly to fieldbuses.
- The *field level* comprises functions performed by different types of field devices. The simplest of these are I/O modules, which perform translations between physical signals and controller data. There may also be more advanced devices, such as smart sensors and actuators, which may be connected to a controller or directly to a server. Field devices often communicate over fieldbuses.

These levels are defined from the premise that the functions within each may require the presence of functions at lower levels but should be able to operate independently of higher-level functions. In addition, the functions within each level share characteristics that affect (among other things) the design of the software that implements them. One example is the different real-time and performance requirements. The control and field levels are dominated by hard and soft real-time deadlines, which often mandate the use of real-time operating systems. Often, the nodes on these levels are based on standards for programmable controllers, such as IEC 61131 [63], instead of relying on applications built directly on top of the operating system. To ensure availability, redundant hardware architectures may be used, in which the actual control of the process is performed by a primary processor, with additional processors working in stand-by mode and able to take over in case the primary processor fails.

Although the functions in the server layer may also be subject to response-time requirements, they tend to be dominated by a desire to maximize average throughput. Thus, they are usually implemented on top of general-purpose operating systems, such as Windows or Unix, and other platform products, such as database management systems. This furthermore makes the use of component technologies, such as COM+ and EJB, a realistic possibility. Redundancy may also be employed at this level, typically in the form of server groups. Unlike in the redundant architectures used at the lower levels, the servers in a group usually perform load balancing. Thus, if one server fails, the system will continue to operate with reduced performance. The user interface

functions of the workplace level are usually not subject to real-time requirements. They are often implemented using graphical design tools and possibly such technologies as ActiveX controls and JavaBeans.

Another characterizing feature of the levels is the difference in product life cycles. As a general rule, hardware and software components at lower levels are updated less often than at higher levels. According to experiences from ABB, applications have a life span of 3–5 years at the workplace level, 5–8 years at the server level, 8–15 years at the controller level, and 10–20 years at the field level. One result of this is that applications at one level are often required to work with legacy applications at lower levels, but less often at higher levels. For instance, new releases of client applications at the workplace level typically need to work with existing server software, while it is more common for new releases of server software also to require updated client applications. On the other hand, new server releases are usually required to support legacy hardware and software at the control and field levels. This difference in life cycles is in part motivated by the unidirectional dependence between the levels, which means that updates at one level is likely to disturb functions at all higher levels. Thus, in general, upgrades at lower levels entail more widespread disturbances and associated costs. Another factor that tends to make product updates more costly at, in particular, the control and field levels, is the possible need of disrupting the controlled process.

As already mentioned, applications at the workplace and server levels are often organized as client-server applications, where the server level is responsible for any communication with controllers and field devices. To simplify the implementation of client applications that can work with equipment from different vendors, a COM-based standard called *OLE for Process Control* (OPC) [64] has been created. OPC defines a set of COM interfaces for supporting basic data access as well as such functionality as alarm and event handling, historic data access, batch processing, etc. Many vendors of process equipment now provide OPC servers that implement (a subset of) these interfaces, which client applications can access using DCOM. The OPC standard is managed by an industry association called the OPC Foundation, which has over 400 member organizations and lists more than 140 manufacturers of OPC-compliant products. A standard that can be used for communication between servers and controllers is the *Manufacturing Message Specification* (MMS) [65], which specifies services suitable for such applications as data exchange and download of control software. There are also standards for communication between controllers, such as IEC 61131-5 [66]. As for the field level, a number

of fieldbuses have been standardized [67], some of which are particularly popular within certain industry sectors or geographical areas. A strong current trend is the increased popularity of fieldbuses based on standard network technologies, such as TCP/IP and Ethernet.

A very common function in industrial control systems, especially in the control level, is sampled *control loops* [68], which are also found in many other control systems, e.g. in vehicles, household appliances, and medical equipment. Control applications can be categorized into *continuous*, *discrete*, and *hybrid* control. In the first category, a controller samples continuous signals at regular intervals and computes streams of data to produce approximations of continuous output signals. An example application is the control of a valve to keep the flow of a fluid constant in the presence of varying supply pressure. In the second category, the controller reacts to discrete events and affects discrete actions. For instance, a controller could detect the level of fluid in a tank reaching minimum or maximum levels, and turn the supply on or off accordingly. Hybrid control applications combine both the other two types of control.

Continuous control applications can further be divided into *closed-loop control* and *open-loop control*. In the case where a single output of a physical process is being controlled using closed-loop control, the controller measures this output, called the *controlled variable*, and compares it with the desired value, the *reference*. Based on the difference, an input signal to the process, called the *manipulated variable*, is produced to drive the output in the desired direction. In this way, the controller can make the process output track a variable reference, or keep it constant in the presence of external disturbances. Figure 2-9 illustrates the principle, which is also known as *feedback control*. For simplicity, sensors and actuators are not shown, but taken to be part of the controller.

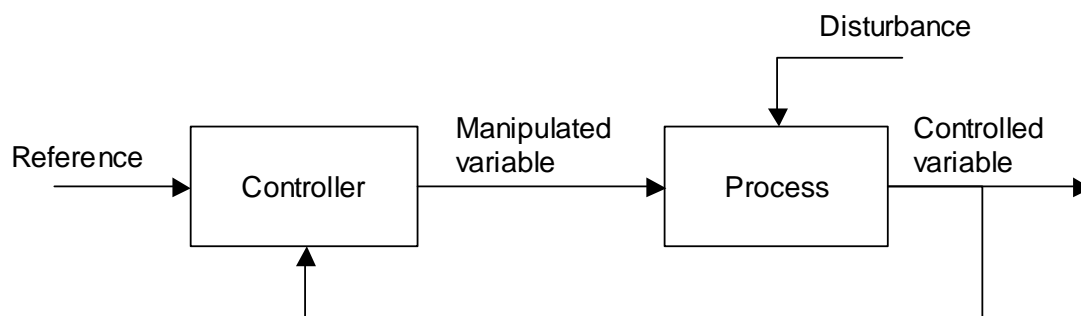


Figure 2-9 Closed-loop control system

In some cases, it may be advantageous to use the principle of open-loop control, also called *feed-forward control*. As illustrated in Figure 2-10, the controller measures the disturbance and sets the manipulated variable so as to keep the process output equal to the reference. This requires that the process is well understood so that the combined effect of the measured disturbance and the computed input can be accurately predicted. In addition to pure closed-loop and open-loop applications, there are applications where both the disturbance and the process output are measured. Also, there are *multi-variable control* applications in which multiple process variables are measured and controlled.

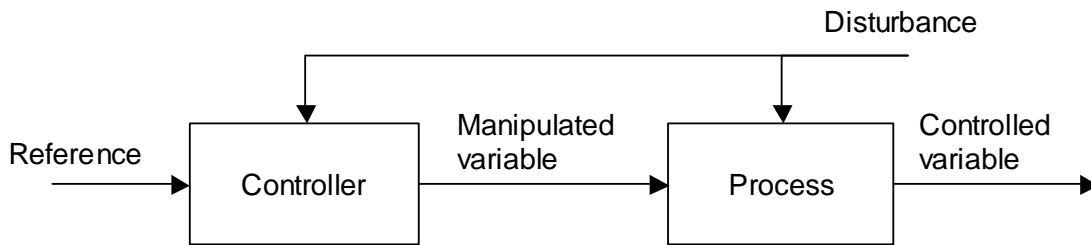


Figure 2-10 Open-loop control system

In the purest form of discrete control, the controller is only equipped with digital (i.e. binary) inputs and outputs, and the control software can be viewed as emulating digital electronic circuits. This has been utilized in graphical programming tools. Figure 2-11 shows a simple example of such a program in which the output “Run” becomes true when the input “Start” becomes true, and then stays true until the input “Stop” becomes true. The block marked “ ≥ 1 ” is a logical or-gate and the block marked “&” is a logical and-gate with its lower input inverted.

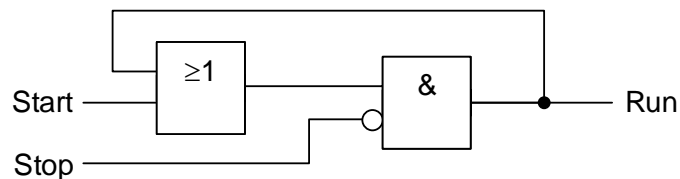


Figure 2-11 Example logic for discrete control system

In continuous control loops, the process is usually modeled as a system of differential equations, with the inputs and outputs being functions of time. Often,

the controls software is also implemented so as to approximate a system of differential equations. In such systems it is essential that the frequency with which the controller reads input signals and updates output signals, the *sampling frequency*, is sufficiently high to ensure faithful approximation of the control equations. This translates into a response time constraint on the computations the controller performs at each sample. In a programmable controller, the programmer should be able to set the sampling frequency, and this frequency should be guaranteed with some accuracy. This leads to hard real-time requirements for the controller product. In discrete and hybrid control, real-time requirements also occur to ensure the timing of actions in relation to events.

A particularly common type of closed-loop control is called *proportional-integral-differential* (PID) control. In this simple type of control, the manipulated variable m is computed from the error variable e – i.e. the difference between the current and desired value – as in the formula below.

$$m(t) = K_p e(t) + K_I \int_0^t e(\tau) d\tau + K_D \frac{de(t)}{dt}$$

The code below shows how this can be approximated in software. The code is written in the *Structured Text* language, which is one of the programming languages defined in the IEC 61131-3 standard [69].

```
Error := Reference - Input;
Sum := Sum + Error;
Output = Kp*Error + Ki*Sum*T + Kd*(Error - Err_old)/T;
Err_old := Error;
```

Traditionally, controller products have been designed for either continuous or discrete control. Two important categories of programmable controller products have been *distributed control systems* (DCSs) for continuous control and *programmable logic controllers* (PLCs) for discrete control. In the past, PLCs usually only supported simple computations on digital data, and the costs for these were considerably lower than for DCSs, which were required to perform at least numerical computations. However, the dramatic reduction in the price of computing hardware has resulted in both more sophisticated PLCs and less expensive DCSs, trends that have led to a convergence of these product categories into a single category of products often called *programmable controllers*. Such products still vary considerably in e.g. price and functionality, though.

2.3.3 Software Components in Embedded Real-Time Systems

As already noted, software components and component models are not widely used in embedded real-time systems and the dominant research direction is the definition of new component models for this domain. Typical characteristics of such models include source code components, static system configuration, and relatively narrow application domains. An example is the Koala component model for consumer electronics software, which was briefly described above. Other examples of such component models include *PECOS* [70] for field devices and *SaveCCM* [71] for vehicle control systems. Presumably, models are designed to exhibit such characteristics in order to ensure that systems can meet their requirements with respect to timing predictability, resource usage, and other important quality attributes in the domains targeted by the specific component models.

Möller and others have attempted to capture the requirements for component models to be used for embedded real-time systems [72]. They base their work primarily on interviews with senior technical staff from different companies. The work is slanted towards safety-critical software for heavy vehicles. Based on the interviews, the authors formulate technical requirements and development requirements. In addition, they present derived requirements, based on perceived implicit information from the interviews. The technical requirements are summarized in the list below.

- *Analyzable*. As the participating companies strive for better analysis of system behaviors, it is desirable for a component model to support such analysis. Provided that each component is tested and deemed functionally correct, the main analysis issues are related to composition and extra-functional properties, including timing.
- *Testable and debuggable*. Testing and debugging must be possible and supported by tools. It is desirable that components are tested in isolation before being integrated in the system.
- *Portable*. Components and supporting infrastructure should be as platform independent as possible, such that they can be ported to different operating systems and hardware with minimal effort. Ideally, components should even be as independent as possible from the infrastructure.
- *Resource constrained*. As the affected products are sensitive to computing hardware costs, the resource usage of components and infrastructure

should be minimized. Ideally, the use of components should not result in any run-time overhead.

- *Component modeling.* The component model should be based on a standard modeling language, such as UML. Developing new modeling techniques is not considered economically feasible.
- *Computational model.* Components should be passive, i.e. not contain their own threads of execution. A computational model where components are allocated to threads during composition is desirable.

Next, the authors present a set of requirements related to the development process, which are briefly described in the list below.

- *Introducible.* To manage costs and risks, it should be possible to adopt the component model gradually
- *Reusable.* It must be possible to reuse components in other systems than that for which they were originally developed. Ideally, it should even be possible to reuse components in systems based on different platforms.
- *Maintainable.* Components should be easy to change and maintain without breaking existing systems. Tool support is desirable, e.g. to support versioning.
- *Understandable.* To minimize effort and increase quality, the component model and systems based on it should be easy to understand. The implementation of error prone functions should desirably be supported by tools, e.g. utilizing automatic code generation.

Based on the above requirements, which were explicitly expressed in the interviews, the authors have synthesized two derived requirements, which are summarized below.

- *Source code components.* To allow white-box testing, source code components are preferable to binaries. The aim is not to modify components, so a glass-box approach may be sufficient, although it may be desirable to perform compile-time optimization to reduce resource usage.
- *Static configuration.* In the interest of analyzability, testability, limited resource usage, and understandability, compile-time configuration of systems is preferable to run-time configuration.

The derived requirements, along with the focus on the relatively narrow application domain of safety-critical software for heavy vehicles, place this work within the main research direction identified above. As already noted, this dissertation explores an alternative to this direction, by investigating the possibilities of using a model based on binary components and using COM as the starting point. In the following, the technical and development requirements listed above are compared to the characteristics of binary components and COM.

Analyzability is not a characteristic of COM components or binary executable software in general. To be able to analyze the behavior of systems built from COM components, suitable models describing the behavior of each component is required in addition to the components themselves. A component supplier might be responsible for providing such models and guaranteeing that the executable component complies with the models. Testing and debugging of COM components and component-based systems has strong tool support, most notably in Visual Studio. If component source code is available, debugging of systems can be performed in a white box fashion. COM does not prescribe a programming language for implementing components, and the portability of components is largely dependent on how they are programmed. The COM run-time system is probably fairly simple to port to different platforms, due to its relative simplicity. As demonstrated in later chapters of this dissertation, the use of COM results in some time and memory overheads but these are very modest, as is the size of the COM run-time system. COM defines its own language for defining interfaces, and translation between this language and UML is straightforward. UML is also suitable for specifying systems based on COM components. In fact, parts of UML are based on notations first used in connection with COM. A COM component is basically passive, as its methods must be invoked by a client for any code in the component to be executed. There is, however, no rule that prevents a component's methods from creating new threads, thereby making the component active.

As demonstrated in the industrial case study in Chapter 5, COM is well suited for gradual adoption. This is, in fact, a major reason that the model is adopted as part of the evolutionary approach of this dissertation. COM supports reusability through the separation of interfaces and implementation. There are no advanced facilities, such as parameterization, to enhance reusability further, however. But as COM gives component developers a high degree of freedom with respect to how components are implemented, it is quite possible to achieve increased reusability through recompilations, possibly using pro-

programming language mechanisms for e.g. parameterization. The maintainability of a COM component depends largely on how it is programmed. COM defines a standard policy of versioning, and observing this policy reduces the risk that a component change breaks existing systems. COM is sometimes conceived as complicated by developers not familiar with the model. Its widespread use in the desktop and information system domains, however, means that there are a high number of developers who understand the model well. COM has support for automatic handling of synchronization through apartments and communication through DCOM. The approach proposed in this dissertation introduces more flexible support for synchronization as well as services to support other functions, such as timing and execution control.

As the above discussion shows, the use of binary software components for embedded real-time systems is not seriously at odds with the requirements expressed by industrial participants in an interview study. Although the main research direction is still source code components and static configuration, some approaches based on binary components have recently been proposed. One of these is the *Robocup* component model [73], which is based on Koala and also targets the consumer electronics domain. The primary goal of this model is to combine the robustness and reliability of models like Koala with the flexibility of models like COM, especially with respect to run-time upgrades. A Robocup component, which is also called a *component package*, consists of a number of optional *models*, each of which may be human-oriented or machine-oriented. An example of a human-oriented model is documentation and examples of machine-oriented models include simulation models, interface models, and executable models. A component's executable model is also called an *executable component* and corresponds to a software component as defined in Section 2.2.1. In this way, Robocup distinguishes the units of trading - component packages - from the units of deployment - executable components. Since this dissertation is concerned with software components in the latter sense, the following discusses Robocup executable components in some more detail.

A Robocup executable component is similar to a COM component in many ways. It provides a set of *services*, each providing and possibly requiring a set of named *interfaces*. Such a service plays the same role as an object in COM and are subject to instantiation, while interfaces are identical to those of COM. Differences are that the interfaces of a service are named - allowing one service to provide and require multiple copies of interfaces of the same type - and that a service can only invoke the services of other components and the run-time sys-

tem through explicitly required interfaces. The named interfaces are similar to *ports* in UML 2.0 and the terms *interface* and *port* are used as synonyms in Robocup. Executable components can be specified in the *Robocup Interface Definition Language* (RIDL). Below is an example RIDL specification of a component similar to the COM component used in the example application in Chapters 6 and 7. For brevity, the definition of the interfaces *IActuator* and *ISensor* are left out as well as most of the definition of *IController*. These interfaces should contain essentially the same operations as in those chapters. The ellipses within brackets are placeholders for the specification of GUIDs.

```
interface IController {...} {
    void set_DesiredValue(in double value);
    void Start();
    ...
};

service SController {...} {
    provides {
        IController controller;
    };
    requires {
        IActuator actuator;
        ISensor sensor;
    };
};

service SPIDController {...} {
    complies SController;
    ...
};

component CPIDController {...} {
    provides SPIDController
};
```

The example illustrates how components are specified by referring to the services they provide, which are in turn specified by referring to the interfaces they provide and require and assigning names to these. Unlike in COM, services are not specified inside component specifications, and a service can be provided by several different components. In addition to the explicitly specified services, all components implement a special service termed the *service*

manager, and all service provide an interface derived from *rcIService* that can be used to retrieve pointers to a service's provided interfaces and supply a service with pointers for its required interfaces, called *binding*.

The example also illustrates the use of a "complies-with" relationship in the specification of the service called *SPIDController*. The meaning of the relationship is that *SPIDController* provides at least the same interfaces as *SController* and requires the same interfaces as *SController*. Thus, any code written to work with *SController* will also work with *SPIDController*. This is an example of *syntactic substitutability* as described in Chapter 3. The ellipsis in the *SPIDController* specification is a placeholder for the remaining parts of the service specification. It should include the same provided and required interface as for *SController* and may include additional provided interfaces. It may *not* include any required interfaces that are not also required by *SController*. In general, an entity can be substituted for another also if it requires fewer interfaces. In Robocup, however, the complies-with relation requires the sets of required interfaces to be identical, since a service's *scIService* interface contains one binding operation for each required interface. Thus, if a service required fewer interfaces than another service, the *scIService* interface of the former would provide fewer operations than that of the latter and substitution would not be possible. As the complies-with relation is explicitly specified, however, it would probably be quite easy to allow the compliant service to have fewer required interfaces and provide an *scIService* interface with null-operations for binding of the "lacking" interfaces.

Robocup also specifies a component model implementation called the *Robocup Runtime Environment (RRE)*, which is similar to the COM library in some ways. For example, a primary function of the RRE is to support instantiation of services, through a mechanism similar to that of COM. Upon request of a service instance, the RRE looks up the component providing the service and its physical location in the *RRE Registry*. It then loads the component if necessary and retrieves its service manager, which creates the requested service instance. If the service is provided by multiple components, the RRE is free to choose which component to use for instantiation. If a registered service complies with the requested service, the RRE can instead provide an instance of the former. A primary goal of Robocup is to provide the possibilities for run-time component upgrades lacking in Koala. This is achieved by the *Robocup Download Framework*. This framework consists of five *roles*, which may run on the target device or other nodes. The *initiator* role initiates and coordinates the download process in response to an external event. To verify the presence of all entities

involved in the process, it communicates with a *locator*. The locator locates a *repository* containing the component to be downloaded, the *target* of the download, and a *decider*. The role of the decider is to determine whether it is possible to download the component to the target, which is done by matching profiles obtained from the repository and the target. If the decider confirms that the download can be performed, the initiator either informs the repository, which *pushes* the component to the target, or the target, which *pulls* the component from the repository. In either case, the target completes the process by registering the component.

Since its initial development, the Robocup model has been extended by the projects *Space4U* [74] and *Trust4All* [75]. While the former focuses on off-line prediction of run-time properties of Robocup system, based on scenarios and simulations, the latter introduces run-time mechanisms resembling the approach to software components services presented in this dissertation. The topic of the Trust4All project is fault management, which is achieved by interception of invocations of operations on Robocup interfaces, which also includes invocations of system operations by components. The interception is performed by software entities called *middlemen*, which are inserted when components are bound to each other or to the run-time systems. These middlemen, which correspond to proxy-objects in COM+ and the approach presented later in this thesis, are generated at bind-time (i.e. at run-time) by an extension of the RRE called the *Middleman Generator* (MG). The function of a Trust4All middleman is to first detect failures by comparing the invocations it intercepts with a behavior model for the component in question. If an invocation is not considered a failure, it is forwarded to the proper receiver. When a failure is detected, the middleman tries to diagnose the failure, i.e. determine the fault that caused it, by the application of *symptom rules*. Next, it attempts to identify the best action to repair the fault by applying a set of *repair rules*. When the best action has been found it is taken, which can be done in several different ways. For instance, the component can be informed about the suspected fault and try to repair it itself. Alternatively, the middleman may retry failed invocations or restart the component in question. A key concept of Trust4All is that pre-specified symptom and repair rules cannot be expected to be optimal. Therefore, the run-time system monitors the success rates of diagnostics and repair actions, as well as measured costs of repair actions. The rules are then updated to maximize the expected success rate and minimize the expected costs for future diagnostics and repair actions.

The Robocup component model is similar to the approach proposed in this dissertation, since it uses binary components and is inspired by COM. Its concept of services, however, should not be confused with the concept of software component services described in Chapter 6 of this dissertation. On the other hand, the extensions provided by the Trust4All project are similar to the latter concept, since they use interception of operation invocations to augment the functionality of software components. The main difference between Trust4All middlemen and the proxies of the approach proposed in this dissertation is that middlemen are generated at run-time while the proxies are generated and compiled off-line. Still, the fault management concept of Trust4All could very well have been implemented as a software component service in the latter approach. The fact that proxies are generated off-line would not prevent them from using a set of symptom and repair rules that could be dynamically updated.

2.4 References

- [1] E. W. Dijkstra, "The Structure of the 'THE'-Multiprogramming System." In *Communications of the ACM*, volume 11, issue 5, 1968.
- [2] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules." In *Communications of the ACM*, volume 15, issue 12, 1972.
- [3] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture." In *ACM SIGSOFT Software Engineering Notes*, volume 17, issue 4, 1992.
- [4] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd edition. Addison-Wesley, 2003.
- [6] Institute of Electrical and Electronics Engineers, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Std. 1471-2000, 2000.
- [7] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.
- [8] J. Bosch, *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

- [9] C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*. Addison-Wesley, 1999.
- [10] P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd edition. Addison-Wesley, 2003.
- [11] O.-J. Dahl and K. Nygaard, "SIMULA - An ALGOL-Based Simulation Language." In *Communications of the ACM*, volume 9, issue 9, 1966.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissidies, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.
- [14] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, 2002.
- [15] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts, *Software Cost Estimation with COCOMO II*. Prentice-Hall, 2000.
- [16] PO. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-Level Modifiability Analysis (ALMA)." In *Journal of Systems and Software*, volume 69, issues 1-2, 2002.
- [17] M. Svahnberg, *Supporting Software Architecture Evolution*. PhD Thesis, Blekinge Institute of Technology, 2003.
- [18] R. Land, *Software System In-House Integration*. PhD Thesis, Mälardalen University, 2006.
- [19] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages." In *IEEE Transactions on Software Engineering*, volume 26, issue 1, 2000.
- [20] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [21] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd edition. Addison-Wesley, 2005.
- [22] H.-E. Eriksson, M. Penker, B. Lyons, and D. Fado, *UML 2 Toolkit*. Wiley, 2003.

- [23] P. Kruchten, "The 4+1 View Model of Architecture." In *IEEE Software*, volume 12, issue 6, 1995.
- [24] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edition. Addison-Wesley, 2002.
- [25] G. T. Heineman and W. T. Councill (editors), *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [26] I. Crnkovic and M. Larson (editors), *Building Reliable Component-Based Software Systems*. Artech House Books, 2001.
- [27] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard." In *IEEE Software*, volume 12, issue 6, 1995.
- [28] D. Box, *Essential COM*. Addison-Wesley, 1997.
- [29] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals*, 4th edition. Microsoft Press, 2004.
- [30] J. Murray, *Inside Microsoft Windows CE*, Microsoft Press, 1998.
- [31] C. Wehner, *Tornado and VxWorks*. Books on Demand, 2006.
- [32] F. E. Redmond III, *DCOM: Microsoft Distributed Component Object Model*. Wiley, 1997.
- [33] D. Chappell, *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [34] R. Englander, *Developing Java Beans*. O'Reilly, 1997.
- [35] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java™ Language Specification*, 3rd edition. Addison-Wesley, 2005.
- [36] T. Lindholm and F. Yelling, *The Java™ Virtual Machine Specification*, 2nd edition. Addison-Wesley, 1999.
- [37] D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.
- [38] D. S. Platt, *Introducing Microsoft .NET*, 3rd edition. Microsoft Press, 2003.
- [39] E. Cerami, *Web Services Essentials*. O'Reilly, 2002.
- [40] A. Wigley, S. Wheelwright, R. Burbidge, R. MacLoed, and M. Sutton, *Microsoft .NET Compact Framework*. Microsoft Press, 2003.

- [41] B. Burke and R. Monson-Haefel, *Enterprise JavaBeans 3.0*, 5th edition. O'Reilly, 2006.
- [42] I. Singh, B. Stearns, and M. Johnson, *Designing Enterprise Applications with the J2EE™ Platform*, 2nd edition. Addison-Wesley, 2002.
- [43] Object Management Group, *CORBA Component Model*, V4.0. OMG formal/2006-04-01, 2006.
- [44] J. Siegel, *CORBA 3 Fundamentals and Programming*, 2nd edition. Wiley, 2002.
- [45] A. Parsons and N. Randolph, *Professional Visual Studio 2005*. Wrox, 2006.
- [46] K. C. Wallnau, J. Stafford, S. A. Hissam, and M. Klein, "On the Relationship of Software Architecture to Software Component Technology." In *Proceedings of the 6th International Workshop on Component-Oriented Programming*, 2001.
- [47] S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau, "Enabling Predictable Assembly." In *Journal of Systems and Software*, volume 65, issue 3, 2003.
- [48] B. E. Rector and J. M. Newcomer, *Win32 Programming*. Addison-Wesley, 1997.
- [49] P. Narasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: Support for Real-Time Fault-Tolerant CORBA." In *Concurrency and Computation: Practice and Experience*, volume 17, issue 12, 2005.
- [50] R. Weinreich and J. Sametinger, "Component Models and Component Services: Concepts and Principles." In G. T. Heineman and W. T. Council (editors), *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [51] J. Löwy, *COM and .NET Component Services*. O'Reilly, 2001.
- [52] D. Sceppa, *Programming ADO*. Microsoft Press, 2000.
- [53] D. Sceppa, *Programming Microsoft ADO .NET 2.0 Core Reference*, 2005 edition. Microsoft Press, 2006.
- [54] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques." In *Proceedings of the Western Electronic Show and Convention*, 1970.

- [55] A. W. Brown and K. C. Wallnau, "Engineering of Component-Based Systems." In *Proceedings of the 2nd International Conference on Engineering of Complex Computer Systems*, 1996.
- [56] K. C. Wallnau, S. A. Hissam, and R. C. Seacord, *Building Systems from Commercial Components*. Addison-Wesley, 2001.
- [57] C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper (editors), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- [58] P. A. Laplante, *Real-Time Systems Design and Analysis: An Engineer's Handbook*, 2nd edition. IEEE Press, 1997.
- [59] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multi-programming in a Hard Real-Time Environment." In *Journal of the ACM*, volume 20, issue 1, 1973.
- [60] Institute of Electrical and Electronics Engineers, *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std. 610.12-1990, 1990.
- [61] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.
- [62] J. A. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems." In *IEEE Software*, volume 8, issue 3, 1991.
- [63] International Electrotechnical Commission, *Programmable Controllers – Part 1: General Information*, 2nd edition. International Standard IEC 61131-1, 2003.
- [64] F. Iwanitz and J. Lange, *OPC: Fundamentals, Implementation and Application*, 3rd edition. Hüthig Fachverlag, 2006.
- [65] ESPRIT Consortium CCE-CNMA (editors), *MMS: A Communication Language for Manufacturing*. Springer, 1995.
- [66] International Electrotechnical Commission, *Programmable Controllers – Part 5: Communications*. International Standard IEC 61131-5, 2000.
- [67] N. P. Mahalik (editor), *Fieldbus Technology: Industrial Network Standards for Real-Time Distributed Control*. Springer, 2003.
- [68] K. J. Åström and B. Wittenmark, *Computer Controlled Systems: Theory and Design*, 3rd edition. Prentice-Hall, 1996.

- [69] International Electrotechnical Commission, *Programmable Controllers – Part 3: Programming Languages*, 2nd edition. International Standard IEC 61131-3, 2003.
- [70] T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, and P. Müller, “Components for Embedded Software – The PECOS Approach.” In *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.
- [71] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren, “SaveCCM – A Component Model for Safety-Critical Real-Time Systems.” In *Proceedings of the 30th EROMICRO Conference*, 2004.
- [72] A. Möller, J. Fröberg, and M. Nolin, “Industrial Requirements on Component Technologies for Embedded Systems.” In *Proceedings of the 7th International Symposium on Component-Based Software Engineering*, 2004.
- [73] J. Muskens, M. R. V. Chaudron, and J. J. Lukkien, “A Component Framework for Consumer Electronics Middleware.” In C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper (editors), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- [74] E. Bondarev, J. Muskens, P. de With, M. Chaudron, and J. Lukkien, “Predicting Real-Time Properties of Component Assemblies: A Scenario-Simulation Approach.” In *Proceedings of the 30th EUROMICRO Conference*, 2004.
- [75] R. Su and M. R. V. Chaudron, “Self-adjusting Component-Based Fault Management.” In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2006.

Chapter 3

Specification of Software Components

with Kung-Kiu Lau and Shui-Ming Ho

3.1 Introduction

In its simplest form a software component contains some *code* (that can be executed on certain platforms) and an interface that provides (the only) access to the component. The code represents the *operations* that the component will perform when invoked. The interface tells the component-user everything he needs to know in order to deploy the component. Components can of course be deployed in many different contexts.

Ideally, components should be *black boxes*, to enable users to (re)use them without needing to know the details of their inner structure. In other words, the interface of a component should provide *all* the information needed by its users. Moreover, this information should be the *only* information they need. Consequently, the interface of a component should be the *only* point of access to the component. It should therefore contain all the information that users need to know about the component's *operations* (that is, what its code enables it to do) and its *context dependencies* (that is, how and where the component can be deployed). The code, on the other hand, should be completely inaccessible (and invisible), if a component is to be used as a black box.

The specification of a component is therefore the specification of its interface. This must consist of a precise definition of the component's operations and context dependencies and nothing else. Typically, the operations and context dependencies will contain the *parameters* of the component.

The specification of a component is useful to both component *users* and component *developers*. For users, the specification provides a definition of its interface, viz. its operations and context dependencies. Since it is only the interface that is visible to users, its specification must be precise and complete. For developers, the specification of a component also provides an abstract definition of its internal structure. Whilst this should be invisible to users, it is useful to developers (and maintainers), not least as documentation of the component.

In this chapter, we discuss the specification of software components. We will identify all the features that should be present in an idealized component, indicate how they should be specified, and show how they are specified using current component specification techniques.

3.2 Current Component Specification Techniques

The specifications of components used in practical software development today are mostly limited to what we will call syntactic specifications. This form of specification includes the specifications used with technologies such as Microsoft's Component Object Model (COM) [1], the Object Management Group's Common Object Request Broker Architecture (CORBA) [2], and Sun's JavaBeans [3]. The first two of these use different dialects of the Interface Definition Language (IDL) while the third uses the Java programming language to specify component interfaces. In this section, COM is mainly used to illustrate the concepts of syntactic specification of software components.

First, we take a closer look at the relationships between components and interfaces. A component provides the implementation of a set of named interfaces, or types, each interface being a set of named operations. Each operation has zero or more input and output parameters and a syntactic specification associates a type with each of these. Many notations also permit a return value to be associated with each operation, but for simplicity we do not distinguish between return values and output parameters. In some specification techniques it is also possible to specify that a component requires some interfaces, which must be implemented by other components. The interfaces provided and required by a component are often called the incoming and outgoing interfaces of the component, respectively.

Figure 3-1 is a UML class diagram [4] showing the concepts discussed above and the relationships between them. Note that instances of the classes shown

on the diagram will be entities such as components and interfaces, which can themselves be instantiated. The model is therefore a UML metamodel, which can be instantiated to produce other models. It is worth noting that this model allows an interface to be implemented by several different components, and an operation to be part of several different interfaces. This independence between interfaces and the components that implement them is an essential feature of most component specification techniques. The possibility of an operation being part of several interfaces is necessary to allow inheritance, or subtyping, between interfaces. The model also allows parameters to be simultaneously input and output parameters.

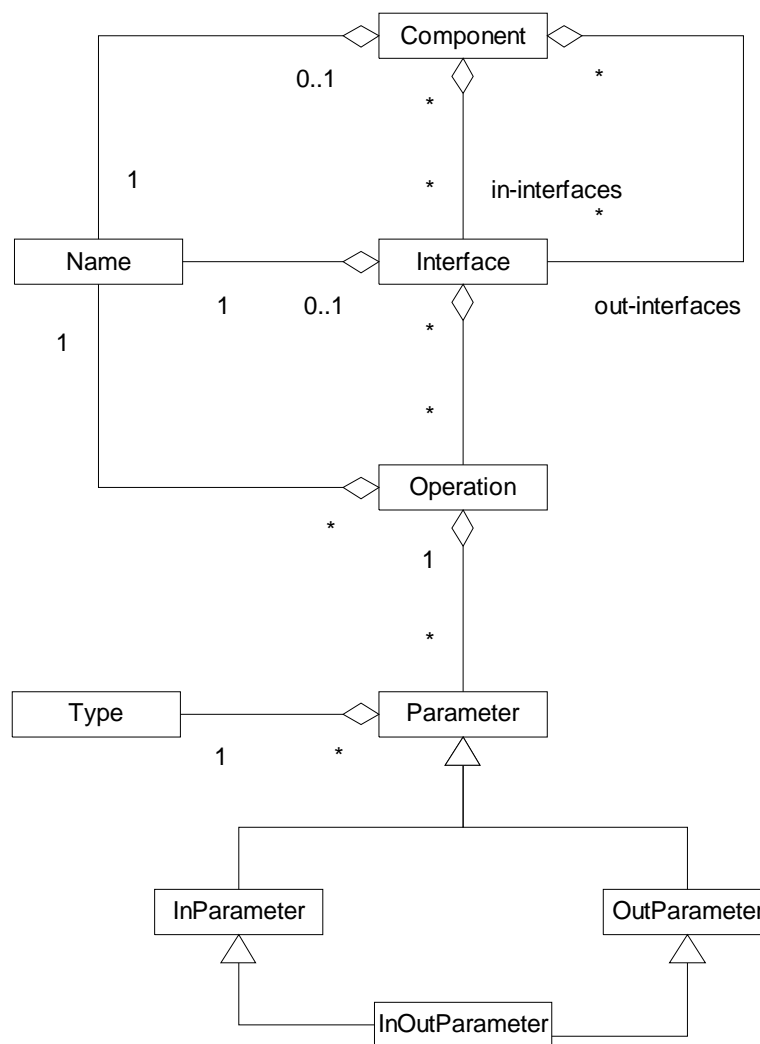


Figure 3-1 UML metamodel of the concepts used in syntactic specification of software components

The model presented above is intended to be a generic representation of the relationships between components, interfaces, and operations. In practice, these relationships vary between specification techniques. For example, one can distinguish between object-oriented specifications and what might be called procedural specifications. In this chapter we will only consider object-oriented specifications that are used by current technologies. This leads to no loss of generality, as procedural specification can be seen as a special case of object-oriented specification. There are subtle differences in the precise nature of the relationship between a component and its interfaces in different object-oriented specification techniques. In COM, for example, a component implements a set of classes, each of which implements a set of interfaces. The statement that a component implements a set of interfaces thus holds by association. In more traditional object-oriented specification techniques, a component is itself a class that has exactly one interface. The statement that a component implements a set of interfaces still holds, because this interface can include, or be a subtype of, several other interfaces.

As an example of a syntactic specification, we now consider the specification of a COM component. Below is a slight simplification of what might be the contents of an IDL file. First, two interfaces are specified, including a total of three operations which provide the functionality of a simple spell checker. Both interfaces inherit from the standard COM interface `IUnknown`. (All COM interfaces except `IUnknown` must inherit directly or indirectly from `IUnknown`. See [1] for more information about the particulars of COM.) All operations return a value of type `HRESULT`, which is commonly used in COM to indicate success or failure. A component is then specified (called a library in COM specifications), this implementing one COM class, which in turn implements the two interfaces previously specified. This component has no outgoing interfaces.

```
interface ISpellCheck : IUnknown
{
    HRESULT check(
        [in] BSTR *word, [out] bool *correct);
};

interface ICustomSpellCheck : IUnknown
{
    HRESULT add([in] BSTR *word);
    HRESULT remove([in] BSTR *word);
};
```

```
library SpellCheckerLib
{
  coclass SpellChecker
  {
    [default] interface ISpellCheck;
    interface ICustomSpellCheck;
  };
};
```

Relating this specification to the model above, there is one instance of `Component`, which is associated with two instances of `Interface`. Taking a closer look at the first interface, it is associated with a single instance of `Operation`, which is itself associated with one instance of `InParameter` and two instances of `OutParameter`, representing the two named parameters and the return value.

The information that can be obtained from a component specification such as the above is limited to what operations the component provides, and the number and types of their parameters. In particular, there is no information about the effect of invoking the operations, except for what might be guessed from names of operations and parameters. Thus, the primary uses of such specifications are type checking of client code and as a base for interoperability between independently developed components and applications. Different component technologies have different ways of ensuring such interoperability. For example, COM specifies the binary format of interfaces while CORBA defines a mapping from IDL to a number of programming languages.

An important aspect of interface specifications is how they relate to substitution and evolution of components. Evolution can be seen as a special case of substitution where a newer version of a component is substituted for an older version. Substituting a component Y for a component X is said to be safe if all systems that work with X will also work with Y . From a syntactic viewpoint, a component can safely be replaced if the new component implements at least the same interfaces as the older components, or, in traditional object-oriented terminology, if the interface of the new component is a subtype of the interface of the old component. For substitution to be safe however, there are also constraints on the way that the semantics of operations can be changed, as we shall see in the next section.

3.3 Specifying the Semantics of Components

While syntactic specifications of components are the only form of specifications in widespread use, it is widely acknowledged that semantic information about a component's operations is necessary to use the component effectively. Examples of such information are the combinations of parameter values an operation accepts, an operation's possible error codes, and constraints on the order in which operations are invoked. In fact, current component technologies assume that the user of a component is able to make use of such semantic information. For instance, COM dictates that the error codes produced by an operation are immutable, i.e. changing these is equivalent to changing the interface. These technologies do not, however, support the specification of such information. In the example with COM, there is no way to include information about an operation's possible error codes in the specification.

Several techniques for designing component-based systems that include semantic specifications are provided in the literature. In this section, we shall examine the specification technique presented in [53], which uses UML and the Object Constraint Language (OCL) [54] to write component specifications. OCL is included in the UML specification. Another well-known method that uses the same notations is Catalysis [55]. The concepts used for specification of components in these techniques can be seen as an extension of the generic model of syntactic specification presented in the previous section. Thus, a component implements a set of interfaces that each consists of a set of operations. In addition, a set of pre-conditions and post-conditions is associated with each operation. Pre-conditions are assertions that the component assumes to be fulfilled before an operation is invoked. Post-conditions are assertions that the component guarantees will hold just after an operation has been invoked, provided the operation's pre-conditions were true when it was invoked. In this form of specification, nothing is said about what happens if an operation is invoked while any of its pre-conditions are not fulfilled. Note that pre- and post-conditions is not a novel feature of component-based software development, and is used in a variety of software development techniques, such as the Vienna Development Method [56] and Design by Contract [57].

Naturally, an operation's pre- and post-conditions will often depend on state maintained by the component. Therefore, the notion of an interface is extended to include a model of that part of a component's state that may affect or be affected by the operations in the interface. Now, a pre-condition will in

general be a predicate over the operation’s input parameters and this state, while a post-condition is a predicate over both input and output parameters as well as the state just before the invocation and just after. Furthermore, a set of invariants may be associated with an interface. An invariant is a predicate over the interface’s state model that will always hold. Finally, the component specification may include a set of inter-interface conditions, which are predicates over the state models of all the component’s interfaces.

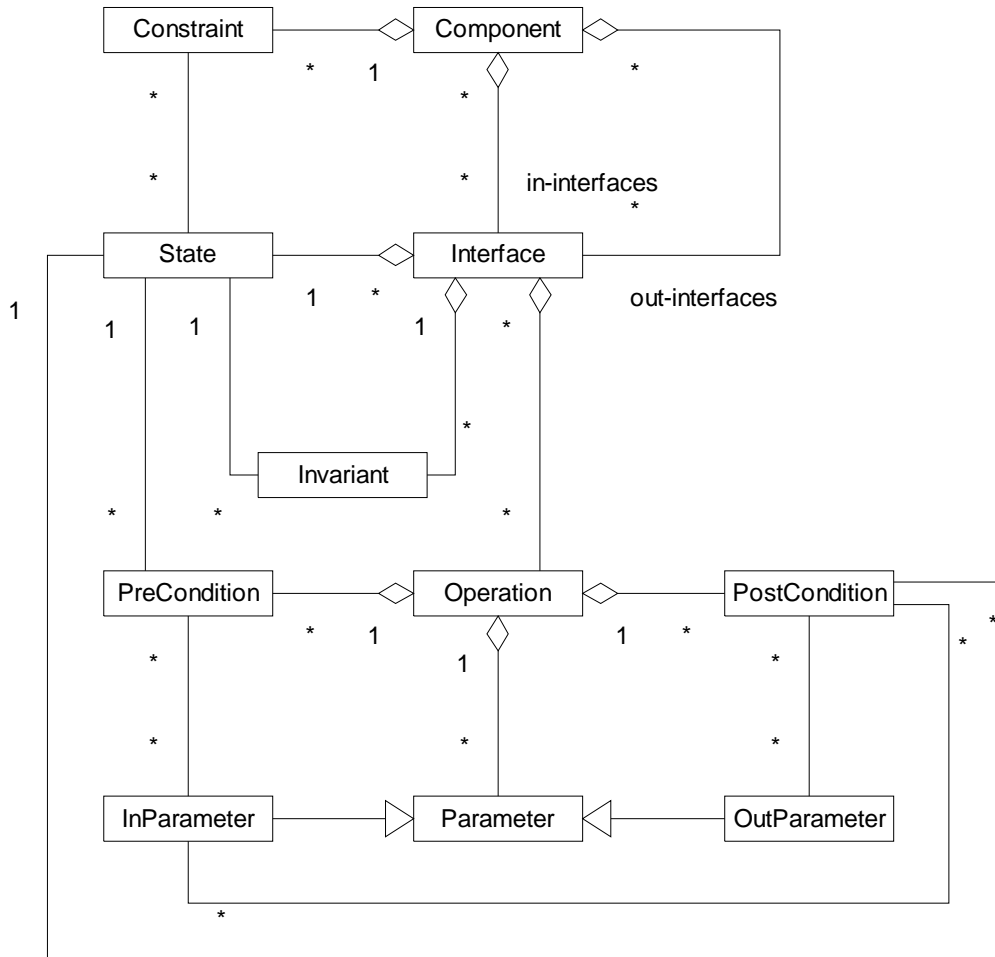


Figure 3-2 UML metamodel of the concepts used in semantic specification of software components

The concepts introduced here and the relationships among them are shown on the UML class diagram in Figure 3-2. For the sake of readability, the classes Name, Type, and InOutParameter are not shown, since they have no direct

relationships with the newly introduced classes. Note that this model allows the same state to be associated with several interfaces. Often, the state models of different interfaces of a component will overlap rather than be exactly the same. This relationship cannot be expressed in the model since we cannot make any assumptions about the structure of state models. Note also how each post-condition is associated with both input and output parameters and only one instance of `State`. The states before and after an invocation are represented by two separate instances of this single instance of (the metaclass) `State`.

In the model presented above, a partial model of the state of a component is associated with each interface, to allow the semantics of an interface's operations to be specified. It is important to note that this is not intended to specify how state should be represented within the component. While state models in component specifications should above all be kept simple, the actual representation used in the component's implementation will usually be subject to efficiency considerations, depend on the programming language, and so on. It is also worth mentioning that the above model is valid for procedural as well as object-oriented specification techniques.

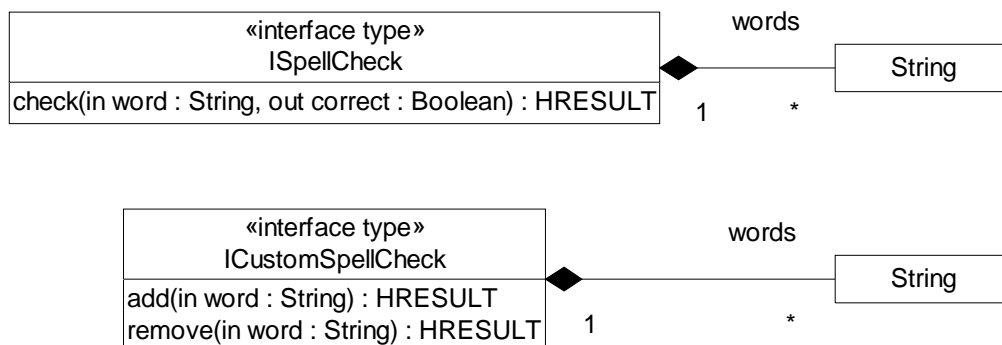


Figure 3-3 Example interface specification diagram

Before discussing the ramifications of this model any further, we now look at an example specification using the technique of [53]. Figure 3-3 is an example of an interface specification diagram. It shows the two interfaces introduced in the previous section as classes with the `<<interface type>>` stereotype. Thus, all the information in the syntactic interface specifications is included here. The state models of the interfaces are also shown. A state model generally takes the form of one or more classes having at least one composition re-

relationship with the interface the state belongs to. The special stereotype <<interface type>> is used instead of the standard <<interface>> since this would not allow the state models to be associated with the interfaces in this way.

The interface specification diagram is only a part of the complete interface specifications. The pre- and post-conditions that specify the semantics of the operations as well as any invariants on the state model is specified separately in OCL. Below is a specification of the three operations of the two interfaces above. There are no invariants on the state models in this example.

```

context ISpellCheck::check(
  in word : String, out correct : Boolean) : HRESULT
pre:
  word <> " "
post:
  SUCCEEDED(result) implies
    correct = words->includes(word)

context ICustomSpellCheck::add(
  in word : String) : HRESULT
pre:
  word <> " "
post:
  SUCCEEDED(result) implies
    words = words@pre->including(word)

context ICustomSpellCheck::remove(
  in word : String) : HRESULT
pre:
  word <> " "
post:
  SUCCEEDED(result) implies
    words = words@pre->exluding(word)

```

The pre-condition of the first operation states that if it is invoked with an input parameter that is not the empty string, the post-condition will hold when the operation returns. The post-condition states that if the return value indicates that the invocation was successful then the value of the output parameter is true if word was a member of the set of words and false otherwise. The specifications of the two last operations illustrate how post-conditions can refer to

the state before the invocation using the `@pre` suffix. This specification technique uses the convention that if a part of an interface's state is not mentioned in a post-condition, then that part of the state is unchanged by the operation. Thus, `words = words@pre` is an implicit post-condition of the first operation. All the specifications refer to an output parameter called `result`, which represents the return value of the operations. The function `SUCCEEDED` is used in COM to check whether a return value of type `HRESULT` indicates success or failure.

Similarly to interface specification diagrams, component specification diagrams are used to specify which interfaces components provide and require. Figure 3-4 is an example of such a diagram, specifying a component that provides the two interfaces specified above. The component is represented by a class with stereotype `<<comp spec>>` to emphasize that it represents a component specification. UML also has a standard component concept, which is commonly used to represent a file that contains the implementation of a set of concrete classes.

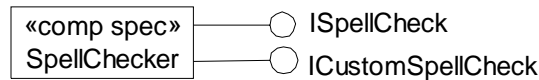


Figure 3-4 Example component specification diagram

The component specification is completed by the specification of its inter-interface constraints. The component in this example has one such constraint, specifying that the sets of words in the state models of the two interfaces must be the same. This constraint relates the operations of the separate interfaces to each other, such that invocations of `add` or `remove` affect subsequent invocations of `check`. The constraint is formulated in OCL below.

```

context SpellChecker
  ISpellCheck::words = ICustomSpellCheck::words
  
```

An important property of the model presented above is that state models and operation semantics are associated with interfaces rather than with a component. This means that the semantics is part of the interface specification. Consequently, a component cannot be said to implement an interface if it implements operations with the same signatures as the interface's operations but with different semantics. It should be noted that the terminology varies in the

literature on this point, as interfaces are sometimes seen as purely syntactic entities. In such cases, specifications that also include semantics are often called contracts. UML, for instance, defines an interface to be a class with only abstract operations, which can have no state associated with it.

While the main uses of syntactic specifications are type checking and ensuring interoperability, the utility of semantic specifications is potentially much larger. The most obvious use is perhaps tool support for component developers as well as developers of component-based application. For the benefit of component developers, one can imagine an automatic testing tool that verifies that all operations produce the correct post-conditions when their pre-conditions are satisfied. For this to work, the tool must be able to obtain information about a component's current state. A component could easily be equipped with special operations for this purpose, which would not need to be included in the final release. Similarly, for application developers, one can imagine a tool that generates assertions for checking that an operation's pre-conditions are satisfied before the operation is invoked. These assertions could either query a component about its current state, if this is possible, or maintain a state model of their own. The last technique requires that other clients do not affect the state maintained by a component, however, since the state model must be kept synchronized with the actual state. Such assertions would typically not be included in a final release, either.

With a notion of interface specification that includes semantics, the concept of substitution introduced in the previous section can now be extended to cover semantics. Clearly, if a component Y implements all the (semantically specified) interfaces implemented by another component X , then Y can be safely substituted for X . This condition is not necessary, however, for substitution to be safe. What is necessary is that a client that satisfies the pre-conditions specified for X will always satisfy the pre-conditions specified for Y , and that the client can rely on the post-conditions ensured by X also to be ensured by Y . This means that Y must implement operations with the same signatures as the operations of X , and with pre- and post-conditions that ensures the condition above. More specifically, if X implements an operation O , where $\text{pre}(O)$ is the conjunction of its pre-conditions and $\text{post}(O)$ the conjunction of its post-conditions, Y must implement an operation O' with the same signature such that $\text{pre}(O')$ implies $\text{pre}(O)$ and $\text{post}(O)$ implies $\text{post}(O')$. In other words, the interfaces implemented by Y can have weaker pre-conditions and stronger post-conditions than the interfaces implemented by X . It follows from this that the state models used for specifying the interfaces of X and Y need not be identi-

cal. This condition for semantically safe substitution of components is an application of Liskov’s principle of substitution [58].

Note that the above discussion is only valid for sequential systems. For multi-threaded components or components that are invoked by concurrently active clients, the concept of safe substitution must be extended as discussed in [59]. Finally, it must be noted that a client may still malfunction after a component substitution, even if the components fulfill semantic specifications that satisfy the above condition. This can happen, for instance, if the designers of the client and the new component have made conflicting assumptions about the overall architecture of the system. The term “architectural mismatch” has been coined to describe such situations [44].

The component specification diagram in Figure 3-4 shows how we can indicate which interfaces are offered by a component. In this example, we indicated that the spell checker offered the interfaces `ISpellCheck` and `ICustomSpellCheck` and used the constraint

```
ISpellCheck::words = ICustomSpellCheck::words
```

to specify that the interfaces act upon the same information model. We could, however, extend such diagrams to indicate the interfaces on which a component depends. This is illustrated in Figure 3-5.

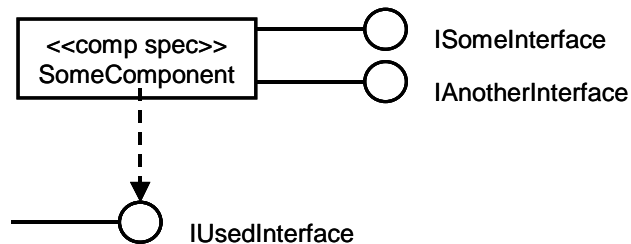


Figure 3-5 Component specification showing an interface dependency

We can also specify realization contracts using collaboration interaction diagrams. For example, in Figure 3-6 we state that whenever the operation *op1* is called, a component supporting this operation must in invoke the operation *op2* in some other component. Component specification diagrams and collaboration interaction diagrams may therefore be used to define behavioral dependencies.

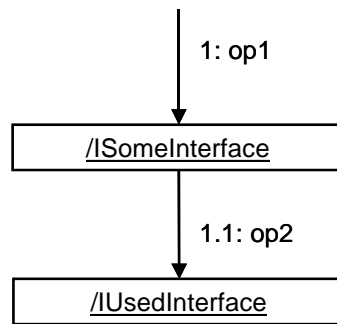


Figure 3-6 Collaboration interaction diagram

3.4 Specifying Extra-Functional Properties of Components

The specification of extra-functional properties of software components has recently become a subject of interest, mainly within the software architecture community. In [60], it is argued that the specification of architectural components is not properly addressed by conventional software doctrine. Architectural components are components of greater complexity than algorithms and data structures. Software components, as defined above, generally belong to this class. Conventional software doctrine is the view that software specifications must be *sufficient and complete* (say everything a user needs to know and is permitted to rely on about how to use the software), *static* (written once and frozen), and *homogeneous* (written in a single notation).

To use an architectural component successfully, information about more things than its functionality is required. This includes *structural* properties, governing how a component can be composed with other components; *extra-functional* properties, such as performance, capacity, and environmental assumptions; and *family* properties, specifying relations among similar or related components. It is not realistic to expect specifications to be complete with respect to all such properties, due to the great effort that would require, even if the developer of a component were able to anticipate all aspects of the component its users might care about. Often, this is even unrealistic in itself. Since we cannot expect software components to be delivered with specifications that are sufficient and complete, and since developers are likely to discover new kinds of dependencies as they attempt to use independently developed components together, specifications should be *extensible*. Specifications should also be *heterogeneous*, since the diversity of properties that might be of interest is unlikely to be suitably captured by a single notation.

The concept of credentials is proposed in [60] as a basis for specifications that satisfy the requirements outlined above. A credential is a triple $\langle \text{Attribute}, \text{Value}, \text{Credibility} \rangle$, where Attribute is a description of a property of a component, Value a measure of that property, and Credibility a description of how the measure has been obtained. A specification technique based on credentials must include a set of registered attributes, along with notations for specifying their value and credibility, and provisions for adding new attributes. A technique could specify some attributes as required and others as optional. The concept has been partially implemented in the architecture description language UniCon [61], which allows an extendable list of $\langle \text{Attribute}, \text{Value} \rangle$ pairs to be associated with a component. The self-describing components of Microsoft's new .NET platform [62] includes a concept of attributes in which a component developer can associate attribute values with a component and define new attributes by sub-classing an existing attribute class. Attributes are part of a component's metadata, which can be programmatically inspected, and is therefore suitable for use with automated development tools.

The concept of credentials has been incorporated in an approach to building systems from pre-existing components called Ensemble [63]. This approach focuses on the decisions that designers have to make, in particular when faced with a choice between competing technologies, competing products within a technology, or competing components within a product. In Ensemble, a set of credentials may be associated with a single technology, product, or component, or with a group of such elements. In addition, a variation of credentials is introduced to handle measures of properties that are needed but have not yet been obtained. These are called postulates and can be described as credentials where the credibility is replaced by a plan for obtaining the measure. The credential triple is thus extended with a flag `isPostulate`.

Returning our focus to the specification of single components, we now extend the ideas of Ensemble to allow a set of credentials to be associated with a component, an interface, or an operation. A UML metamodel with the concepts of syntactic specification augmented with credentials is shown in Figure 3-7. The class Name and the subclasses of Parameter have been omitted for brevity. Note that the concept of credentials is complementary to the specification of a component's functionality and completely orthogonal to the concepts introduced for semantic specifications. Since the specification of extra-functional properties of software components is still an open area of research, it would probably be premature to proclaim this as a generic model.

(and invisible). The specification of a component therefore must consist of a precise definition of the component's operations and context dependencies. In current practice, component specification techniques specify components only syntactically. The use of UML and OCL to specify components represents a step towards semantic specifications. Specification of extra-functional properties of components is still an open area of research, and it is uncertain what impact it will have on the future of software component specification.

3.6 Corrections to the Original Version

This chapter contains some corrections to the originally published version of the paper. These are all related to the UML metamodels of component specifications. In Figure 3-1, the multiplicities of `Component` and `Interface` in their association with `Name` have been changed from “1” to “1..0”. In Figure 3-2, the multiplicity of `State` in its association with `OutParameter` has been changed from “2” to “1” and the description of the figure in the text has been updated accordingly. Specifically, the text

Note also how each post-condition is associated with both input and output parameters and only one instance of `State`. The states before and after an invocation are represented by two separate instances of this single instance of (the metaclass) `State`.

on page 74 in this thesis replaces

Note also how each post-condition is associated with both input and output parameters and two instances of the state model, representing the state before and after an invocation.

of the original version. Finally, in Figure 3-7, the multiplicity of the three classes associated with `Credential` have been changed from “1” to “0..1”.

3.7 References

- [1] Microsoft Corporation, *The Component Object Model Specification, v0.99*, 1996.
- [2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG formal/00-10-01, 2000.

- [3] Sun Microsystems, *JavaBeans Specification, Version 1.01*, 1997.
- [4] Object Management Group, *Unified Modeling Language 1.3 Specification*. OMG formal/00-03-01, 2000.
- [5] J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.
- [6] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
- [7] D. D'Souza and A. C. Wills, *Objects, Components and Frameworks: The Catalysis Approach*. Addison-Wesley, 1998.
- [8] C. B. Jones, *Systematic Software Development using VDM*. Prentice-Hall, 1986.
- [9] B. Meyer, *Object-Oriented Software Construction*, 2nd edition. Prentice-Hall, 2000.
- [10] B. Liskov, "Data Abstraction and Hierarchy." In *Addendum to the Proceedings of OOPSLA '87*, 1987.
- [11] H. Schmidt and J. Chen, "Reasoning About Concurrent Objects." In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference*, 1995.
- [12] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard." In *IEEE Software*, volume 12, issue 6, 1995.
- [13] M. Shaw, "Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does." In *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.
- [14] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them." In *IEEE Transactions on Software Engineering*, volume 21, issue 4, 1995.
- [15] J. Conrad, P. Dengler, B. Francis, J. Glynn, B. Harvey, B. Hollis, R. Ramachandran, J. Schenken, S. Short, and C. Ullman, *Introducing .NET*. Wrox Press, 2000.
- [16] K. C. Wallnau and J. Stafford, "Ensembles: Abstractions for a New Class of Design Problem." In *Proceedings of the 27th Euromicro Conference*, 2001.
- [17] H. Schmidt and W. Zimmerman, "A Complexity Calculus for Object-Oriented Programs." In *Object-Oriented Systems*, volume 1, issue 2, 1994.

Chapter 4

Adopting a Software Component Model in Real-Time Systems Development

Abstract

Component-based software engineering (CBSE) and the use of (de-facto) standard component models have gained popularity in recent years, particularly in the development of desktop and server-side software. This paper presents a motivation for applying CBSE to real-time systems and discusses the consequences of adopting a software component model in the development of such systems. Specifically, the consequences of adopting Microsoft's COM, DCOM, and .NET models are analyzed. The most important aspects of these models are discussed in an incremental fashion. The analysis considers both real-time systems in general, and a real-life industrial control system where some aspects the COM model have been adopted. It is concluded that adopting these models makes it possible to meet real-time requirements, but that some overhead must be expected and that special precautions may have to be taken to prevent loss of real-time predictability.

4.1 Introduction

Component-based software engineering (CBSE) denotes the assembling of software products from pre-existing smaller products, generally called components. In particular when this is done using standard or de-facto standard component models and supporting technologies [1]. A component model generally defines a concept of components and rules for their design-time composition and/or run-time interaction, and is usually accompanied by one or more component technologies, implementing support for composition and/or inter-operation.

In recent years, the use of component models has gained popularity in the development of desktop and server-side software. Two popular models in desktop applications are Sun's *JavaBeans* [2] and Microsoft's *ActiveX controls* [3], where the latter is built on top of the more basic *Component Object Model* (COM) [4]. Both of these are particularly suited for components to be used with visual composition tools. The best-known models in the server domain are Sun's *Enterprise JavaBeans* (EJB) [5], Microsoft's COM extension *COM+* [6], and the Object Management Group's new *CORBA Component Model* (CCM) [6]. These models offer similar support for transactional processing and persistent data management.

This paper discusses the possibilities of using such component models in real-time systems. In particular, the feasibility of using COM, the most basic of these models, and its distributed extension is analyzed and illustrated through a case study. Microsoft's latest model *.NET* [8] is also briefly discussed. Section 4.2 presents motivations for adopting a component model, both in real-time systems generally and in a real-world industrial control system. Section 4.3 discusses the implications of adopting different aspects of a particular component model. An overview of related work is given in Section 4.4. Finally, Section 4.5 concludes the paper and outlines future work.

4.2 Motivation

The general motivation for component-based software engineering is the prospect of increased productivity and timeliness of software development projects. Indeed, this is as desirable for real-time and embedded software as for any other application. It could also be argued that some characteristics of CBSE make it particularly attractive for real-time systems. For instance, real-time software often requires more extensive testing, so the use of pre-tested components may be particularly time saving in the development of such system. Another example is that many embedded systems, such as mobile telephones, could benefit from reuse of components across products and models. Conversely, there are also barriers to CBSE particular to real-time and embedded systems. Most obviously, there may be a risk that component models and technologies may introduce unacceptable overhead or loss of predictability.

An example of a real-time system where the use of a component model has been useful is the industrial control system by ABB called *ControlIT* (<http://www.abb.com>). This product is a modular controller consisting of a

central processing unit with two expansion buses. One bus is for I/O modules of different types and is used to connect the controller to physical signals. The other bus is for communication interfaces and allows the controller to communicate with other devices using different media and protocols. The controller also has two built-in serial ports and redundant Ethernet ports.

ABB's development organization is globally distributed, and the interest in component models first arose from a wish to make it easier for different development centers to add I/O and communication support to the system. It was decided to redesign the system's architecture so that all code particular to a certain I/O module, communication interface, or protocol resides in a separate component called a protocol handler. To achieve this, rules and formats for interaction between these protocol handlers and the rest of the system had to be decided on. In other words, a component model was needed. In the following analysis of adopting different aspects of a component model, the usefulness and liabilities of each particular aspect in connection with protocol handlers will be discussed. The use of a component model to support integration of protocol handlers in ABB's control system is further described in [9], where it is demonstrated that the new architecture supports distributed development and reduces the time required to implement I/O and communication support.

4.3 Adopting Microsoft Models

Among the most commonly used component models for desktop and server applications are Microsoft's Component Object Model (COM) and its extension Distributed COM (DCOM) [10]. There is also great interest in the company's new generation of technologies, commonly denoted .NET, which also defines a component model [8]. This section explores the possibilities of using these models in real-time systems. The most important aspects of these models will be discussed in an incremental fashion, assuming that it may be desirable in some situations also to adopt the models in such a fashion.

4.3.1 COM Interfaces

A key principle of COM and other component models is that interfaces are specified separately from both the components that implement them and those that use them. COM defines a dialect of the *Interface Definition Language* (IDL)

that is used to specify object-oriented interfaces. Interfaces are object-oriented in the sense that their operations are to be implemented by a class and passed a reference to a particular instance of that class when invoked. The code that uses a component does not refer directly to any objects, however. Instead, the operations of an interface supported by an object are invoked via what is known as an *interface pointer*. A concept known as interface navigation makes it possible for the user to obtain a pointer to every interface supported by the object. For a further description of this topic, see e.g. [10].

COM also defines a run-time format for interface pointers. What an interface pointer really references is an *interface node*, which in turn, contains a pointer to a table of function pointers, called a *VTABLE*. Typically, the node also contains a pointer to an object's instance data, although this is up to the implementation (of the supporting component technology). This use of VTABLEs is identical to the way that many C++ compilers implement virtual functions. Thus, the time and space overhead associated with accessing an object through an interface pointer is the same as that incurred with virtual C++ functions. This time overhead is very modest. The memory overhead should also be acceptable, perhaps except for the most resource constrained embedded systems. Figure 4-1 illustrates the typical format of interface nodes.

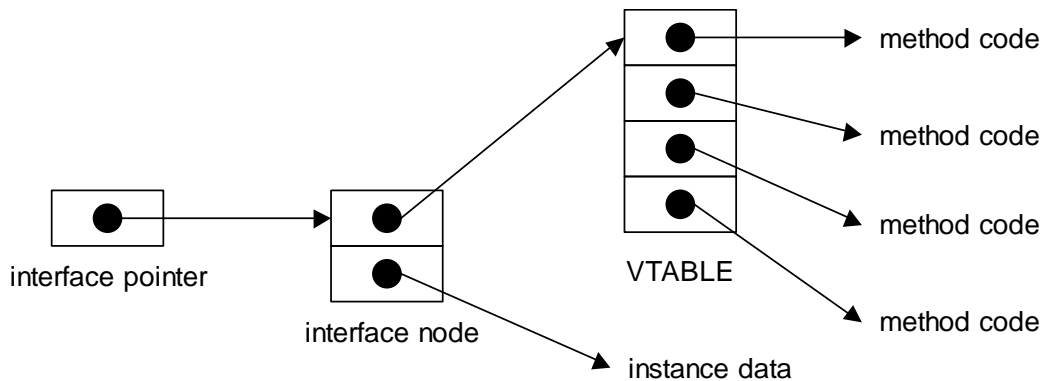


Figure 4-1 Typical format of COM interface nodes

For most real-time systems, a more serious concern than these modest overheads is that interface navigation introduces a possible source of run-time errors. If the user of a component asks an object for a pointer to an interface that the object does not support, this will not be detected during compilation. It may be argued, in fact, that this is the principal difference between interface

navigation and interface inheritance in traditional object-oriented programming. This can be seen as a necessary price to pay for the otherwise desirable reduced compile-time dependence between components.

Most real-time systems are based on multi-tasking and are often built on top of a real-time operating system (RTOS) using some kind of priority-based scheduling. Developers of components for real-time systems will generally need to make assumptions about how their components will be used in a multi-tasking environment. The safest option will be always to assume that an object can be concurrently used by several tasks, and guard all methods with the necessary synchronization. For reasons of efficiency, however, it may be more desirable to require the code that uses the component to provide any necessary synchronization. The exact circumstances under which such protection is necessary are thus an important part of the component's documentation.

The use of COM IDL to specify interfaces and VTABLEs to implement interface pointers work well for protocol handlers. The concept of multiple interfaces per object with interface navigation is useful since different protocol handlers must provide different functionality. The object-oriented nature of COM interfaces where each interface pointer refers to a particular instance of a class also matches the needs of the ABB control system. Multiple instances of the same protocol handler are useful, e.g. when a controller is equipped with two identical communication interfaces, linking it to two separate networks of the same type. The latest version of the control system uses COM interfaces, but not the other parts of COM discussed below.

4.3.2 Instantiation and Dynamic Linking

The previous section stated that the code of a COM component is implemented in classes, without discussing how instances are created. Also, nothing was said about how and when the code in different components is linked together. COM defines a policy for instantiation, which is intended to ensure that different components can be installed in a system at different times. When a component is installed, information about it must be registered somewhere in the system, linking the identity of its classes to the code that implement these. COM also requires a run-time library, called the COM library, to be installed on the system. When some code wants to use a component, it uses an operation provided by the COM library to ask for an instance of a class and an

initial interface pointer to it. If the code of the component is not already loaded into memory, the COM library uses the registered information to locate the code and load it before an instance is created. This process is illustrated in Figure 2.

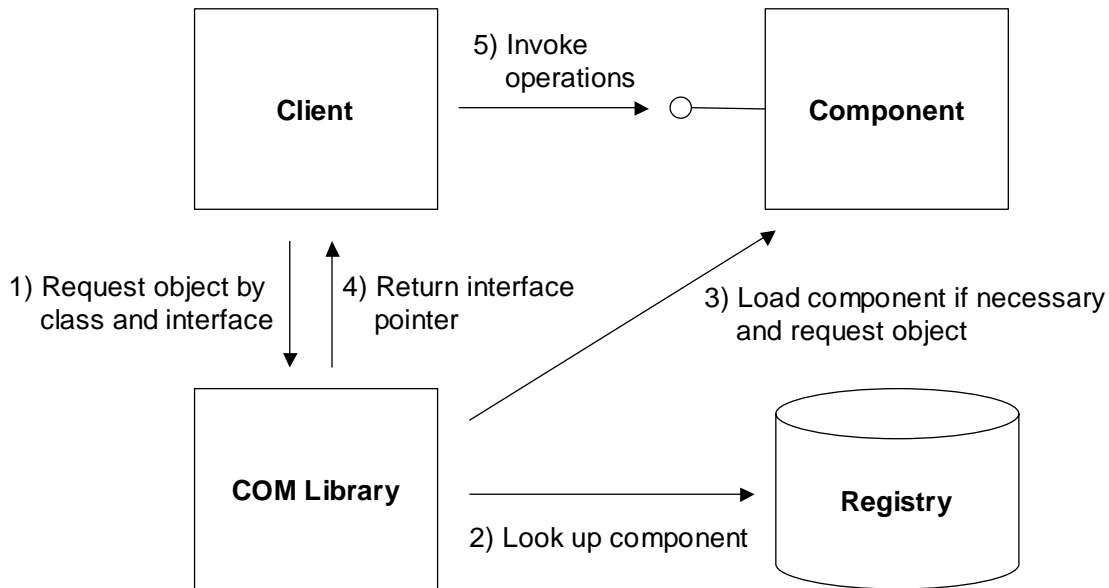


Figure 4-2 Instance creation and dynamic loading of code in COM

Thus, creation of an instance involves searching the information about registered classes and possibly loading of code. This leads to a noticeable overhead when compared to instantiation in for instance C++. Furthermore, this overhead will vary, depending on whether the code implementing a class has already been loaded or not. This variability can be eliminated, however, by designing the software such that all components that may be used will be loaded at start-up. Note that removal of instances is subject to the same variability, since the COM standard states that code can be unloaded when the last instance that rely on it is removed.

A benefit that follows from COM's way of creating instances is that the code that implements a component can be built independently of any code that uses the component. Since instantiation involves passing the identity of the desired class as a parameter to a system operation, it is a possible source of run-time errors, which is not present during instantiation in traditional object-oriented programming, since attempting to instantiate a class that does not exist will

result in a compilation error in this case. Again, this is a necessary price to be paid for decreased coupling.

COM's principle of instantiation is well suited for creating instances of protocol handlers, since no knowledge of the set of available protocol handlers should be built into the system. The overhead associated with looking up classes and dynamic loading of code is expected to be tolerable, especially since the software is designed such that protocol handlers need only be instantiated and deleted during program download. Thus, the extra time taken by this way of instantiation will not interfere with the continuous operation of the system. An additional benefit of using this technique for instantiation is that protocol handlers can be deployed (and updated) independently of the rest of the system. Future versions of the control system may include a COM library and employ dynamic linking of components. It is possible that a commercial component technology, such as WindRiver's implementation of COM for the VxWorks RTOS (<http://www.windriver.com>) will be used.

4.3.3 Location Transparency with DCOM

DCOM is an extension of COM, which allows component-based applications to be distributed across memory spaces or physical machines. This is realized using auxiliary objects known as proxies and stubs. When some code asks the COM library to create an instance of a class that is implemented in a component in a different location, the instance is created in the remote location along with a stub. The code that asked for the instance is passed an interface pointer to a proxy object, created on its side. When an operation is invoked via this interface pointer, the proxy translates this to a remote procedure call (RPC) to the remote stub, which in turn invokes the corresponding operation on the real object. It may also be necessary to create a proxy-stub pair at other times than object instantiation. This happens when an interface pointer is passed as a parameter to an operation of an object in a remote location. This process is known as marshalling. Proxy and stub code is usually generated automatically from IDL specifications. Figure 3 illustrates the use of proxy and stub objects

The ability to deal with memory spaces may not be of great consequence to real-time systems, since real-time operating systems do not traditionally use memory spaces. The ability to deal with such may, however, be useful in parallel processor architectures. DCOM may be useful in simplifying the implementation of distributed real-time systems. The transparency to the program-

mer of accessing remote objects is not completely valid for real-time systems, however. Since the timing of object operations will differ between local and remote invocations, real-time software developers will still need to consider whether their code uses components in another location or not. It is also useful for developers of components to be aware of whether their components will be remotely accessed. For instance, one may consider exploiting the ability to define asynchronous interfaces for such components. An additional benefit of using DCOM in real-time systems is that it may simplify the implementation of communication between these systems and COM-based desktop applications, such as operator stations.

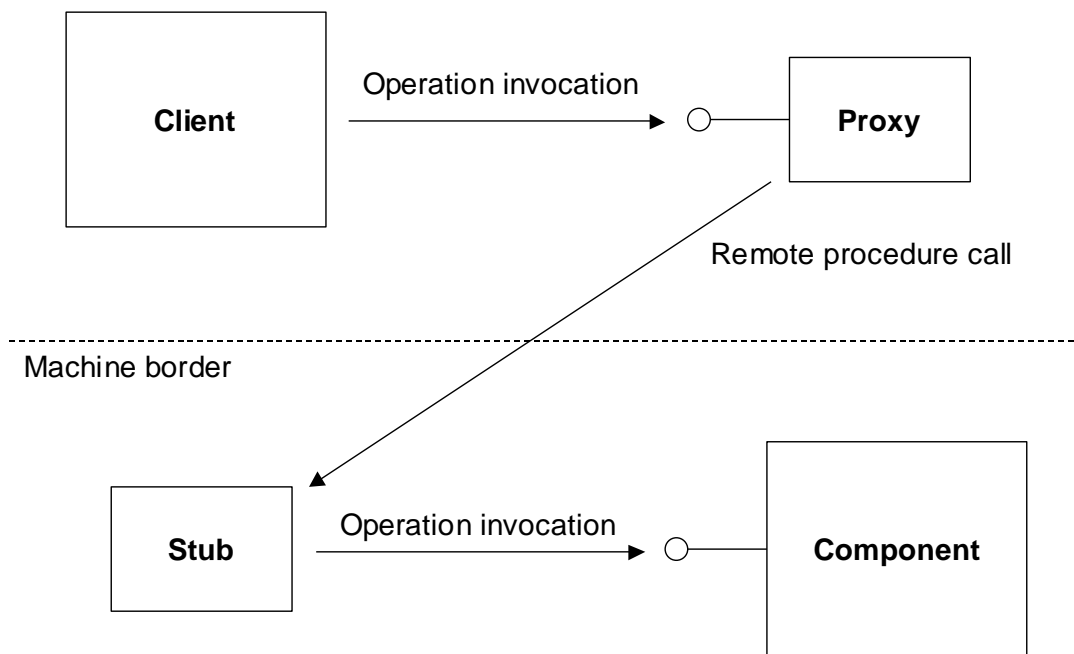


Figure 4-3 Use of proxy and stub objects in DCOM

In addition to the extra time overhead associated with remote invocation and marshalling, DCOM also requires more space than COM, to store the proxy and stub code as well as the RPC mechanism. The proxy and stub are generally quite small and executes relatively quickly, however, so the time and space overhead is mostly due to the RPC mechanism and underlying protocol stack. Therefore, using DCOM does not result in much of an overhead for distributed real-time systems, where RPC or some other communication mechanism would be needed anyway.

A possible reason for using DCOM in ABB's control system, is that protocol handlers could be located in the communication interfaces or I/O modules they support, rather than in the central processing unit. The usefulness of this is not obvious, however, especially when considering the required additional overhead. Thus, there are no current plans to adopt DCOM in the system.

4.3.4 The Next Generation: .NET

The name .NET is used by Microsoft to denote a comprehensive set of new technologies. This includes a new component model, intended to replace COM/DCOM. A notable development is that .NET moves the responsibility of providing certain functionality from the components to a more sophisticated run-time system. In particular, COM/DCOM requires components to provide a considerable amount of "house-keeping" functionality that is taken care of by the .NET run-time. Much of the flexibility that follows from having such implementations in each component is maintained under .NET, where the operation of the run-time system with respect to individual components can be affected by setting declarative attributes.

A potential advantage of this development is increased reliability, since it may be assumed that more effort may be invested in ensuring the quality of a run-time system to be re-used in a large number of systems. Another attractive consequence of having more code in a common run-time is that the total size of the software may decrease. Obviously, this advantage grows with the number of components in the system. On the other hand, using a sophisticated run-time system, possibly without using much of its functionality, may lead to unnecessarily large software. This is a particular problem for resource constrained embedded systems. Fortunately, Microsoft has defined a special compact version of .NET that limits this problem somewhat. What is assumed to be the greatest strength of .NET is the potential for increased development productivity. This relies both on the aforementioned run-time system with its associated libraries, and on advanced development tools. As usual, this gain is achieved at the expense of some run-time overhead. While it seems clear that this cost is acceptable for desktop software, the corresponding question for real-time systems is more open.

4.4 Related Work

There are some work on software component models and real-time or embedded systems in recent literature. This work is dominated by efforts to define component models particularly targeted at real-time embedded systems or even narrower application domains. Examples include Philip's Koala component model for consumer electronics [11], the component model for industrial field devices developed in the PECOS project [12], the commercial product ControllShell [13], which is based on visual composition and automatic code generation, and the more academic ACCORD approach [14] of aspect-oriented component-based development of real-time systems. Work on using "mainstream" component models in real-time systems is less common. One example is [15], which also discusses COM. This work, however, focuses on extensions to COM rather than the consequences of using the existing model in real-time systems.

4.5 Conclusion and Future Work

This paper has discussed the idea of using a software component model in real-time systems. In particular, using Microsoft models, both from the perspective of real-time systems in general and from that of ABB's control system. In general, it has been seen that each of the levels of adopting the models that have been discussed, introduces some degree of time and space overhead. In addition, new potential sources of run-time errors are introduced, corresponding to compilation errors in traditional object-oriented programming. It is concluded that COM/DCOM may be used for real-time systems, provided that any overhead is acceptable or can be compensated by hardware, and that the software designer takes care that the potential run-time errors are not allowed to materialize and result in a loss of predictability.

The major conclusions to be drawn from the discussions in this paper are as follows. COM interfaces, which provide a way to separate the specification of interfaces from component implementation, carry with them a very modest time and memory overhead. Compared to interface inheritance in object-oriented programming, COM interfaces introduce a potential source of run-time errors. COM's mechanism for instantiating objects and loading code at run-time has a considerable overhead when compared to instantiation in for example C++. This overhead is subject to a certain variability, which may be

avoided by careful application design. DCOM is an extension of COM that allows applications to access COM objects across memory spaces and physical machine boundaries. The time and space overhead associated with this is dominated by the underlying communication mechanisms. The new .NET platform promises increased development productivity, but it remains to be seen to what extent it is suitable for real-time systems.

The immediate future work planned as a continuation of this paper is to strengthen the analyses with empirical evidence by conducting experiments and collecting measurements. Preferably, this should be done on a real-time platform using a commercial or self-made COM implementation. In the longer perspective, an intriguing idea is to develop a COM-based component model particularly intended for real-time software. This idea is primarily inspired by how COM+ supports the implementation of functionality such as transactional processing, which is considered a major challenge in distributed information systems. Corresponding challenges for real-time systems include issues such as concurrency, synchronization, and timing. In addition to easing the implementation it would be desirable for such a model to support compositional reasoning, i.e. the deduction of a system's properties from the known properties of its parts. A natural starting point for achieving this is the existing work on prediction enabled component technologies (PECT).

4.6 References

- [1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
- [2] Sun Microsystems, *JavaBeans Specification, Version 1.01*, 1997.
- [3] D. Chappell, *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [4] Microsoft Corporation, *The Component Object Model Specification, v0.99*, 1996.
- [5] Sun Microsystems, *Enterprise JavaBeans Specification, Version 2.0*, 2001.
- [6] D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.
- [7] Object Management Group, *CORBA Components, Version 3.0, formal/02-06-65*, 2002.
- [8] J. Lowy, *Programming .NET Components*. O'Reilly, 2003.

- [9] F. Lüders, *Use of Component-Based Software Architectures in Industrial Control Systems*. Techn. Lic. thesis, Mälardalen University, 2003.
- [10] F. E. Redmond III, *DCOM: Microsoft Distributed Component Object Model*. Wiley, 1997.
- [11] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [12] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born, "A Component Model for Field Devices." In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment*, 2002.
- [13] S. A. Schneider, V. W. Chen, and G. Pardo-Castellote, "ControlShell: Component-Based Real-Time Programming." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1995.
- [14] A. Tešanović, D. Nyström, J. Hansson, and C. Norström, "Towards Aspectual Component-Based Development of Real-Time Systems." In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications*, 2003.
- [15] D. Chen, A. Mok, and M. Nixon, "Real-Time Support in COM." In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, 1999.
- [16] S. A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau, "Enabling Predictable Assembly." In *Journal of Systems and Software*, volume 65, issue 3, 2003.

Chapter 5

Adopting a Component-Based Software Architecture for an Industrial Control Systems – A Case Study

with Ivica Crnkovic and Per Runeson

Abstract

This chapter presents a case study from a global company developing a new generation of programmable controllers to replace several existing products. The system needs to incorporate support for a large number of I/O systems, network types, and communication protocols. To leverage its global development resources and the competency of different development centers, the company decided to adopt a component-based software architecture that allows I/O and communication functions to be realized by independently developed components. The architecture incorporates a subset of a standard component model. The process of re-designing the software architecture is presented, along with the experiences made during and after the project. An analysis of these experiences shows that the component-based architecture effectively supports distributed development and that the effort required for implementing certain functionality has been substantially reduced while, at the same time, the system's performance and other run-time quality attributes have been kept on a satisfactory level.

5.1 Introduction

Component-based software engineering (CBSE) denotes the disciplined practice of building software from pre-existing smaller products, generally called software components, in particular when this is done using standard or de-facto standard component models [7, 16]. The popularity of such models has increased greatly in the last decade, particularly in the development of desk-

top and server-side software, where the main expected benefits of CBSE are increased productivity and timeliness of software development projects.

The last decade has also seen an unprecedented interest in the topic of software architecture [2, 15] in the research community as well as among software practitioners. CBSE has notable implications on a system's software architecture, and an architecture that supports CBSE, e.g. by mandating the use of a component model, is often called a component-based software architecture.

This chapter presents an industrial case study from the global company ABB, which is a major supplier of industrial automation systems, including programmable controllers. The company's new family of controllers is intended to replace several existing products originally developed by different organizational units around the world, many of which were previously separate companies, targeting different, though partly overlapping, markets and industries. As a consequence, the new controller products must incorporate support for a large number of I/O systems, network types, and communication protocols. To leverage its global development resources and the competency of different development centers, ABB decided to adopt a component-based software architecture that allows I/O and communication functions to be realized by independently developed components.

This chapter is organized as follows. The remainder of this section describes the questions addressed by the case study and motivates the choice of method. Section 5.2 presents the context of the case study, including a description of the programmable controller and its I/O and communication functions as well as the organizational and business context. The process of componentizing the system's software architecture is presented in Section 5.3. Section 5.4 analyzes the results of the project and identifies some experiences of general interest. A brief overview of related work is provided in Section 5.5. Section 5.6 presents conclusions and some ideas for further work.

5.1.1 Questions Addressed by the Case Study

The general question addressed by the case study is what advantages and liabilities the use of a component-based software architecture entails for the development of an industrial control system. Due to the challenges of the industrial project studied, the potential benefit that a component-based architecture makes it easier to extend the functionality of the software has been singled out for investigation.

More specifically, the project allows the two following situations to be compared:

- The system has a monolithic software architecture and all functionality is implemented at a single development center.
- The system has a component-based software architecture and pre-specified functional extensions can be made by different development centers.

By pre-specified functional extensions we mean extensions in the form of components adhering to interfaces already specified as part of the architecture. This fact is presumed to be significant, while the fact that the functionality in question happens to be related to I/O and communication is not.

In addition to the question of whether the component-based architecture reduces the effort required to make such functional extension, the study also addresses the questions of whether any such reduction is sufficient to justify the effort invested in redesigning the architecture and after how many extensions the saved effort surpasses the invested effort. Since the system in question is subject to hard real-time requirements, the potential effect of the architecture on the possibility of satisfying such requirements is also studied. Finally, the architecture's possible effect on performance is analyzed.

5.1.2 Case Study Method

The research methodology used is a flexible design study, conducted as a participant observation case study [14]. The overall goal of the study is to observe the process of componentization, and evaluate the gains of a component-based architecture. It is not possible to demarcate such a complex study object in a fixed design study. Neither is there an option to isolate and thereby study alternative options. Instead we address the problem using a case study approach, where one study object is observed in detail and conclusions are drawn from this case.

In order to enable best possible access to the information on the events in the case, the observations are performed by an active participant. The main researcher is also an active practitioner during the study. As a complement, interviews are conducted after the case study to collect data on costs and gains of the component approach, thus conducting data triangulation.

Participatory research always includes a threat with respect to researcher bias. In order to increase the validity of the observations, a researcher was introduced late in the research process as a “critical friend”. The long researcher involvement in this case study reduces on the other hand the threat with respect to respondent bias.

Case studies are by definition weak with respect to generalization, in particular when only a single case is observed. However, to enable learning across organizational contexts, we present the context of the case study in some detail. Hence, the reader may find similarities and differences compared to their environment, and thus judge the transferability of the research.

5.2 Context of the Case Study

Following a series of mergers and acquisitions, ABB became the supplier of several independently developed programmable controllers for the process and manufacturing industries. The company subsequently decided to continue development of only a single family of controllers for these and related industries, and to base all individual controller products on a common software platform.

To be able to replace all the different existing products used in different regional areas and industry sectors, these controllers needed to incorporate support for a high number of communication protocols, network types, and I/O systems, including legacy systems from each of the previously existing controllers as well as current and emerging industry standards.

A major challenge in the development of the new controller platform was to leverage the software development resources at different development centers around the world and their expertise in different areas. In particular, it was desirable to enable different development centers to implement different types of I/O and communication support. Additional challenges were to make the new platform sufficiently general, flexible, and extendable to replace existing controllers, as well as to capture new markets.

The solution chosen to meet these challenges was to base the new platform on one of the existing systems while adopting a component-based software architecture with well-defined interfaces for interaction between the main part of

the software and I/O and communication components developed throughout the distributed organization.

As the starting point of the common controller software platform, one of the existing product lines was selected. This system is based on the IEC 61131-3 industry standard for programmable controllers [8]. The software has two main parts 1) the ABB Control Builder, which is a Windows application running on a standard PC, and 2) the system software of the ABB controller family, running on top of a real-time operating system (RTOS) on special-purpose hardware. The latter is also available as a Windows application, and is then called the ABB Soft Controller.

A representative member of the ABB controller family is the AC 800M modular controller. This controller has two built-in serial communication ports as well as redundant Ethernet ports. In addition, the controller has two expansion busses. One of these is used to connect different types of input and output modules through which the controller can be attached to sensors and actuators. The other expansion bus is used to connect communication interfaces for different types of networks and protocols. The picture in Figure 5-1 shows an AC 800M controller equipped with two communication interfaces (on the left) and one I/O module (on the right).



Figure 5-1 An AC 800M programmable controller

The Control Builder is used to specify the hardware configuration of a control system, comprising one or more controllers, and to write the programs that will execute on the controllers. The configuration and the control programs together constitute a control project. When a control project is downloaded to the control system the system software of the controllers is responsible for interpreting the configuration information and for scheduling and executing the control programs.

Figure 5-2 shows the Control Builder with a control project opened. The project consists of three structures, showing the libraries used by the control programs, the control programs themselves, and the hardware configuration, respectively. The latter structure is expanded to show a configuration of a single AC 800M controller, equipped with an analogue input module (AI810), a digital output module (DO810), and a communication interface (CI851) for the PROFIBUS-DP protocol [10].

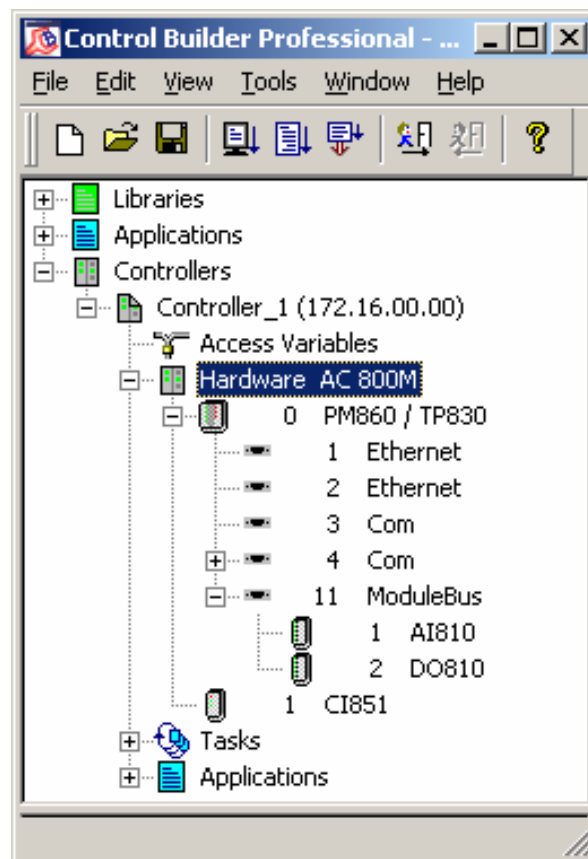


Figure 5-2 The ABB Control Builder

To be attractive in all parts of the world and a wide range of industry sectors, the common controller must incorporate support for a large number of I/O systems, communication interfaces, and communication protocols. During the normal operation of a controller, i.e. while the control programs are not being updated, there are two principal ways for it to communicate with its environment, denoted I/O (Input/Output) and variable communication, respectively.

To use I/O, variables of the control programs are connected to channels of input and output modules using the program editor of the Control Builder. For instance, a Boolean variable may be connected to a channel on a digital output module. When the program executes, the value of the variable is transferred to the output channel at the end of every execution cycle. Variables connected to input channels are set at the beginning of every execution cycle. Real-valued variables may be attached to analogue I/O modules. Figure 5-3 shows the program editor with a small program, declaring one input variable and one output variable. Notice that the I/O addresses specified for the two variables correspond to the position of the two I/O modules (AI810 and DO810, respectively) in Figure 5-2.

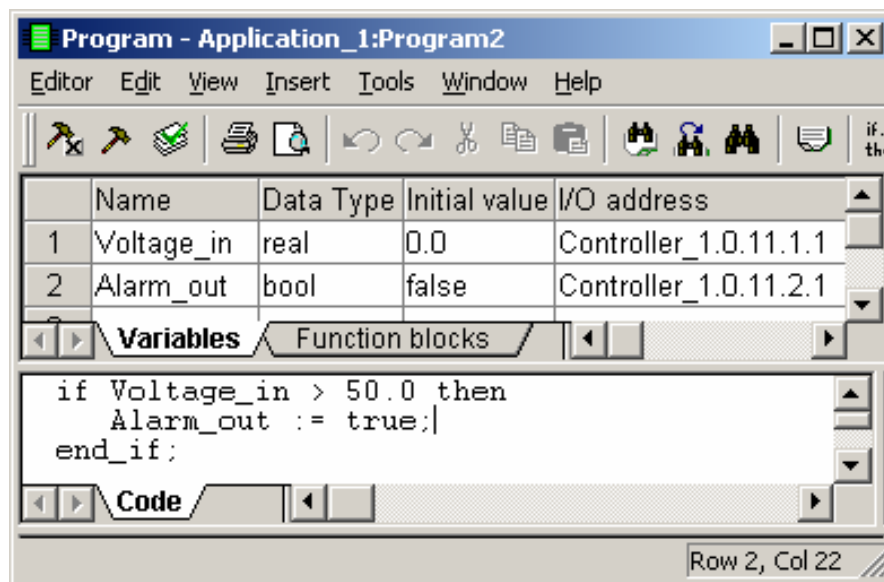


Figure 5-3 The program editor of the ABB Control Builder

Variable communication is a form of client/server communication and is not synchronized with the cyclic program execution in the way that I/O is. A server supports one of several possible protocols and has a set of named vari-

ables that may be read or written by clients that implement the same protocol. An ABB Controller can be made a server by connecting program variables to so-called access variables in a special section of the Control Builder (see Figure 5-2). Servers may also be other devices, such as field-bus devices [10].

A controller can act as a variable communication client by using special routines for connecting to a server and reading and writing variables via the connection. Such routines for a collection of protocols are available in the Communication Library, which is delivered with the Control Builder. The communication between a client and a server can take place over different physical media, which, in the case of the AC 800M, are accessed either via external communication interfaces or the built-in Ethernet or serial ports.

Control projects are usually downloaded to the controllers via a TCP/IP/Ethernet-based control network, which may optionally be redundant. A control project may also be downloaded to a single controller via a serial link. In both cases, downloading is based on the Manufacturing Message Specification (MMS) protocol [5], which also supports run time monitoring of hardware status and program execution. The system software of a controller, including the RTOS, can be updated from a PC via a serial link. Figure 5-4 shows an example of a control system configuration.

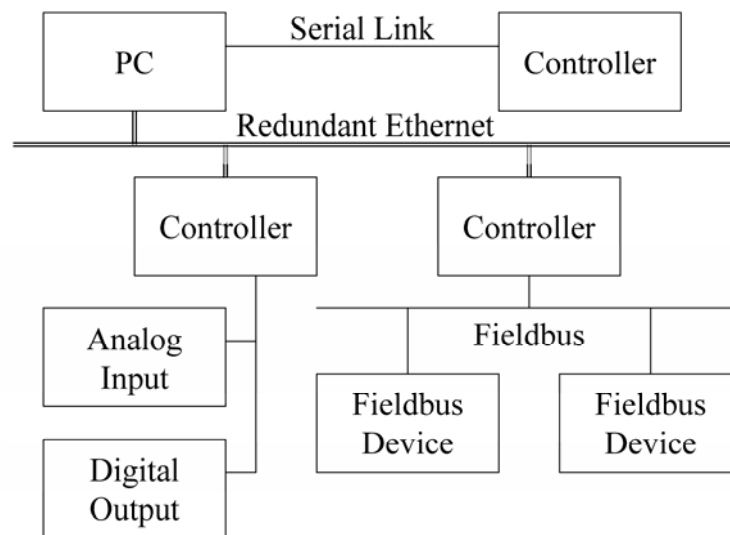


Figure 5-4 Example control system configuration

5.3 Componentization

5.3.1 Reverse Engineering of the Existing Software Architecture

The first step in the componentization of the architecture of the Control Builder and the controller system software was to get an overview of the existing architecture of the software, which was not explicitly described in any document. The software consists of a large number of source code modules, each of which is used to build the Control Builder or the controller system software or both, with an even larger number of interdependencies. An analysis of the software modules with particular focus on I/O and communication functions yielded the course-grained architecture depicted in Figure 5-5.

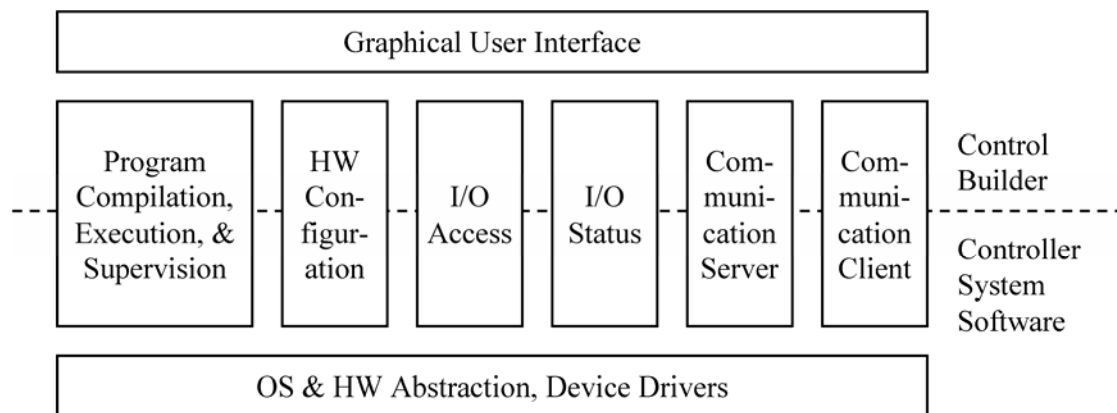


Figure 5-5 The original software architecture

The boxes in the figure represent logical components of related functionality. Each box is implemented by a number of modules, and is not readily visible in the source code. Many modules are also used as part of other products, which are not discussed further here. This architecture is thus a product-line architecture [3], although the company has not yet adopted a systematic product line approach. On the controller side, which is the focus of this chapter, the architecture has two distinct layers [15]. The lower layer (the box at the bottom of the figure) provides an interface to the upper layer (the rest of the boxes), which allows the source code of the upper layer to be used on different hardware platforms and operating systems. The complete set of interdependencies between modules within each layer was not captured by the analysis.

To illustrate how some modules are used to build both the Control Builder and the controller system software, we consider the handling of hardware configurations. The hardware configuration is specified in the Controllers structure of the Control Builder. For each controller in the system, it is specified what additional hardware, such as I/O modules and communication interfaces, it is equipped with. Further configuration information can be supplied for each piece of hardware, leading to a hierarchic organization of information, called the hardware configuration tree. The code that builds this tree in the Control Builder is also used in the controller system software to build the same tree there when the project is downloaded. If the configuration is modified in the Control Builder and downloaded again, only a description of what has changed in the tree is sent to the controller.

The main problem with this software architecture is related to the work required to add support for new I/O modules, communication interfaces, and protocols. For instance, adding support for a new I/O system possibly required source code updates in all the components except the User Interface and the Communication Server, while a new communication interface and protocol could require all components except I/O Access to be updated.

As an example of what type of modifications may have been needed to the software, we consider the incorporation of a new type of I/O module. To be able to include a device (I/O module or communication device) in a configuration, a hardware definition file for that type of device must be present on the computer running the Control Builder. For an I/O module, this file defines the number and types of input and output channels. The Control Builder uses this information to allow the module and its channels to be configured using a generic configuration editor. This explains why the user interface did not need to be updated to support a new I/O module. The hardware definition file also defines the memory layout of the module, so that the transmission of data between program variables and I/O channels can be implemented in a generic way.

For most I/O modules, however, the system is required to perform certain tasks, for instance when the configuration is compiled in the Control Builder or during start-up and shutdown in the controller. In the architecture described above, routines to handle such tasks had to be hard-coded for every type of I/O module supported. This required software developers with a thorough knowledge of the source code. The situation was similar when adding support for communication interfaces and protocols. The limited number

of such developers therefore constituted a bottleneck in the effort to keep the system open to the many I/O and communication systems found in industry.

5.3.2 Component-Based Software Architecture

To make it much easier to add support for new types of I/O and communication, it was decided to split the logical components mentioned above into their generic and specific parts. The generic parts, commonly called the generic I/O and communication framework, contains code that is shared by all hardware and protocols implementing certain functionality. Routines that are specific to a particular type of hardware or protocol are implemented in separate components, called protocol handlers, installed on the PC running the Control Builder and on the controllers. This component-based architecture is illustrated in Figure 5-6.

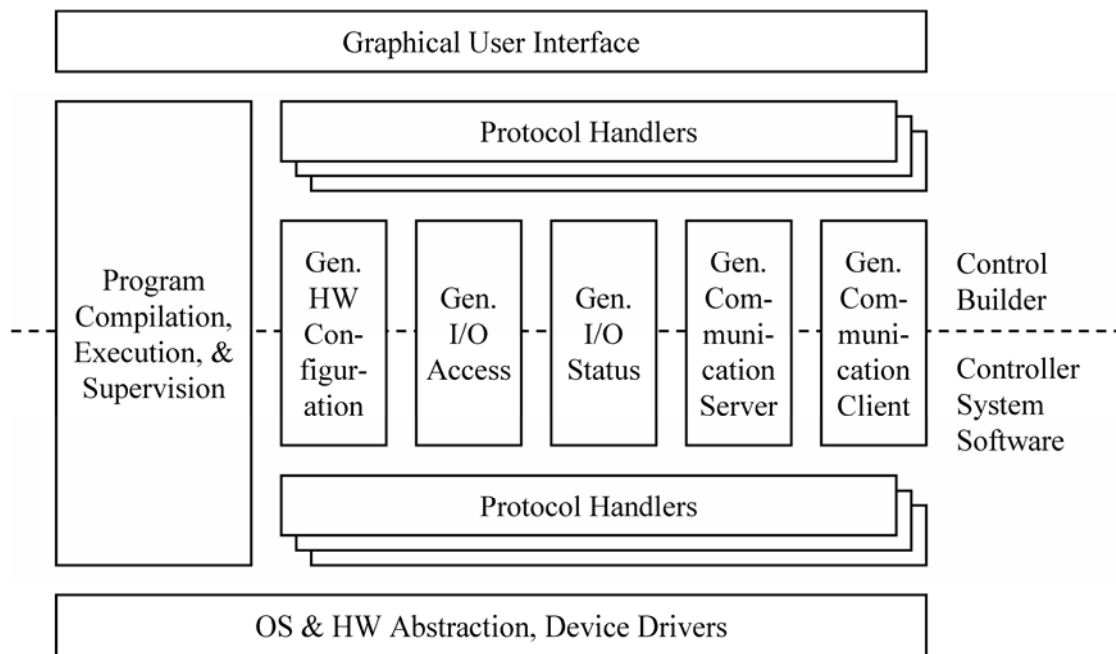


Figure 5-5 Component-based software architecture

Focusing again on the controller side, and comparing this architecture with the previous one, the protocol handlers can be seen as an additional half-layer between the framework and the bottom layer. To add support for a new I/O module, communication interface, or protocol in this architecture, it is only

necessary to add protocol handlers for the PC and the controller along with a hardware definition file and possibly a device driver. The format of hardware definition files is extended to include the identities of the protocol handlers as described below.

Essential to the success of the approach, is that the dependencies between the framework and the protocol handlers are fairly limited and, even more importantly, well specified. One common way of dealing with such dependencies is to specify the interfaces provided and required by each component [9]. The new control system uses the Component Object Model (COM) [4] to specify these interfaces, since COM provides suitable formats both for writing interface specification, using the COM Interface Description Language (IDL), and for run-time interoperability between components.

For each of the generic components, two interfaces are specified: one that is provided by the framework and one that may be provided by protocol handlers. In addition, interfaces are defined to give protocol handlers access to device drivers and system functions. The identities of protocol handlers are provided in the hardware definition files as the Globally Unique Identifiers (GUIDs) of the COM classes that implement them.

COM allows several instances of the same protocol handler to be created. This is useful, for instance, when a controller is connected to two separate networks of the same type. Also, it is useful to have one object, implementing an interface provided by the framework, for each protocol handler that requires the interface.

An additional reason that COM has been chosen is that commercial COM implementations are expected to be available on all operating systems that the software will be released on in the future. The Control Builder is only released on Windows, and it is expected that most future control products will be based on VxWorks, although some products are based on pSOS, for which a commercial COM implementation does not exist. In the first release of the component-based system the protocol handlers were implemented as C++ classes, which are linked statically with the framework. This works well because of the close correspondence between COM and C++, where every COM interface has an equivalent abstract C++ class.

An important constraint on the design of the architecture is that hard real-time requirements, related to scheduling and execution of control programs, must not be affected by interaction with protocol handlers. Thus, all code in the

framework responsible for instantiation and execution of protocol handlers, always executes at a lower priority than code with hard deadlines.

5.3.3 Interaction between Components

When a control system is configured to use a particular device or protocol, the Control Builder uses the information in the hardware definition file to load the protocol handler on the PC and execute the protocol specific routines it implements. During download, the identity of the protocol handler on the controller is sent along with the other configuration information. The controller system software then tries to load this protocol handler. If this fails, the download is aborted and an error message is displayed by the Control Builder. This is very similar to what happens if one tries to download a configuration, which includes a device that is not physically present. If the protocol handler is available, an object is created and the required interface pointers obtained. Objects are then created in the framework and interface pointers to these passed to the protocol handler.

After the connections between the framework and the protocol handler has been set up through the exchange of interface pointers, a method will usually be called on the protocol handler object that causes it to continue executing in a thread of its own. Since the interface pointers held by the protocol handler reference objects in the framework, which are not used by anyone else, all synchronization between concurrently active protocol handlers can be done inside the framework.

To make this more concrete, we now present a simplified description of the interaction between the framework and a protocol handler implementing the server side of a communication protocol on the controller. This relies mainly on the two interfaces `IGenServer` and `IPhServer`. The former is provided by the framework and the latter by protocol handlers implementing server side functionality. Figure 5-7 is a UML structure diagram showing the relationships between interfaces and classes involved in the interaction between the framework and such a protocol handler. The class `CMyProtocol` represents the protocol handler. The interface `IGenDriver` gives the protocol handler access to the device driver for a communication interface.

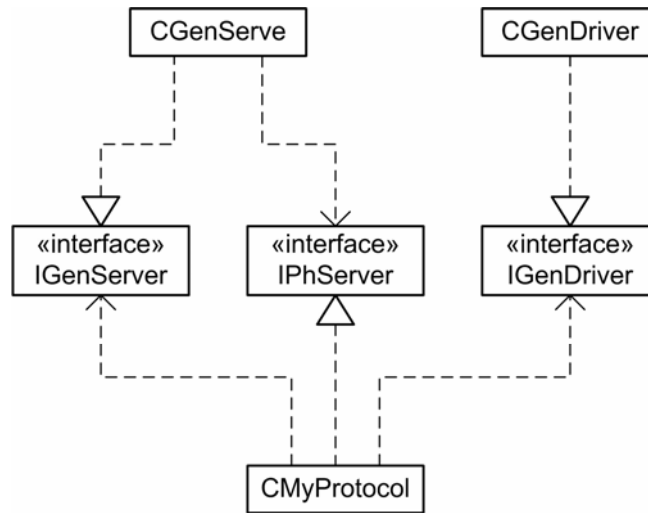


Figure 5-7 Interfaces for communication servers

A simplified definition of the IPhServer interface is shown below. The first two operations are used to pass interface pointers to objects implemented by the framework to the protocol handler. The other two operations are used to start and stop the execution of the protocol handler in a separate thread.

```

interface IPhServer : IUnknown
{
    HRESULT SetServerCallback(
        [in] IGenServer *pGenServer);
    HRESULT SetServerDriver(
        [in] IGenDriver *pGenDriver);
    HRESULT ExecuteServer();
    HRESULT StopServer();
};
    
```

The UML sequence diagram in Figure 5-8 shows an example of what might happen when a configuration is downloaded to a controller, specifying that the controller should provide server-side functionality. The system software first invokes the COM operation CoCreateInstance to create a protocol handler object and obtain an IPhServer interface pointer. Next, an instance of CGenServer is created and a pointer to it passed to the protocol handler using SetServerCallback. Similarly, a pointer to a CGenDriver object is passed using SetDriverCallback. Finally, ExecuteServer is invoked, causing the protocol handler to start running in a new thread.

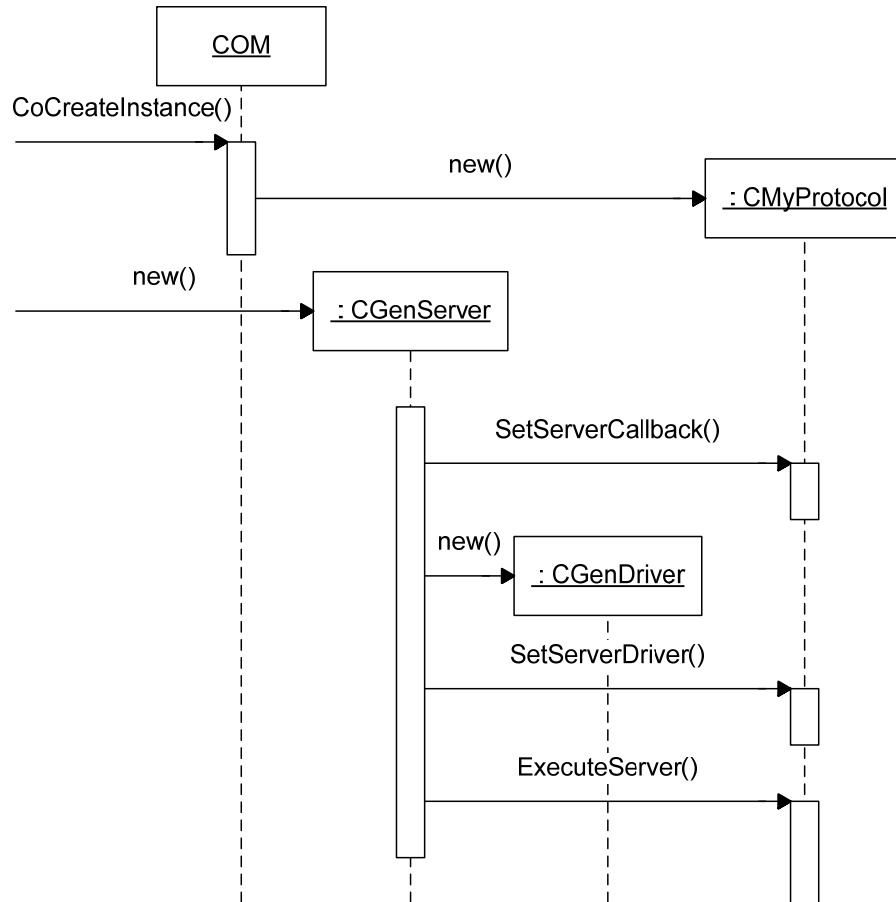


Figure 5-8 Call sequence to set up connections

To see how the execution of the protocol handler proceeds, we first look at a simplified definition of `IGenServer`. The first two operations are used to inform the framework about incoming requests from clients to establish a connection and to take down an existing connection. The last two operations are used to handle requests to read and write named variables, respectively. The index parameter is used with variables that hold structured data, such as records or arrays. All the methods have an output parameter that is used to return a status word.

```

interface IGenServer : IUnknown
{
    HRESULT Connect([out] short *stat);
    HRESULT Disconnect([out] short *stat);
    HRESULT ReadVariable(
        [in] BSTR *name, [in] short index,

```

```

    [out] tVal *pVal, [out] short *status);
HRESULT WriteVariable(
    [in] BSTR *name, [in] short index,
    [in] tVal *pVal, [out] short *status);
};

```

Running in a thread of its own, the protocol handler uses the IGenDriver interface pointer to poll the driver for incoming requests from clients. When a request is encountered the appropriate operation is invoked via the IGenServer interface pointer, and the result of the operation, specified by the status parameter, reported back to the driver and ultimately to the communication client via the network. As an example, Figure 5-9 shows how a read request is handled by calling ReadVariable. The definition of the IGenDriver interface is not included in this discussion for simplicity, so the names of the methods invoked on this interface are left unspecified in the diagram. Write and connection oriented requests are handled in a very similar manner to read requests.

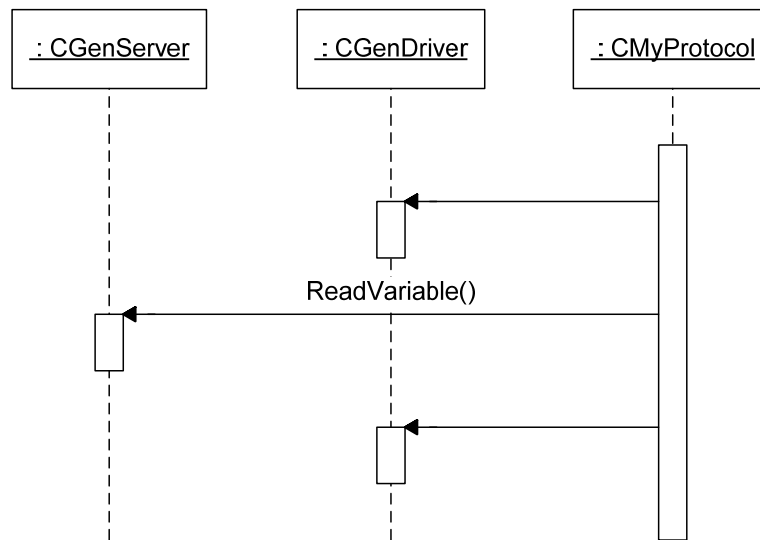


Figure 5-9 Call sequence to handle variable read

The last scenario to be considered here, is the one where configuration information is downloaded, specifying that a protocol handler that was used in the previous configuration should no longer be used. In this case, the connections between the objects in framework and the protocol handler must be taken down and the resources allocated to them released. Figure 5-10 shows how

this is accomplished by the framework first invoking `StopServer` and then `Release` on the `IPhServer` interface pointer. This causes the protocol handler to decrement its reference count, and to invoke `Release` on the interface pointers that have previously been passed to it. This in turn, causes the objects behind these interface pointers in the framework to release themselves, since their reference count reaches zero. Assuming that its reference count is also zero, the protocol handler object also releases itself. If the same communication interface, and thus the protocol handler object, had also been used for different purposes, the reference count would have remained greater than zero and the object not released.

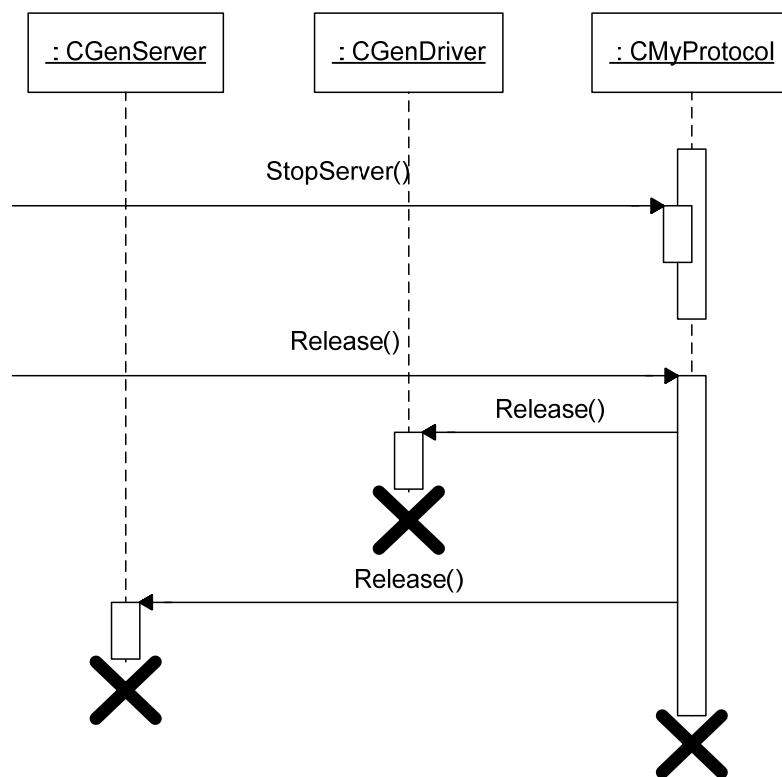


Figure 5-10 Call sequence to take down connections

5.4 Experiences

The definitive measure of the success of the project described in this chapter is how large the effort required to redesign the software architecture has been compared to the effort saved by the new way of adding I/O and communica-

tion support. It is important to remember, however, that in addition to this cost balance, the business benefits gained by shortening the time to market must be taken into account. Also important, although harder to assess, are the long time advantages of the increased flexibility that the component-based software architecture is hoped to provide.

At the time of writing, the parts of the generic I/O and communication framework needed to support communication protocols have been completed, requiring an estimated effort of 15–20 person-years. A number of protocols have been implemented using the new architecture. The total effort required to implement a protocol (including the protocol handler, a device driver, firmware for the communication interface, and possibly IEC 61131-3 function blocks) is estimated to be 3–6 person-years. The reduction in effort compared to that required with the previous architecture is estimated to vary from one third to one half, i.e. 1–3 person-years per protocol. Assuming an average saving of 2 person-years per protocol handler, the savings surpass the investment after the implementation of 8–10 protocols. Table 5-1 summarizes these effort estimations, which were made by technical management at ABB and are primarily based on reported working hours.

System tests have shown that the adoption of the chosen subset of COM has resulted in acceptable system performance. The ability to meet hard real-time requirements has not been affected by the component-based architecture, since all such requirements are handled by threads that cannot be interrupted by the protocol handlers.

Table 5-1 Summary of effort estimation for the two software architectures

Software architecture:	Original	Component-based
Investment in framework:	0	12-15 person years
Cost per protocol:	4-9 person years	3-6 person years
Saving per protocol:	0	1-3 person years
Return on investment:	-	8-10 protocols

An interesting experience from the project is that the componentization is believed to have resulted in a more modularized and better documented system.

Two characteristics generally believed to enhance quality. This experience concurs with the view of Szyperski [16] that adopting a component-based approach may be used to achieve modularization, and may therefore be effective even in the absence of externally developed components. The reduction in the effort required to implement communication protocols is partly due to the fact that the framework now provides some functionality that was previously provided by individual protocol implementations. This is also believed to have increased quality, since the risk of each protocol implementation introducing new errors in this functionality has been removed.

Another interesting experience is that techniques that were originally developed to deal with dynamic hardware configurations have been successfully extended to cover dynamic configuration of software components. In the ABB control system, hardware definition files are used to specify what hardware components a controller may be equipped with and how the system software should interact with different types of components. In the redesigned system, the format of these files has been extended to specify which software components may be used in the system. The true power of this commonality is that existing mechanisms for handling hardware configurations, such as manipulating configuration trees in the Control Builder, downloading configuration information to a control system, and dealing with invalid configurations, can be reused largely as is. The idea that component-based software systems can benefit by learning from hardware design is also aired in [16].

Another lesson of general value is that it seems that a component technology, such as COM, can very well be used on embedded platforms and even platforms where run-time support for the technology is not available. Firstly, we have seen that the space and computation overhead that follows from using COM is not larger than what can be afforded in many embedded systems. In fact, used with some care, COM does not introduce much more overhead than do virtual methods in C++. Secondly, in systems where no such overhead can be allowed, or systems that run on platforms without support for COM, IDL can still be used to define interfaces between components, thus making a future transition to COM straightforward. This takes advantage of the fact that the Microsoft IDL compiler generates C and C++ code corresponding to the interfaces defined in an IDL file as well as COM type libraries. Thus, the same interface definitions can be used with systems of separately linked COM components and statically linked systems where each component is realized as a C++ class or C module.

Among the problems encountered with the componentization, the most noticeable was the difficulty of splitting functionality between independent components, i.e. between the framework and the protocol handlers, and thus determining the interfaces between these components. In all probability, this was in large parts due to the lack of any prior experiences with similar efforts within the development organization. Initially, the task of specifying interfaces was given to the development center responsible for developing the framework. This changed during the course of the project, however, and the interfaces ultimately used were in reality defined in an iterative way in cooperation between the organizational unit developing the framework and those developing protocol handlers. Other problems are of a non-technical nature. An example is the potential problem of what business processes to use if protocol handlers are to be deployed as stand-alone products. So far, protocol handlers have only been deployed as parts of complete controller products, comprising both hardware and software.

5.5 Related Work

A well-published case study with focus on software architecture is that of the US Navy's A-7E avionics system [13]. Among other things, this study demonstrated the use of information hiding to enhance modifiability while preserving real-time performance. Although the architecture of the A-7E system is not component-based in the modern sense, an important step was taken in this direction by decomposing the software into loosely coupled modules with well-defined interfaces.

A more recent study, describing the componentization of a system with the aim to make it easier to add new functionality, has been conducted in the telecommunications domain [1]. In this case study, the monolithic architecture of Ericsson's Billing Gateway Systems is redesigned into one based on distributed components, and a component-based prototype system implemented. In contrast to our case, the system does not have hard real-time requirements, although performance is a major concern. The study shows that componentization of the architecture can improve the maintainability of the system while still satisfying the performance requirements.

There is a substantial body of work on component-based software for control systems and other embedded real-time systems, which, unlike this chapter, focuses on the development of new component models to address the specific

requirements that a system or application domain has with respect to performance, resource utilization, reliability, etc. One of the best known examples is the Koala component model [12] for consumer electronics, which is internally developed and used by Philips. Two other examples with particular relation to the work presented in this chapter are the PECOS component model [6], which was developed with the participation of ABB for use in industrial field-devices, and the DiPS+ component framework [11], which targets the development of flexible protocol stacks in embedded devices.

The primary advantage of such models over more general-purpose models is their effective support for optimization with respect to the most important aspects for the particular application domain. A typical disadvantage is the lack of efficient and inexpensive tools on the market. For instance, building proprietary development tools in parallel with the actual product development may incur significant additional costs.

5.6 Conclusions and Future Work

The experiences described above show that the effort required to add support for communication protocols in the controller product has been considerably reduced since the adoption of the new architecture. Comparing the invested effort of 15–20 person-years with the saving of 1–3 person-years per protocol handler it is concluded that the effort required to design the component-based software architecture is justified by the reduction in the effort required to make pre-specified functional extensions to the software and that the savings surpass the investment after 8–10 such extensions. Based on current plans for protocol handlers to be implemented, it is expected that the savings exceed the investment within 3 years from the start of the project.

In addition to these effort savings and the perceived quality improvements, the component-based architecture has resulted in the removal of the bottleneck at the single development centre and the possibility of developing the framework and several protocol handlers concurrently. This could potentially lead to business benefits such as reduced time to market. Concerning the overhead introduced by the component model, which is small in the current system but may be larger if and when more COM support is incorporated, we believe that the business climate in which industrial control systems are developed justifies a modest increase in hardware resource requirements in exchange for a noticeable reduction in development time.

The experiences with the use of a component-based software architecture in ABB's control system could be further evaluated. For instance, as more protocol handlers are completed, the confidence in the estimated reduction of effort can be increased. Another opportunity is to study the effect on other system properties, such as performance or reliability. A challenge is that this would require that meaningful measures of such properties could be defined and that measures could be obtained from one or more versions of the system before the componentization.

Since a number of protocol handlers have been implemented and even more are planned, there is probably a good opportunity to study the experiences of protocol implementers, which may shed additional light on the qualities of the adopted architecture and component model. One possibility would be to conduct a survey, which might include several development centers. Further opportunities to study the use of a software component model in a real-time system might be offered by a future version of the controller that adopts more of COM and possibly uses a commercial COM implementation.

An issue that may be addressed in the future development at ABB is inclusion of a COM-runtime system with support for dynamic linking between components. Commercially available COM implementations will probably be used for systems based on Windows and VxWorks. Dynamic linking will simplify the process of developing and testing protocol handlers. A potentially substantial effect of dynamic linking is the possibility of adding and upgrading protocol handlers at runtime. This might allow costly production stops to be avoided while, for instance, a controller is updated with a new communication protocol. Another possible continuation of the work presented here, would be to extend the component approach beyond I/O and communication. An architecture where general functionality can be easily integrated by adding independently developed components, would be a great benefit to this type of system, which is intended for a large range of control applications.

5.7 References

- [1] H. Algestam, M. Offesson, and L. Lundberg, "Using Components to Increase Maintainability in a Large Telecommunication System." In *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, 2002.
- [2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd edition. Addison-Wesley, 2003.

- [3] J. Bosch, *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
- [4] D. Box, *Essential COM*. Addison-Wesley, 1997.
- [5] ESPRIT Consortium CCE-CNMA (eds.), *MMS: A Communication Language for Manufacturing*. Springer-Verlag, 1995.
- [6] T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, and P. Müller, "Components for Embedded Software – The PECOS Approach." In *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.
- [7] G. T. Heineman and W. T. Councill (editors), *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [8] International Electrotechnical Commission, *Programmable Controllers – Part 3: Programming Languages*, 2nd edition. IEC Std. 61131-3, 2003.
- [9] F. Lüders, K.-K. Lau, and S.-M. Ho, "Specification of Software Components." In I. Crnkovic and M. Larsson (editors), *Building Reliable Component-Based Software Systems*. Artech House Books, 2000.
- [10] N. Mahalik (editor), *Fieldbus Technology: Industrial Network Standards for Real-Time Distributed Control*. Springer, 2003.
- [11] S. Michiels, *Component Framework Technology for Adaptable and Manageable Protocol Stacks*. PhD thesis, K. U. Leuven, 2004.
- [12] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [13] D. Parnas, P. Clements, and D. Weiss, "The Modular Structure of Complex Systems." In *IEEE Transactions on Software Engineering*, volume 11, issue 3, 1985.
- [14] C. Robson, *Real World Research: A Resource for Social Scientists and Practitioner-researchers*, 2nd edition. Blackwell, 2002.
- [15] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [16] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd edition. Addison-Wesley, 2002.

Chapter 6

A Prototype Tool for Software Component Services in Embedded Real-Time Systems

with Daniel Flemström, Anders Wall, and Ivica Crnkovic

Abstract

The use of software component models has become popular during the last decade, in particular in the development of software for desktop applications and distributed information systems. However, such models have not been widely used in the domain of embedded real-time systems. There is a considerable amount of research on component models for embedded real-time systems, or even narrower application domains, which focuses on source code components and statically configured systems. This paper explores an alternative approach by laying the groundwork for a component model based on binary components and targeting the broader domain of embedded real-time systems. The work is inspired by component models for the desktop and information systems domains in the sense that a basic component model is extended with a set of services for the targeted application domain. A prototype tool for supporting these services is presented and its use illustrated by a control application.

6.1 Introduction

The use of software component models has become increasingly popular during the last decade, especially in the development of software for desktop applications and distributed information systems. Popular component models include *JavaBeans* [5] and *ActiveX* [4] for desktop applications and *Enterprise JavaBeans* (EJB) [11] and *COM+* [15] for distributed information systems. In addition to basic standards for naming, interfacing, binding, etc., these models

also define standardized sets of run-time services oriented towards the application domains they target. Unlike for these domains, there has been no widespread use of software component models in the domain of real-time and embedded systems, presumably due to the special requirements such systems have to meet with respect to timing predictability and limited use of resources. Much research has therefore been directed towards defining new component models for real-time and embedded systems. Typically, such models are based on static configurations of source code components and target relatively narrow application domains. Examples include the *Koala* component model for consumer electronics [22], *PECOS* for industrial field devices [6], and *SaveCCM* for vehicle control systems [7].

An alternative approach is to strive for a component model based on binary components and targeting a broader domain of applications, similar to the domain targeted by a typical real-time operating system. The approach pursued in this paper is to provide a combination of restrictions and extensions of an existing component model to adapt it to our target domain. Adapting an existing component model has several advantages: It may be possible to use existing (integrated) development environments; existing components can be re-used or adapted for the real-time domain; integration with application from other domains becomes significantly simpler, and so on.

Our previous work has demonstrated that the key concepts of the *Component Object Model* (COM) [3] can be beneficially used in the development of an embedded real-time system [10]. A study of COM and its extension *Distributed COM* (DCOM) [17] shows that these models are not inherently incompatible with real-time requirements, although some restrictions on how the models are used may be necessary to ensure predictability [9]. Some reasons that COM is an attractive starting point are that the model is relatively simple, commercial COM implementations are already available for a few real-time operating systems, and COM is already well-known and accepted in industry. The goal of this paper is to lay the groundwork for a software component model for embedded real-time systems by using the basic concepts of COM as the starting point and extending the basic model with standardized services of general use for this application domain, much like COM+ extends COM with services for distributed information systems.

The remainder of the paper is organized as follows. In Section 6.2 we clarify what we mean by software component services and identify some useful services for embedded real-time systems. Section 6.3 is an overview of a proto-

type tool we are developing to support such services, including an example control application to demonstrate the use of the tool. Related work is reviewed in Section 6.4 and conclusions and some ideas for further work are presented in Section 6.5.

6.2 Component Services

In this paper we define component services as solutions to common problems that can be added to components without modifying them and with little or no adaptation of application code. This is similar to the concept of component services in EJB and COM+, where examples of services include transaction control, data persistence, and security. Our focus is on services that address common challenges in embedded real-time systems, including logging, synchronization, and timing control. Traditionally, such functions have to be hand coded and off line deduced using complex theories, which can be very time consuming and sometimes impossible in complex industrial systems. If third party components are used, it may also be impossible to implement functions by modifying the components. In the following subsections we describe some of the services we have identified in more depth and outline how they may be implemented. In general, we propose that services are implemented through the use of proxy objects, which are automatically generated from configuration files written in an XML based format.

6.2.1 Logging

A logging service allows the sequence of interactions between components to be traced. Our suggested solution for achieving this is to use a proxy object as illustrated in the UML class diagram in Figure 2-1. In the diagram, the object C2 implements an interface IC2 for which we wish to apply a logging service. A proxy object that also implements IC2 is placed between C2 and a client that uses the operations exposed through IC2. The operations implemented by the proxy forward all invocations to the corresponding operations in C2 in addition to writing information about parameter values, return codes, and invocation and return times to some logging medium. To add logging of all operation invocations through an interface, we simply add an entry in the configuration file:

```

<application>
...
<component name="myProject.C2">
  <interface name="IC2">
    <service type ="Logging"/>
  </interface>
</component>
...
</application>

```

No programming is required in the client C1 or the component C2. To add logging only for a particular operation, the entry is modified as follows:

```

<interface name="IC2">
  <operation name="DoSomething">
    <service type ="Logging"/>
  </operation>
</interface>

```

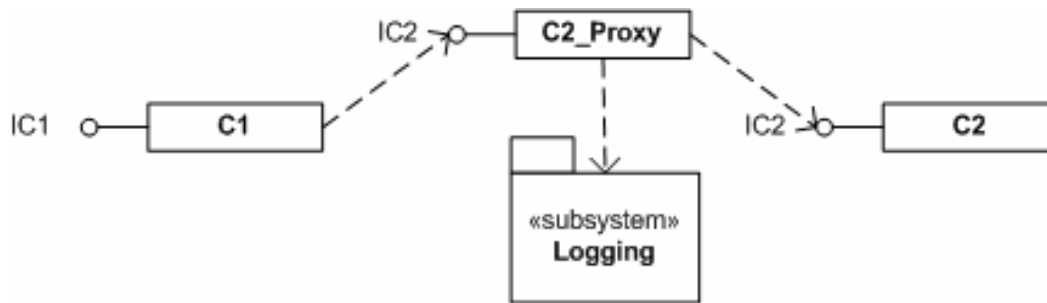


Figure 6-1 Implementing a logging service through a proxy object

6.2.2 Execution Time Measurement

This service allows operation invocations to be monitored and information about execution times accumulated. Different measurements, such as worst-case, best-case, and average execution time may be collected. A possible use of the information is to dynamically adapt an on-line scheduling strategy. The suggested solution is to use a forwarding proxy that measures the time elapsed from each operation call till it returns and collects the desired timing information. As with the logging service, the time measurement service is specified in the configuration file:

```

<interface name="IC2">
  <service type="Timing">
    <measurement type="Mean" />
    <measurement type="Worst"/>
  </service>
</interface>

```

Again, no programming is required.

6.2.3 Synchronization

A synchronization service allows components that are not inherently thread-safe to be used in multi-threaded applications. The suggested solution is to use forwarding proxies that use the basic mechanisms of the underlying operating system to implement the desired synchronization policies. A synchronization policy may be applied to a single operation or to a group of operations, e.g. all operations of an interface or a component. Several different policies may be useful and will be described further in this section. Most synchronization policies rely on blocking and it may be useful to combine such policies with timeouts to limit blocking time. If the blocking time for an operation call reaches the timeout limit, the proxy return an error without forwarding the call. A more advanced timeout policy is one where the proxy tries to determine if a call can be satisfied without violating the timeout limit a priori and, if not, returns an error immediately.

The simplest synchronization policy is *mutual exclusion*, which blocks all operation calls except one. After the non-blocked call completes, the waiting calls are dispatched one by one according to the priority policy. This policy may be applied merely by adding an entry in the configuration file but, if timeouts are used, the client should be able to handle the additional error codes that may arise. Another class of synchronization policies is different *reader/writer* policies. These differs from the previously described policy in that any number of calls to read operations may execute concurrently, while each call to write operations has exclusive execution. Thus, the operations subjected to a reader/writer policy must be classified as either writer or reader operations, depending on whether they may modify state or not. Concurrent read calls are scheduled according to their priorities.

Using this policy requires that it be specified for each operation whether it is a read or write type of operation. This can be done in the component specifica-

tion (e.g. a COM IDL file) or in the configuration file. If this is left unspecified for an operation, the proxy must assume it may write data. No programming is required, except possibly to handle error codes resulting from timeouts. For all synchronization policies, we may select if the priority of the dispatching thread should be the same as the calling thread, or explicitly specified in the configuration file. A specification of a reader/writer policy may look as follows:

```
<interface name="IC2">
  <service type="Synchronization" policy="RWPolicy"/>
  <operation name="DoSomething" type="Write"/>
  <operation name="WriteData" type="Write"/>
  <operation name="ReadData" type="Read" />
</interface>
```

6.2.4 Execution Timeout

This service can be used to ensure that a call to a component's operation always terminate within a specified deadline, possibly signaling a failure if the operation could not be completed within that time. The solution is to use a proxy that uses a separate thread to forward each operation call and then wait until either that thread terminates or the deadline expires. In the latter case the proxy signals the failure by returns an error code. Also, it is possible to specify different options for what should be done with the thread of the forwarded call if the deadline expires. The simplest option is to forcefully terminate the thread, but this may not always be safe since it may leave the component in an undefined and possibly inconsistent state. Another option is to let the operation call run to completion and disregard its output. Obviously, using this service requires that the client is able to handle timeouts. Again, the service is specified in the configuration file:

```
<interface name="IC2">
  <service type="Timeout" deadline="10ms"
    fail="Terminate"/>
</interface>
```

6.2.5 Vertical Services

In addition to the type of services discussed above, which we believe are generally useful for embedded real-time systems, one can imagine many services aimed at more specific application domains, often called *vertical services* [8]. Among the services we have considered are cyclic execution, which are much used in process control loops [1], and support for redundancy mechanisms such as N-version components, which are useful in fault-tolerant systems [2]. The prototype tool presented in the next section includes an implementation of a cyclic execution service.

6.3 Prototype Tool

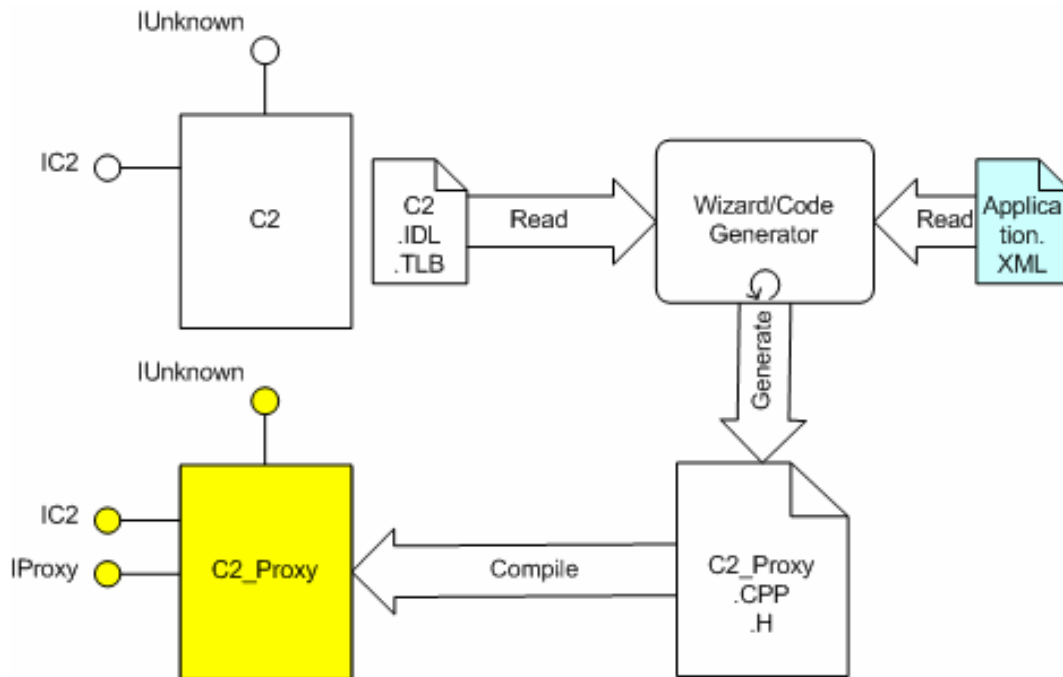


Figure 6-2 Generating a proxy object for a component service

This section outlines a prototype tool we are developing that adds services to COM components on *Microsoft Windows CE*. The tool generates source code for proxy objects implementing services by intercepting method calls to the COM objects. The tool takes as inputs component specifications along with a specification of the desired services for each component. Component specifications

may be in the form of Interface Definition Language (IDL) files or their binary equivalent Type Library (TLB) files. Desired services are either specified in a separate file using an XML-based format or in the tool's graphical user interface, described further below. Note that access to component source code is not required. Based on these inputs, the tool generates a complete set of files that can be used with *Microsoft eMbedded Visual C++* (sic) to build a COM component implementing the proxy objects (i.e., the proxies are themselves COM objects). This process is depicted in Figure 6-2.

6.3.1 Design Consideration

The use of proxy objects for interception is heavily inspired by COM+. However, rather than to generate proxies at run-time, we suggest that these are generated and compiled on a host computer (typically a PC) and downloaded to the embedded system along with the application components. There, the proxy COM classes must be registered in the COM registry in such a way that proxy objects are placed between interacting application components. This process may occur when the software is initially downloaded to the system or as part of dynamic reconfiguration of a system that supports this. In the latter case, one can imagine updating or adding proxies without updating or adding any application components. The current version of the tool only generates proxy code and does not address the registration and run-time instantiation of components. This means that the client code must instantiate each proxy along with the affected COM object and set up the necessary connection between them. A desirable improvement would be to automate this task, either by generating code that performs setup for each proxy object or by extending the COM run-time environment with a general solution.

We consider staying as close as possible to the original COM and COM+ concepts an important design goal for the tool. Another goal is that the programmer or integrator should be able to choose desired services for each component without having to change the implementation or doing any programming. There are however cases, e.g. when adding invocation timeouts, where there is a need for adapting the code of the client component to fully benefit from the service. Specific to COM is that a component is realized by a set of COM classes that, in turn, each implements a number of interfaces. All interfaces have a method called *QueryInterface* that allows changing from one interface to another on the same COM class. Since each proxy is implemented by a

COM class, which must satisfy the definition of QueryInterface, we must generate one proxy for each COM class to which we wish to add any services.

6.3.2 Supported Services

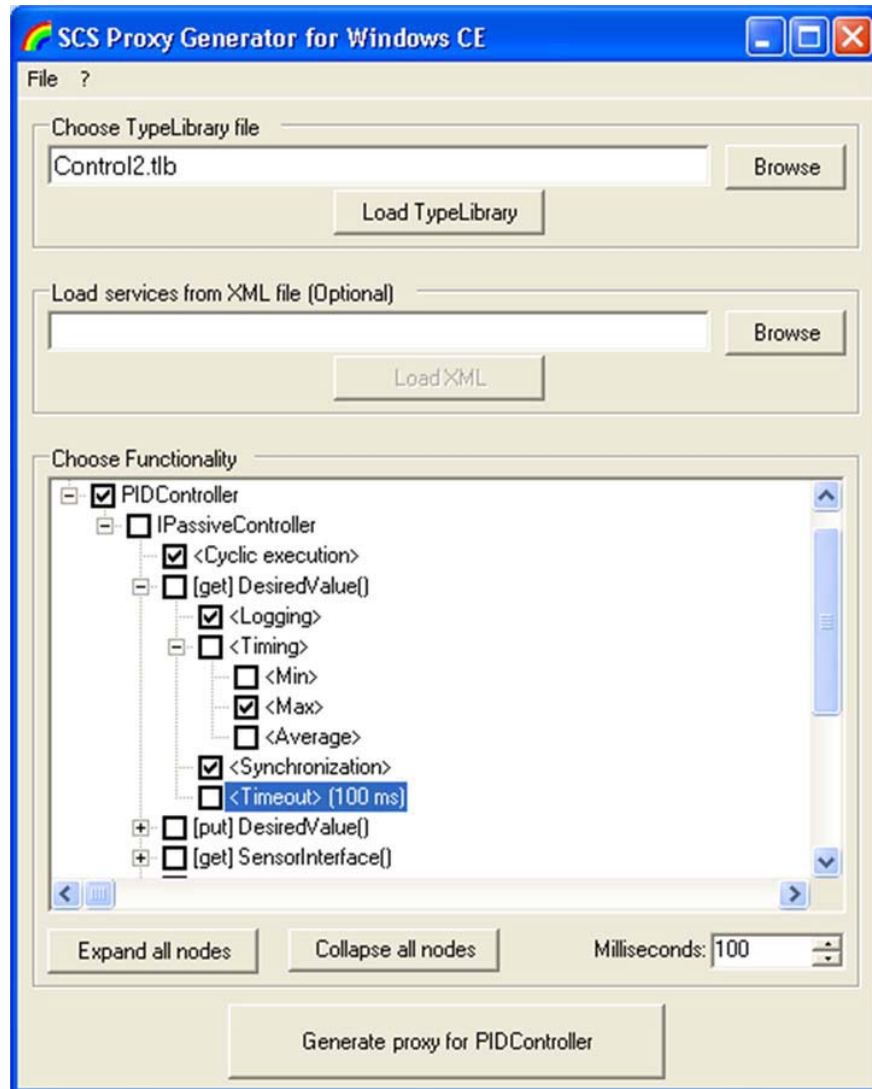


Figure 6-3 The graphical user interface of the prototype tool

Figure 6-3 shows the graphical user interface of the tool. After a TLB or IDL file has been loaded all COM classes defined in the file are listed. Checking the box to the left of a COM class causes a proxy for that class to be generated

when the button at the bottom of the tool is pressed. Under each COM class, the interfaces implemented by the class is listed and, under each interface, the operations implemented by the interface. In addition, the available services are listed with their names set in brackets. Checking the box to the left of a service causes code to be generated that provides the service for the element under which the service is listed. In the current version of the tool, a service for cyclic execution may only be specified for the `IPassiveController` interface (see example below), while all other services may only be specified for individual operations. Checking the box to the left of an interface or operation is simply a quick way of checking all boxes further down in the hierarchy.

If the cyclic execution service is checked, the proxy will implement an interface called `IActiveController` instead of `IPassiveController` (see example below). Checking the logging service results in a proxy that logs each invocation of the affected operation. The timing service causes the proxy to measure the execution time of the process and write it to the log at each invocation (if timing is checked but not logging, execution times will be measured but not saved). The synchronization service means that each invocation of the operation will be synchronized with all other invocations of all other operations on the proxy object for which the synchronization service is checked. The only synchronization policy currently supported is mutual exclusion. The timeout service has a numeric parameter. When this service is selected (by clicking the name rather than the box) as in Figure 6-3, an input field marked Milliseconds is visible near the bottom of the tool. Checking the service results in a proxy where invocations of the operation always terminate within the specified number of milliseconds. In the case that the object behind the proxy does not complete the execution of the operation within this time, the proxy forcefully terminates the execution and returns an error code.

6.3.3 Example Application

To illustrate the use of the tool we have implemented a component that encapsulates a digital Proportional-Integral-Differential (PID) controller []. For the purpose of comparison, we first implemented a component that does not rely on any services provided by the tool. Figure 6-4 shows the configuration of an application that uses this component. `PIDController` is a COM class that implements an interface `IActiveController` and relies on the two interfaces `ISensor` and `IActuator` to read and write data from/to the controlled process. For the purpose of this example, these interfaces are implemented by

the simple COM class DummyProcess that does nothing except returning a constant value to the controller. The interfaces are defined as follows:

```

interface ISensor : IUnknown {
    [propget] HRESULT ActualValue(
        [out, retval] double *pVal);
};

interface IActuator : IUnknown {
    [propget] HRESULT DesiredValue(
        [out, retval] double *pVal);
    [propput] HRESULT DesiredValue(
        [in] double newVal);
};

interface IController : IActuator {
    [propget] HRESULT SensorInterface(
        [out, retval] ISensor **pVal);
    [propput] HRESULT SensorInterface(
        [in] ISensor *newVal);
    [propget] HRESULT ActuatorInterface(
        [out, retval] IActuator **pVal);
    [propput] HRESULT ActuatorInterface(
        [in] IActuator *newVal);
    [propget] HRESULT CycleTime(
        [out, retval] double *pVal);
    [propput] HRESULT CycleTime(
        [in] double newVal);
    [propget] HRESULT Parameter(
        short Index, [out, retval] double *pVal);
    [propput] HRESULT Parameter(
        short Index, [in] double newVal);
};

interface IActiveController : IController {
    [propget] HRESULT Priority(
        [out, retval] short *pVal);
    [propput] HRESULT Priority(
        [in] short newVal);
    HRESULT Start();
    HRESULT Stop();
};

```

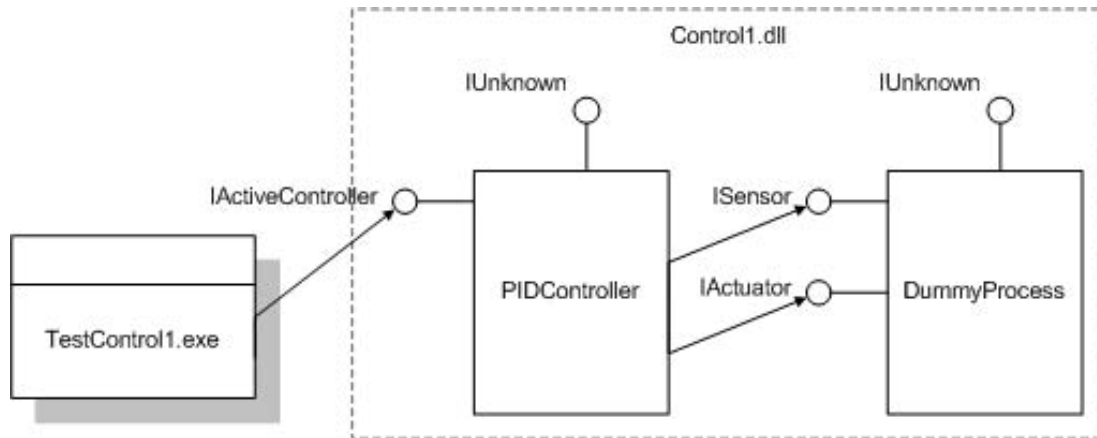


Figure 6-4 An application using a controller component without services

`IController` is a generic interface for a single-variable controller with configurable cycle time and an arbitrary number of control parameters. `PIDController` uses three parameters for the proportional, integral, and differential gain. `IActiveController` extends this interface to allow control of the controller's execution in a separate thread. The reason for splitting the interface definitions like this is that we wish to reuse `IController` for a controller that uses our cyclic execution service rather than maintaining its own thread. Note that `IController` inherits the `DesiredValue` property from `IActuator`. This definition is chosen to allow the interface to be used for cascaded control loops where the output of one controller forms the input to another.

The test application `TestControl1.exe` creates one instance of `PIDController` and one instance of `DummyController`. It then connects the two objects by setting the `SensorInterface` and `ActuatorInterface` properties of the `PIDController` object. After this it sets the cycle time and the control parameters before invoking the `Start` operation. This causes the `PIDController` object to create a new thread that executes a control loop. A simple timing mechanism is used to control the execution of the loop in accordance with the cycle time property. At each iteration the loop reads a value from the sensor interface, which it uses in conjunction with the desired value, the control parameters, and an internal state based on previous inputs to compute and write a new value to the actuator interface. To minimize jitter (input-output delay as well as sampling variability), this part of the loop uses internal copies of all variables, eliminating the need for any synchronization.

Next, the control loop updates its internal variables for subsequent iterations. Since the desired value and the control parameters may be changed by the application while the controller is running, this part of the loop uses a mutual exclusion mechanism for synchronization. In addition to performing its control task the loop timestamps and writes the sensor and actuator data to a log. The control loop is illustrated by the following pseudo code:

```

while (Run) {
    WaitForTimer();
    ReadSensorInput();
    ComputeAndWriteActuatorOutput();
    WriteDataToLog();
    WaitForMutex();
    UpdateInternalState();
    ReleaseMutex();
}

```

Note that, due to the simple timing mechanism, the control loop will halt unless all iterations complete within the cycle time.

Next, we implemented a component intended to perform the same function, but relying on services provided by generated proxies. A test application using this component and generated proxies is shown in Figure 6-5. In this application, `PIDController` is a COM class that implements the `IPassiveController` interface. Note that, although this COM class has the same human readable name as in the application described above, it has a distinct identity to the COM run-time environment. To avoid confusion we use the notation `Control2.PIDController` when appropriate. `IPassiveController` extends `IController` as follows:

```

interface IPassiveController : IController {
    HRESULT UpdateOutput();
    HRESULT UpdateState();
};

```

These operations are used by the `PIDController_Proxy` object to implement a control loop that performs the same control task as in the previous example.

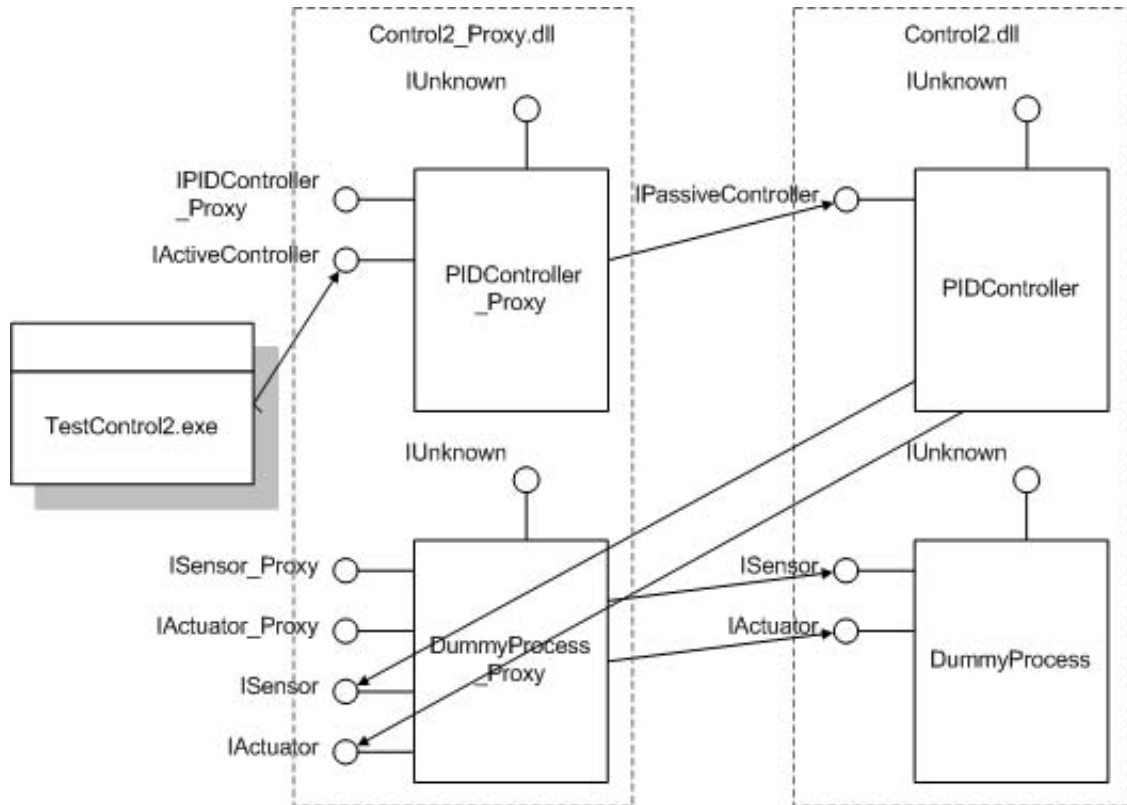


Figure 6-5 An application using a controller component with services

PIDController_Proxy was generated with the use of the tool by checking the cyclic execution service under the Control2.PIDController's IPassiveController interface and the synchronization service under the UpdateState operation as well as the operations for accessing the desired value and the control parameters. The DummyProcess_Proxy provides the interface pointers for the controller's SensorInterface and ActuatorInterface properties. Behind this proxy is a DummyProcess object with the same functionality as in the previous example. DummyProcess_Proxy was generated by the tool with the logging service checked. As a result, all data read and written via the sensor and actuator interfaces are logged. The interfaces ISensor_Proxy, IActuator_Proxy and IPIDController_Proxy are only used to set up the connections between proxies and other objects. They are defined as follows:

```

interface ISensor_Proxy : IUnknown {
    HRESULT Attach([in] ISensor *pTarget);
};

interface IActuator_Proxy : IUnknown {
    HRESULT Attach([in] IActuator *pTarget);
};

interface IPIDController_Proxy : IUnknown {
    HRESULT Attach([in] IPassiveController *pTarget);
};

```

In order to evaluate to two test applications we built and executed them on the Windows CE 4.0 Emulator. Since the timing accuracy on the emulator is 10 milliseconds, it was not possible to measure any timing differences between the two applications. In both cases the controller worked satisfactory for cycle times of 20 milliseconds or more (the measured input-output delay as well as sampling variability was zero – from which we can only conclude that the actual times are closer to zero than 10 milliseconds). For shorter cycle times, both controllers ultimately halted since the limited timer accuracy caused the control loop to fail to complete its execution before the start of the next cycle. Also, we were not able to see any systematic difference in memory usage for the two applications. Clearly, further evaluation of the effects of the services on timing and memory usage is desirable.

To estimate the difference in programming effort and code size for the two applications we compared the amounts of source code and sizes of compiled files. These size metrics for the various components are presented in Table 6-1. The middle column shows the number of non-empty lines of source code. For the first three components, the number only include the source code of the C++ classes implementing the COM objects, i.e. the automatically generated code included in all COM components is not included. Taking these numbers as (admittedly primitive) measurements of programming effort, we see that using the tool to generate service proxies has resulted in a saving of 127 lines or 42 per cent. On the other hand, we see that the effort required for the client program is substantially greater in the case where the proxies are used. This is due to the need for the program to set up the connections between the proxies and the other objects. We conclude that the usefulness of our approach would greatly benefit from automation of this task.

Table 6-1 Size metrics for components

Component	Lines of source code	File size in KB
Control1.dll	300	56.5
Control2.dll	173	53.5
Control2_Proxy.dll	351	60.5
TestControl1.exe	81	12.5
TestControl2.exe	157	14

As for the code size, there is only a small difference between the three COM components, leading to an overhead of roughly 100 per cent from using the proxies. This is largely due to the fact that the implemented COM objects are relatively small, leading to the obligatory house-keeping code of all COM components taking up a large percentage of the code size. For larger COM objects, the relative code sizes approaches the relative sizes of the source code. The small size of the COM objects is also the main reason that the component implementing the proxy objects is the largest of all the components. In addition, the generated code is designed to be robust in the sense that all the operations of the proxy objects verify that the interface pointers have been set before forwarding operation calls. An obvious trade-off would be to sacrifice this robustness for less overhead in execution time as well as space. From the file size of the two test programs we find that the code overhead for setting up the connections between the proxies and the other objects is a little more than 10 per cent. This overhead, unlike the overhead on programming effort, cannot be eliminated by automating the setup task.

6.4 Related Work

The services discussed in this paper have already been adopted by some current and emerging technologies. As a base for our discussions, we have selected a few of the most common solutions for these. In addition, this section briefly reviews some existing research on binary components for real-time systems.

Microsoft's component model COM [3] originally targets the desktop software domain. Thus, it has good support for specifying and maintaining functional aspects of components while disregarding temporal behavior and resource utilization. Often this can only be overcome with a substantial amount of component specific programming. There is no built in support to automatically measure and record execution times for methods in components. This is typically done by third party applications that instrument the code in run-time. These applications are typically not well suited for executing on embedded resource constrained systems. The desktop version of COM, as well as the DCOM package available for Windows CE, has some support for synchronizing calls to components that are not inherently thread safe. This is achieved through the use of so-called *apartments*, which can be used to ensure that only one thread can execute code in the apartment at a time. Since this technique originates from the desktop version of COM, there is no built in support for time determinism and the resource overhead is larger than desired for many embedded systems.

COM+ [15] is Microsoft's extension of their own COM model with services for distributed information systems. These services provide functionality such as transaction handling and persistent data management, which is common for applications in this domain and which is often time consuming and error prone to implement for each component. Builders of COM+ application declare which services are required for each component and the run-time system provides the services by intercepting calls between components. COM+ is a major source of inspiration for our work in two different ways. Firstly, we use the same criteria for selecting which services our component model should standardize, namely that they should provide non-trivial functionality that is commonly required in the application domain. Since our component model targets a different domain than COM+, the services we have selected are different from those of COM+ as well. Secondly, we are inspired by the technique of providing services by interception. This mechanism is also used in other technologies and is sometimes called *interceptors* rather than proxies, e.g. in the *Common Object Request Broker Architecture* (CORBA) [14] and the *MEAD* framework for fault-tolerant real-time CORBA applications [13].

The approach presented in this paper is similar to the concept of aspects and weaving. In [21], a real-time component model called *RTCOM* is presented which has support for weaving of functionality into components as aspects while maintaining real-time policies, e.g. execution times. However, *RTCOM* is a proprietary source code component model. Moreover, functionality is

woven in at the level of source code in RTCOM whereas in our approach, services are introduced at the system composition level.

Another aspect-oriented approach is presented in [18], which describes a method using C# attributes to generate a proxy that handles component replication for fault tolerance. Our work is primarily targeting COM and C++, which does not support attributes as used in that paper. An obstacle to the use of C# for the type of systems we are interested in is the lack of real-time predictability in the underlying *.NET Framework* [16]. The possibility of adding real-time capabilities to the .NET framework are described in [23].

A model for monitoring of components in order to gain more realistic WCET estimations is described in [20]. In this model the WCET is guessed at development time and the component is then continuously monitored at runtime and measurements of execution times are accumulated. This technique is very similar to our execution time measurement service.

Another effort to support binary software components for embedded real-time systems is the *Robocup* project [12], which builds on the aforementioned Koala model and primarily targets the consumer electronics domain. This work is similar to ours in that the component model defined as part of this project is largely based on the basic concepts of COM. Furthermore, the sequel of the project, called *Space4U* [19], also seems to use a mechanism similar to proxy objects, e.g. to support fault-tolerance.

6.5 Conclusion and Future Work

The aim of this work has been to lay the groundwork for component services for embedded real-time systems using COM as a base technology. A major benefit of this approach is that industrial programmers can leverage their knowledge of existing technologies. Also, extending COM with real-time services probably requires less effort than inventing a new component technology from the ground.

The initial experiences with the prototype shows that it is possible to create a tool that more or less invisibly add real-time services to a standard component model. The example application demonstrates that the use of generated proxies to implement services may substantially reduce the complexity of software components. Another conclusion to be drawn from the example is that our

approach would benefit from also automating the configuration of applications with proxies.

We have been able to identify some component services which we believe are useful for embedded real-time systems. As part of our future work, we plan to evaluate the usefulness of the services as well as to extend the set of services. We hope to do this with the help of input from organizations developing products in such domains as industrial automation, telecommunication, and vehicle control systems.

We realize that the proposed solutions imposes some time and memory overhead, and we believe that this is an acceptable price for many embedded real-time systems if using the model reduces the software development effort. It is, however, necessary that this overhead can be kept within known limits. So far, our prototype implementation has been tested with the Windows CE emulator, where we have found no noticeable run-time overheads. In our future work, we plan to evaluate the solution experimentally on a system running Windows CE. Measurements will be made to determine the effect on timing predictability as well as time and memory overhead.

We furthermore aim to empirically evaluate our approach with respect to its effect on development effort and such quality attributes as reliability and reusability. Our hypothesis concerning reliability is that it may improve as a result of reduced complexity of application components, provided of course that the generated proxies are reliable. We also believe reusability may be affected positively, as e.g. the use of synchronization services could make it easier to reuse components across applications that share some functionality but rely on different synchronization policies. The primary evaluating technique will be to conduct replicated student projects where software is developed both with and without the prototype tool. A possible complementary technique is industrial case studies, which implies a lower level of control and replication but may allow more realistic development efforts to be investigated.

6.6 References

- [1] K. J. Åström and B. Wittenmark, *Computer Controlled Systems: Theory and Design*, 2nd edition. Prentice Hall, 1990.
- [2] Avizienis, "The Methodology of N-version Programming." In M. R. Lyu (editor), *Fault Tolerance*. Wiley, 1995.

- [3] D. Box, *Essential COM*. Addison-Wesley, 1997.
- [4] D. Chappell, *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [5] R. Englander, *Developing Java Beans*. O'Reilly, 1997.
- [6] T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, and P. Müller, "Components for Embedded Software – The PECOS Approach." In *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.
- [7] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren, "SaveCCM – A Component Model for Safety-Critical Real-Time Systems. In *Proceedings of the 30th EROMICRO Conference*, 2004.
- [8] G. T. Heineman and W. T. Council (editors), *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [9] F. Lüders, "Adopting a Software Component Model in Real-Time Systems Development." In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop*, 2004.
- [10] F. Lüders, I. Crnkovic, and P. Runeson, "Adopting a Component-Based Software Architecture for an Industrial Control System – A Case Study." In C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper (editors), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- [11] R. Monson-Haefel, B. Burke, and S. Labourey, *Enterprise JavaBeans*, 4th edition. O'Reilly, 2004.
- [12] J. Muskens, M. R. V. Chaudron, and J. J. Lukkien, "A Component Framework for Consumer Electronics Middleware." In C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper (editors), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- [13] P. Narasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: Support for Real-Time Fault-Tolerant CORBA." In *Concurrency and Computation: Practice and Experience*, volume 17, issue 12, 2005.
- [14] Object Management Group, Common object request broker architecture: Core specification. OMG formal/04-03-12, 2004.
- [15] D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.

- [16] D. S. Platt, *Introducing Microsoft .NET*, 3rd edition. Microsoft Press, 2003.
- [17] F. E. Redmond III, *DCOM: Microsoft Distributed Component Object Model*. Wiley, 1997.
- [18] W. Schult and A. Polze, "Aspect-Oriented Programming with C# and .NET." In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2002.
- [19] Space4U Project, "Space4U Public Homepage." <http://www.hitech-projects.com/euprojects/space4u/>, accessed on 28 April 2006.
- [20] D. Sundmark, A. Möller, and M. Nolin, "Monitored Software Components - A Novel Software Engineering Approach." In *Proceedings of the 11th Asia-Pacific Software Engineering Conference, Workshop on Software Architectures and Component Technologies*, 2004.
- [21] A. Tešanović, D. Nyström, J. Hansson, and C. Norström, "Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software." In *Journal of Embedded Computing*, volume 1, issue 1, 2004.
- [22] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [23] Zerzelidis and A. J. Wellings, "Requirements for a Real-Time .NET Framework." In *ACM SIGPLAN Notices*, volume 40, issue 2, 2005.

Chapter 7

Use of Software Component Models and Services in Embedded Real-Time Systems

with Shoaib Ahmad, Faisal Khizer, and Gurjodh Singh-Dhillon

Abstract

While the use of software component models has become popular in the development of desktop applications and distributed information systems, such models have not been widely used in the domain of embedded real-time systems. Presumably, this is due to the requirements such systems have to meet with respect to predictable timing and limited use of resources. There is a considerable amount of research on component models for embedded real-time systems that focuses on source code components, statically configured systems, and relatively narrow application domains. This paper explores the alternative approach of using a mainstream component model based on binary components. The effects of using the model on timing and resource usage have been measured by implementing example applications both with and without using the model. In addition, the use of a prototype tool for supporting software component services has been investigated in the same manner.

7.1 Introduction

The use of software component models has become popular in the development of desktop applications and distributed information systems, where popular component models include *JavaBeans* [1] and *ActiveX* [2] for desktop applications and *Enterprise JavaBeans* (EJB) [3] and *COM+* [4] for information systems. In addition to basic standards for naming, interfacing, binding, etc.,

these models also define standardized sets of run-time services oriented towards the application domains they target. This concept is generally termed software component services [5].

Software component models have not been widely used in the development of real-time and embedded systems. It is generally assumed that this is due to the special requirements such systems have to meet, in particular with respect to timing predictability and limited use of resources such as memory and CPU time. Much research has been directed towards defining new component models for real-time and embedded systems, typically focusing on relatively small and statically configured systems. Most of the published research proposes models based on source code components and targeting relatively narrow application domains. Examples of such models include the *Koala* component model for consumer electronics [6], *PECOS* for industrial field devices [7], and *SaveCCM* for vehicle control systems [8].

An alternative approach is to strive for a component model for embedded real-time systems based on binary components and targeting a broader domain of applications, similarly to the domain targeted by a typical real-time operating system. This paper explores the possibility of using a mainstream component model as the starting point for such a model. Specifically, the use of the *Component Object Model* (COM) [9] with the real-time operating system *Windows CE* [10] is investigated. We have empirically evaluated the effect of using COM by implementing applications both with and without using the model. In addition, we have evaluated the effects of using a prototype tool for supporting software component services in embedded real-time systems.

The rest of this paper is organized as follows. Section 7.2 provides background information on COM and the prototype tool. Section 7.3 presents an automatic control applications that we use as an example to evaluate the use of these technologies. In Section 7.4, we described the tests we have conducted and their results. These results are discussed in Section 7.5. Section 7.6 is an overview of some related work. Conclusions and ideas for future work are presented in Section 7.7.

7.2 Background

7.2.1 The Component Object Model (COM)

Microsoft's Component Object Model (COM) [9] is one of the most commonly used software component models for desktop and server side applications. Although the model is increasingly being replaced by the newer .NET technology [11] in these domains, we believe COM is a more suitable starting point for a model aimed at embedded real-time systems because of its relative simplicity. In particular, the use of automatic memory management (garbage collection) in .NET is a serious barrier against ensuring predictable timing.

A key principle of COM is that interfaces are specified separately from both the components that implement them and those that use them. COM defines a dialect of the Interface Definition Language (IDL) that is used to specify object-oriented interfaces. Interfaces are object-oriented in the sense that their operations are to be implemented by a class and passed a reference to a particular instance of that class when invoked. The code that uses a component does not refer directly to any objects, however. Instead, the operations of an interface supported by an object are invoked via what is known as an interface pointer. A concept known as interface navigation makes it possible for the user to obtain a pointer to every interface supported by the object.

COM also defines a run-time format for interface pointers. What an interface pointer really references is an interface node, which in turn, contains a pointer to a table of function pointers, called a VTABLE. Typically, the node also contains a pointer to an object's instance data, although this is implementation specific. This use of VTABLEs is identical to the way that many C++ compilers implement virtual methods. Thus, the time and space overhead associated with accessing an object through an interface pointer is presumably the same as that incurred with C++ virtual methods. Figure 7-1 illustrates the typical format of interface nodes.

For most real-time systems, a more serious concern than these modest overheads is that interface navigation introduces a possible source of run-time errors. If the user of a component asks an object for a pointer to an interface that the object does not support, this will not be detected during compilation. It may be argued, in fact, that this is the principal difference between interface navigation and interface inheritance in traditional object-oriented program-

ming. This can be seen as a necessary price to pay for the otherwise desirable reduced compile-time dependence between components.

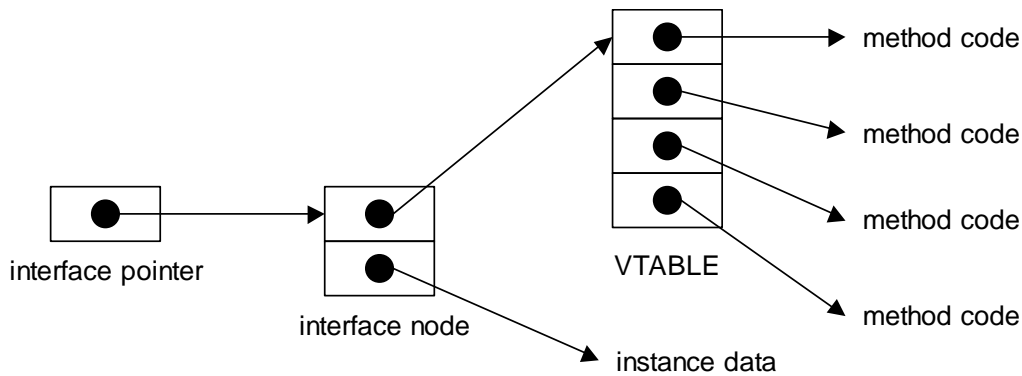


Figure 7-1 Typical format of COM interface nodes

As already mentioned, a COM component is implemented in classes. The mechanism for creating instances of these classes is closely linked with how and when the code in different components is linked together. COM defines a policy for instantiation, which is intended to ensure that different components can be installed in a system at different times. When a component is installed, information about it must be registered somewhere in the system, linking the identity of its classes to the code that implement these. COM also requires a run-time library, called the COM library, to be installed on the system. When some code wants to use a component, it uses an operation provided by the COM library to ask for an instance of a class and an initial interface pointer to it. If the code of the component is not already loaded into memory, the COM library uses the registered information to locate the code and load it before an instance is created. This process is illustrated in Figure 7-2.

Thus, creation of an instance involves searching the information about registered classes and possibly loading of code. This leads to a noticeable overhead when compared to instantiation in for instance C++. Furthermore, this overhead will vary, depending on whether the code implementing a class has already been loaded or not. This variability can be eliminated, however, by designing the software such that all components that may be used will be loaded at start-up. Note that removal of instances is subject to the same variability, since the COM standard states that code can be unloaded when the last instance that rely on it is removed.

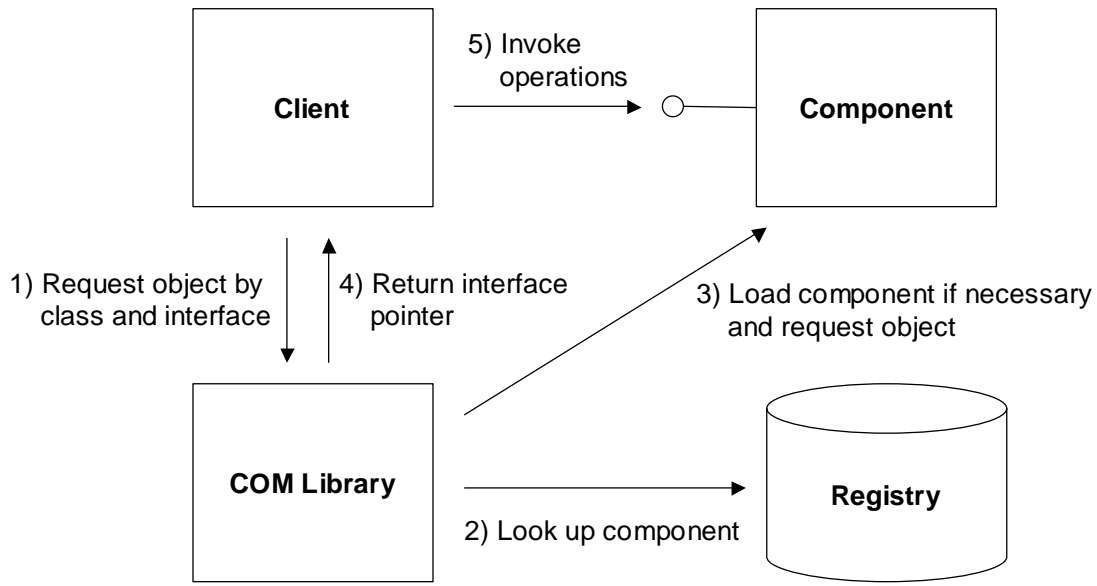


Figure 7-2 Instance creation and dynamic loading of code in COM

A benefit that follows from COM's way of creating instances is that the code that implements a component can be built independently of any code that uses the component. Since instantiation involves passing the identity of the desired class as a parameter to a system operation, it is a possible source of run-time errors, which is not present during instantiation in traditional object-oriented programming, since attempting to instantiate a class that does not exist will result in a compilation error in this case. Again, this is a necessary price to be paid for decreased coupling.

7.2.2 Software Component Services for Embedded Real-Time Systems

A prototype tool for supporting software component services in embedded real time systems is presented in [12]. The tool adds services to COM components on Windows CE through the use of proxy object that intercept method calls. Figure 7-3 illustrates the use of a proxy object that provides a simple logging service. The object C2 implements an interface IC2 for which we wish to apply a logging service. A proxy object that also implements IC2 is placed between C2 and a client that uses the operations exposed through IC2. The operations implemented by the proxy forward all invocations to the corresponding operations in C2 in addition to writing information about parameter values, return codes, and invocation and return times to some logging medium.

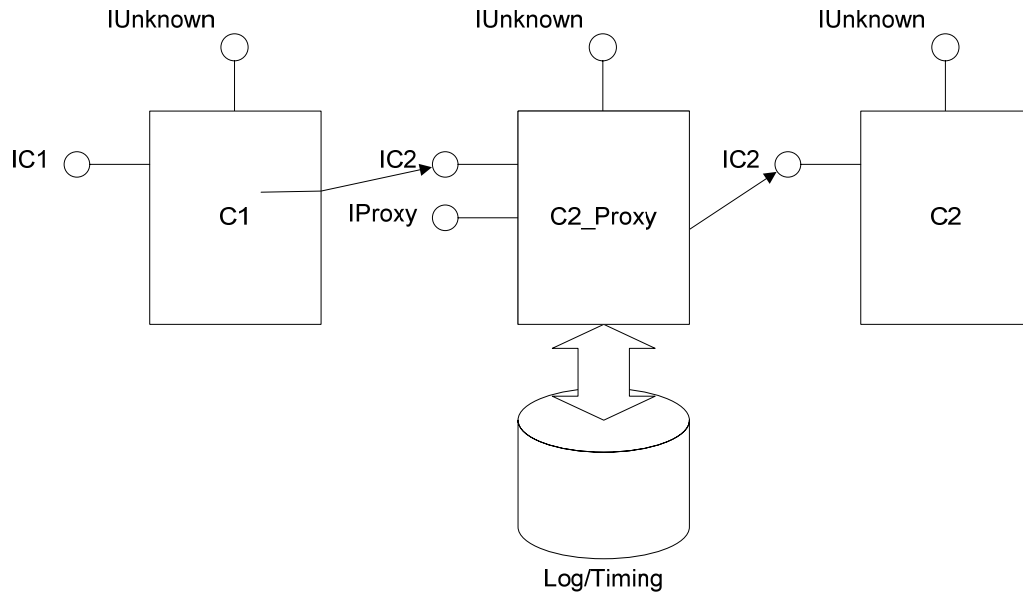


Figure 7-3 A logging service proxy

The tool takes as inputs a component specification along with specifications of desired services and generates source code for a proxy object. Component specifications may be in the form of Interface Definition Language (IDL) files or their binary equivalent Type Library (TLB) files. Desired services are either specified in a separate file using an XML-based format or in the tool's graphical user interface, described further below. Note that access to component source code is not required. Based on these inputs, the tool generates a complete set of files that can be used with *Microsoft eMbedded Visual C++* to build a COM component implementing the proxy objects (i.e., the proxies are themselves COM objects). This process is depicted in Figure 7-4.

This use of proxy objects for interception is inspired by COM+. However, rather than to generate proxies at run-time, they are generated and compiled on a host computer and downloaded to the embedded system along with the application components. This process may occur when the software is initially downloaded to the system or as part of dynamic reconfiguration of a system that supports this. In the latter case, one can imagine updating or adding proxies without updating or adding any application components. The current version of the tool only generates proxy code and does not address the registration and run-time instantiation of components. This means that the client code must instantiate each proxy along with the affected COM object and set up the necessary connection between them.

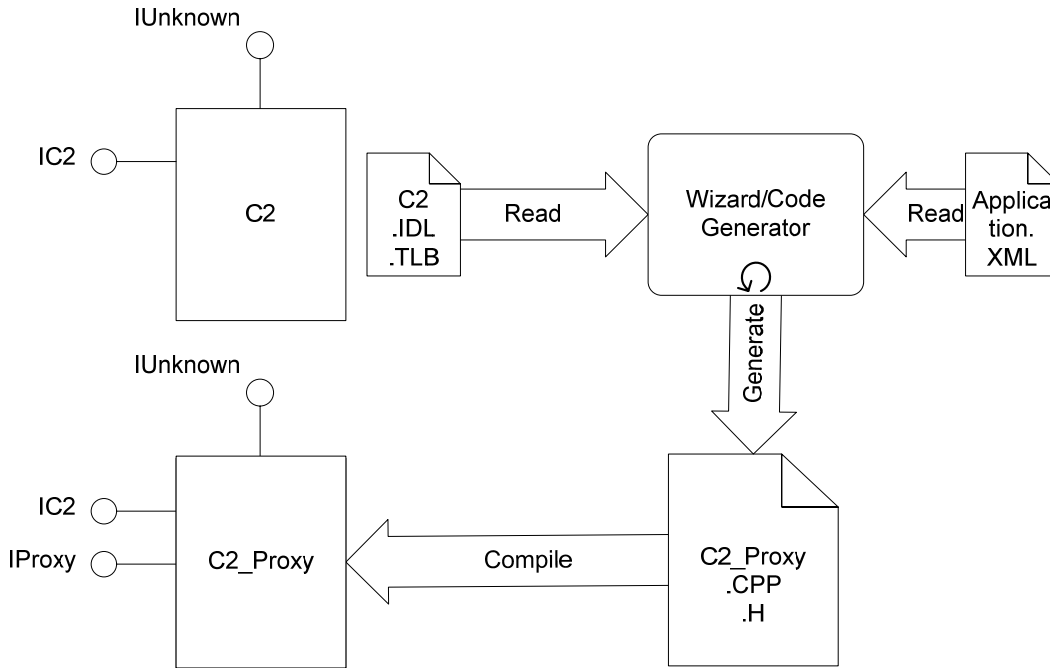


Figure 7-4 Proxy object generation

In addition to logging, the tool supports generating proxies that implement one or more of the following services: execution time measurement of method invocations; synchronization between concurrent invocations; execution time-out on invocations; and cyclic execution of methods.

Figure 7-5 shows the graphical user interface of the tool. After a TLB or IDL file has been loaded all COM classes defined in the file are listed. Checking the box to the left of a COM class causes a proxy for that class to be generated when the button at the bottom of the tool is pressed. Under each COM class, the interfaces implemented by the class is listed and, under each interface, the operations implemented by the interface. In addition, the available services are listed with their names set in brackets. Checking the box to the left of a service causes code to be generated that provides the service for the element under which the service is listed. In the current version of the tool, a service for cyclic execution may only be specified for the `IPassiveController` interface while all other services may only be specified for individual operations. The `IPassiveController` interface is described in connection with the example application in the next section.

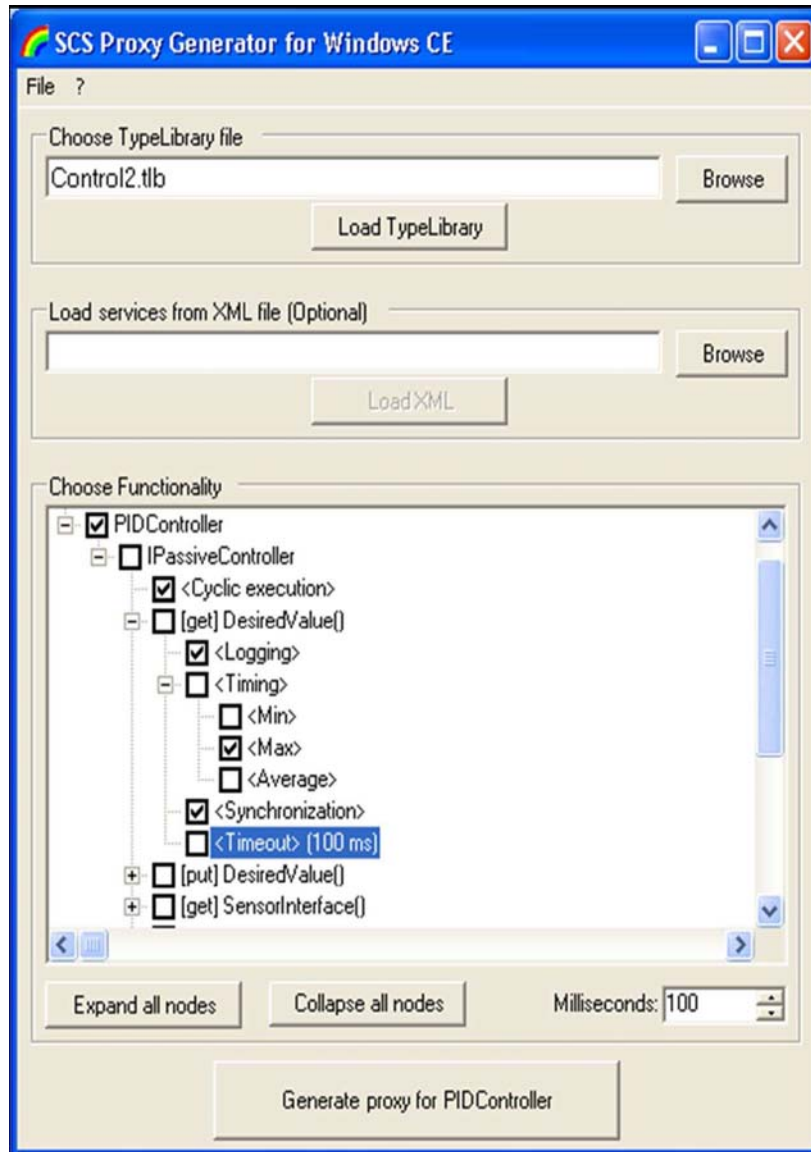


Figure 7-5 User interface of the prototype tool

If the cyclic execution service is checked, the proxy will implement an interface called `IActiveController` instead of `IPassiveController` (see the example in the next section). `IActiveController` includes operations for setting the period and threading priority of the cyclic execution. Checking the logging service results in a proxy that logs each invocation of the affected operation. The timing service causes the proxy to measure the execution time of the process and write it to the log at each invocation (if timing is checked but not logging, execution times will be measured but not saved).

The synchronization service means that each invocation of the operation will be synchronized with all other invocations of all other operations on the proxy object for which the synchronization service is checked. The only synchronization policy currently supported is mutual exclusion. The timeout service has a numeric parameter. When this service is selected (by clicking the name rather than the box) as in Figure 7-5, an input field marked Milliseconds is visible near the bottom of the tool. Checking the service results in a proxy where invocations of the operation always terminate within the specified number of milliseconds. In the case that the object behind the proxy does not complete the execution of the operation within this time, the proxy forcefully terminates the execution and returns an error code.

7.3 Example Application

To evaluate the effects of using both COM and the prototype tool, we used the example application presented in [12]. At the center of this application is a component that encapsulates a *proportional-integral-differential* (PID) controller [13]. Four different versions of the application were implemented. They are presented here in the order in which they were first developed. The four versions are summarized in Table 7-1 at the end of this section.

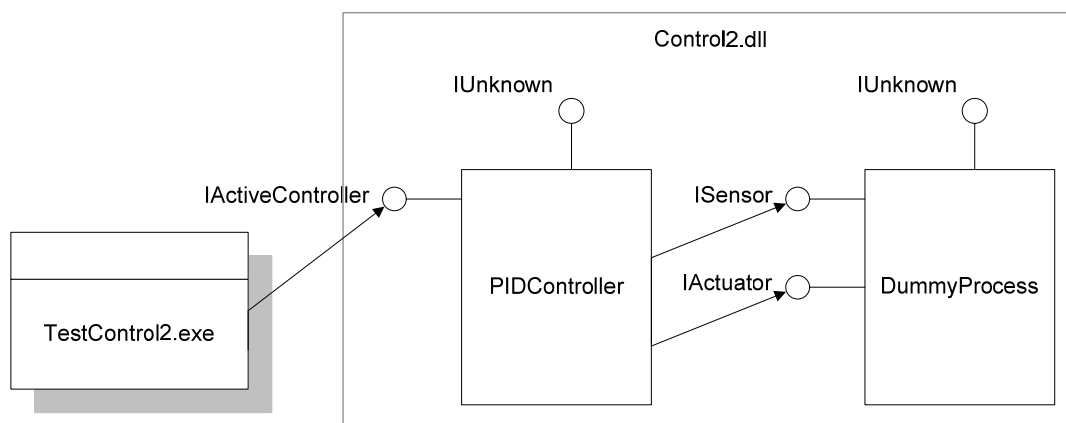


Figure 7-6 Implementation with COM

We first implemented a version using COM, shown in Figure 7-6, which we term Control2. `PIDController` is a COM class that implements an interface `IActiveController` and relies on the two interfaces `ISensor` and `IActuator`.

tuator to read and write data from/to the controlled process. For the purpose of this example, these interfaces are implemented by the simple COM class `DummyProcess` that does nothing except returning a constant value to the controller. The interfaces are defined as follows:

```

interface ISensor : IUnknown
{
    [propget] HRESULT ActualValue(
        [out, retval] double *pVal);
};

interface IActuator : IUnknown
{
    [propget] HRESULT DesiredValue(
        [out, retval] double *pVal);
    [propput] HRESULT DesiredValue(
        [in] double newVal);
};

interface IController : IActuator
{
    [propget] HRESULT SensorInterface(
        [out, retval] ISensor **pVal);
    [propput] HRESULT SensorInterface(
        [in] ISensor *newVal);
    [propget] HRESULT ActuatorInterface(
        [out, retval] IActuator **pVal);
    [propput] HRESULT ActuatorInterface(
        [in] IActuator *newVal);
    [propget] HRESULT CycleTime(
        [out, retval] double *pVal);
    [propput] HRESULT CycleTime(
        [in] double newVal);
    [propget] HRESULT Parameter(
        [in] short Index, [out, retval] double *pVal);
    [propput] HRESULT Parameter(
        [in] short Index, [in] double newVal);
};

interface IActiveController : IController
{
    [propget] HRESULT Priority(
        [out, retval] short *pVal);
};

```

```

    [propput] HRESULT Priority(
        [in] short newVal);
    HRESULT Start();
    HRESULT Stop();
};

```

`IController` is a generic interface for a single-variable controller with configurable cycle time and an arbitrary number of control parameters. `PIDController` uses three parameters for the proportional, integral, and differential gain. `IActiveController` extends this interface to allow control of the controller's execution in a separate thread. (The reason for splitting the interface definitions like this was to reuse `IController` for a controller that uses the cyclic execution service rather than maintaining its own thread.) Note that `IController` inherits the `DesiredValue` property from `IActuator`. This definition was chosen to allow the interface to be used for cascaded control loops where the output of one controller forms the input to another.

The test application `TestControl2.exe` creates one instance of `PIDController` and one instance of `DummyController`. It then connects the two objects by setting the `SensorInterface` and `ActuatorInterface` properties of the `PIDController` object. After this it sets the cycle time and the control parameters before invoking the `Start` operation. This causes the `PIDController` object to create a new thread that executes a control loop. A simple timing mechanism is used to control the execution of the loop in accordance with the cycle time property. At each iteration the loop reads a value from the sensor interface, which it uses in conjunction with the desired value, the control parameters, and an internal state based on previous inputs to compute and write a new value to the actuator interface. To minimize jitter (input-output delay as well as sampling variability), this part of the loop uses internal copies of all variables, eliminating the need for any synchronization.

Next, the control loop updates its internal variables for subsequent iterations. Since the desired value and the control parameters may be changed by the application while the controller is running, this part of the loop uses a mutual exclusion mechanism for synchronization. In addition to performing its control task the loop timestamps and writes the sensor and actuator data to a log. The control loop is illustrated by the following pseudo code:

```

while (Run)
{

```

```

    WaitForTimer();
    ReadSensorInput();
    ComputeAndWriteActuatorOutput();
    WriteDataToLog();
    WaitForMutex();
    UpdateInternalState();
    ReleaseMutex();
}

```

Note that, due to the simple timing mechanism, the control loop will halt unless all iterations complete within the cycle time.

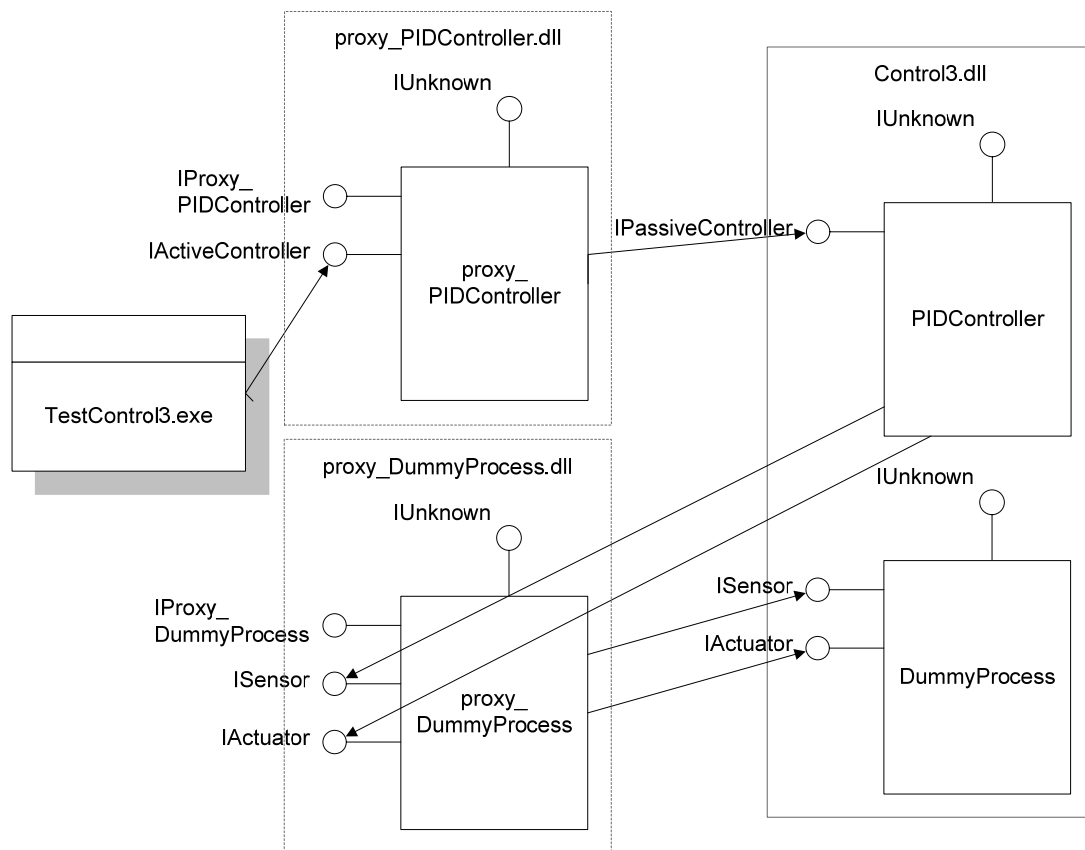


Figure 7-7 Implementation with COM and generated proxies

Next, we implemented a component intended to perform the same function, but relying on services provided by generated proxies. A test application using this component and proxies is shown in Figure 7-7. In this application,

termed `Control3.PIDController` is a COM class that implements the `IPassiveController` interface. Note that, although this COM class has the same human readable name as in the application described above, it has a distinct identity to the COM run-time environment. To avoid confusion we use the notation `Control3.PIDController` when appropriate. `IPassiveController` extends `IController` as follows:

```
interface IPassiveController : IController
{
    HRESULT UpdateOutput();
    HRESULT UpdateState();
};
```

These operations are used by the `proxy_PIDController` object to implement a control loop that performs the same control task as in the previous example.

The `proxy_PIDController` COM class was generated with the use of the tool by checking the cyclic execution service under the `IPassiveController` interface of `Control3.PIDController`. The `proxy_DummyProcess` COM class provides the interface pointers for the controller's `SensorInterface` and `ActuatorInterface` properties. Behind this proxy is a `DummyProcess` object with the same functionality as in the `Control2` application. `proxy_DummyProcess` was generated by the tool with the logging service checked. As a result, all data read and written via the sensor and actuator interfaces are logged. The interfaces `IDummyProcess_Proxy` and `IPIDController_Proxy` are only used to set up the connections between proxies and other objects. They are defined as follows:

```
interface IProxy_DummyProcess : IUnknown
{
    HRESULT AttachISensor([in] IUnknown *pTarget);
    HRESULT AttachIActuator([in] IUnknown *pTarget);
};

interface IProxy_PIDController : IUnknown
{
    HRESULT AttachIPassiveController(
        [in] IUnknown *pTarget);
};
```

To be able to evaluate the overhead introduced by the use of COM and the generated proxies, we implemented two non-component-based versions of the application, each consisting of a single executable file. Figure 7-8 shows the internal structure of these programs, termed Control0 and Control1, as UML class diagrams.

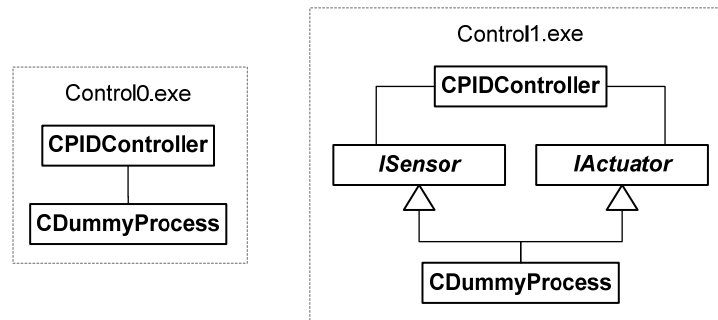


Figure 7-8 Non-component-based implementations

The application termed Control1 was constructed by making very modest modifications to the source code of the Control2 application. The main modification was that the calls to the COM library for creating instances of COM classes were replaced by simple instantiation of C++ classes. The C++ classes CPIDController and CDummyProcess are identical to those used internally to implement the COM classes of Control2. ISensor and IActuator are abstract C++ classes that correspond directly to the COM interfaces of the same names. They are specified in C++ as follows:

```

class ISensor : public IUnknown
{
    virtual HRESULT get_ActualValue(double *pVal) = 0;
};

class IActuator : public IUnknown
{
    virtual HRESULT get_DesiredValue(double *pVal) = 0;
    virtual HRESULT put_DesiredValue(double val) = 0;
};
  
```

Control0 is a modified version of Control1, where the classes are modified such that virtual methods are not used. This means that calls to the methods are not performed using VTABLES of function pointers, and the address of the

methods are determined at compile-time rather than at run-time. The abstract classes are removed, since such classes rely entirely on virtual methods. Table 7-1 summarizes the four different versions of the application.

Table 7-1 Summary of application versions

Name	Description
Control0	Using C++ without virtual methods
Control1	Using C++ with virtual methods
Control2	Using COM
Control3	Using COM and proxy-based services

7.4 Tests

7.4.1 Test Setup

The example application described in the previous section was tested on a system running Window CE 5.00. The hardware used was a PC with a 2.8 GHz Pentium 4 processor. The Windows CE run-time image was built using Microsoft Platform Builder 5.00 with the standard board support package for a Windows CE based PC (CEPC) and the standard setting provided by the “Industrial Controller” platform template. This platform allowed time measurements to be made with a resolution of one millisecond. Each of the four versions of the application was built with Microsoft eMbedded Visual C++ and tested on the target computer one at a time, resetting the target between each test.

For each of the four versions of the example application, two different execution times were measured. The first was the time required for invocation of the `get_ActualValue` method of the `DummyProcess` COM objects or, in the case of `Control0` and `Control1`, of the `CDummyProcess` C++ objects. Given the one millisecond resolution, we were required to modify the control loop of the programs by adding an inner loop that performed two million invocations of `get_ActualValue` instead of a single invocation to obtain usable time measurements. For each of the versions, this measurement was made 170 times.

The second measurement made for each of the versions was the time required for initialization of the application. This initialization includes instantiation of the COM or C++ objects and setting up of the connections between them. This test was performed 20 times for each of the versions of the example application.

In addition to execution times, measurements of memory usage were also performed. However, we were not able to see any difference between the four different versions of the test application on the test platform we used. Also, differences between the size of source code and binary files were presented in [12] and are not repeated here. Thus, the following presentation and discussion of the results focus on execution times.

7.4.2 Results

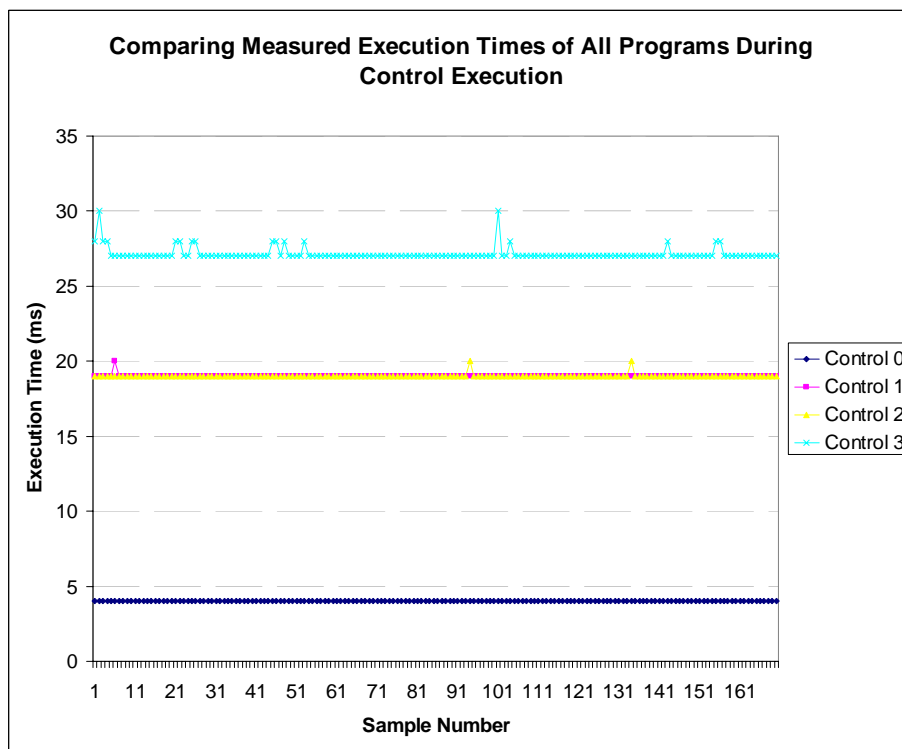


Figure 7-9 Measured execution times

Figure 7-9 shows measured execution times of making two million invocations of `get_ActualValue` for the four different version of the example applica-

tion. The measurements for Control0 (without COM and not using virtual methods) are the lowest with an average of four milliseconds. These measurements show no variation, but given that the resolution is one millisecond the uncertainty per measurement is 25%.

Control1 (without COM but using virtual methods) and Control2 (with COM) give similar results of approximately 19 milliseconds on average and 5% variation. This indicates that the overhead of using COM as well as of using virtual methods in C++ is approximately 15 milliseconds. Taking into account that two million invocations were made per measurement, this corresponds to an invocation overhead of 7.5 nanoseconds for this particular processor.

Control3 (with COM and all invocations passing through a proxy objects) gives approximately 27 milliseconds on average and 11% variation. This indicates an additional overhead of approximately eight milliseconds compared to Control2, corresponding to four nanoseconds per invocation. Table 7-2 summarizes the measurements depicted in Figure 7-9.

Table 7-2 Summary of execution times

Version	Execution time (ms)		
	Min.	Max.	Average
Control0	4	4	4
Control1	19	20	19.00588
Control2	19	20	19.01176
Control3	27	30	27.12353

Figure 7-10 shows measured execution times of application initialization for Control0, Control1, and Control2. The measurements for Control0 (where the initialization consists of instantiating two C++ classes and passing a reference of one instance to the other) give an average of 0.4 milliseconds. For Control1 (where the initialization is very similar) the average is 0.7 milliseconds and for Control2 (where initialization involves calling the COM library to instantiate the COM classes) one millisecond. Given that these values are so small compared to the one millisecond resolution and that only 20 measurements were

collected in each case, they can only be viewed as crude estimations of the real execution time.

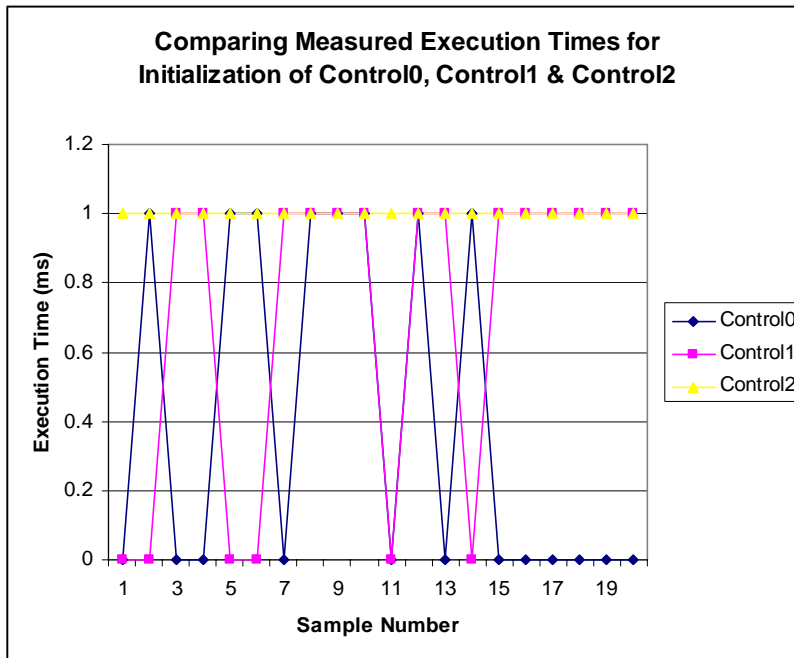


Figure 7-10 Measured initialization times for Control0, Control1, and Control2

Figure 7-11 shows measured execution times of application initialization for Control3. For this implementation (where the initialization comprises calling the COM library to instantiate four different COM classes in three different components and performing a comparatively complex setup task) the average is approximately 2940 milliseconds, which is of course notably higher than for the other implementations. The variation is also quite high with a difference of 4686 milliseconds between the minimum and maximum. If we treat the maximum value as an outlier we get approximately 2730 milliseconds on average and 40% variation. The measured execution times of application initialization are summarized in Table 7-3.

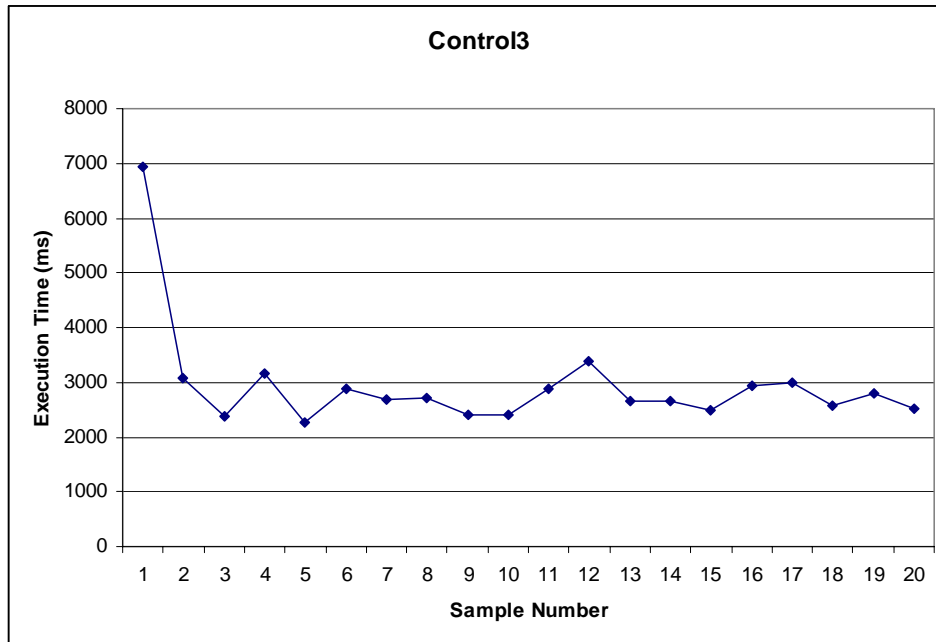


Figure 7-11 Measured initialization times for Control3

Table 7-3 Summary of initialization times

Version	Execution time (ms)		
	Min.	Max.	Average
Control0	0	1	0.4
Control1	0	1	0.7
Control2	1	1	1
Control3	2264	6950	2940.85

7.5 Discussion

The measured execution times during control execution constitute quite strong evidence that the overheads of using both COM and the proxy-based services are modest and, even more importantly in many real-time systems, quite predictable. The overhead of using COM interfaces pointers are found to be es-

essentially equal to that of using C++ virtual methods. It should be noted here, however, that using C++ classes allows mixing of virtual and statically bound methods. Typically, a carefully designed C++ class will use virtual methods only where variability is desired, leading to a lower average overhead than for an equivalent COM class where invocation through interface pointers is mandatory.

Comparing the use of proxy-based services with the use of COM without services, the additional overhead is found to be quite modest. In some cases, however, the increased number of indirections might be expected to lead to an increase in the number of cache misses and thereby a higher penalty. Clearly, the tests presented in this paper, where a single operation has been invoked repeatedly in a loop, is much less likely to result in cache misses than more realistic usage scenarios.

Based on the measured execution times during application initialization, it seems safe to conclude that the times required when C++ or COM is used are of the same order of magnitude, while the time required when proxy-based services are used are several orders of magnitude higher. Nonetheless, these measurements leave much to be desired. For Control0, Control1, and Control2, it would be desirable to perform additional measurements using loops that repeat the initializations to obtain higher values and hence increased accuracy. For Control3, additional measurements to reveal the most time consuming parts of the initialization phase would be very desirable.

7.6 Related Work

Although models based on source code component still seem to dominate, there are other efforts to support binary software components for embedded real-time systems. One example is the *ROBOCOP* research project [14], which builds on the aforementioned Koala component model and primarily targets the consumer electronics domain. The component model defined as part of this project is largely based on the basic concepts of COM. Furthermore, the sequel of the project, called *Space4U* [15], also seems to use a mechanism similar to software component services, e.g. to support fault-tolerance.

The approach to software component services discussed in this paper relies heavily on the technique of providing services by interception. This mechanism is also used in other technologies and is sometimes called interceptors

rather than proxies, e.g. in the *Common Object Request Broker Architecture* (CORBA) [16] and the *MEAD* framework for fault-tolerant real-time CORBA applications [17]. The approach is furthermore similar to the concept of aspects and weaving. In [18], a real-time component model called *RTCOM* is presented which have support for weaving of functionality into components as aspects while maintaining real-time properties. An important difference with our approach is that, in *RTCOM*, functionality is weaved in at the level of source code.

Another effort towards adapting a mainstream component model to the embedded real-time systems domain is presented in [19]. This work aims to extend the Enterprise JavaBeans model with means for specifying timing properties of software components. As it focuses on specification and not run-time issues, it is complementary to our work rather than an alternative. The fact that it is based on EJB rather than COM is not of principal importance, but the lack of Java run-time environments for embedded real-time systems may mean that the approach is further from real-world application.

In general, the concept of software component services can be seen as a special case of middleware. The use of middleware in embedded real-time systems is an active topic of research (and practice) not necessarily related to software components. Similar to our approach of adapting a mainstream component model, efforts have been made to adapt mainstream middleware to the domain of embedded real-time systems [20]. Specialized middleware frameworks for this domain also exist, including *OSA+* [21] that provides services for distributed systems and *Kokyu* [22] that provides flexible scheduling and dispatching services.

7.7 Conclusion and Future Work

The aim of the work presented in this paper has been to investigate the possibility of using a mainstream software component model, as well as an extension of this model with run-time services, for developing embedded real-time systems. We believe that the results show that this is a promising approach, although further investigation, in particular of the overheads related to object instantiation, should be undertaken. The overheads related to invoking operations through COM interfaces as well as through a forwarding proxy were found to be both modest and predictable. Thus, these overheads would probably be quite acceptable in many embedded real-time systems. In particu-

lar, we can conclude that a system that can afford the invocation overhead of C++ virtual methods can also afford COM interfaces, since the cost is nearly identical.

Since we view the use of a mainstream component model like COM as an alternative to more specialized models, it would be interesting to conduct a comparative study of COM (and possibly other mainstream models) and at least one specialized model. A possible object of such a study is the aforementioned Koala component model, which is supported by freely available tools. Differences between models, e.g. in terms of time and memory overheads, should be investigated empirically by implementing example applications.

In addition to the run-time effects on resource usage and predictability, the effects of using the approach on development effort and such quality attributes as reliability and reusability should be evaluated. In our future work, we aim to do this using different empirical techniques, including both controlled experiments and case studies with student participation. In addition, it would be desirable to perform industrial case studies, which implies a lower level of control and replication, but allows more realistic situations to be investigated.

7.8 References

- [1] R. Englander, *Developing Java Beans*. O'Reilly, 1997.
- [2] D. Chappell, *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [3] R. Monson-Haefel, B. Burke, and S. Labourey, *Enterprise JavaBeans*, 4th edition. O'Reilly, 2004.
- [4] D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.
- [5] G. T. Heineman and W. T. Council (editors), *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [6] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [7] T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, and P. Müller, "Components for Embedded Software – The PECOS Approach." In *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.

- [8] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törnngren, "SaveCCM - A Component Model for Safety-Critical Real-Time Systems." In *Proceedings of the 30th EROMICRO Conference*, 2004.
- [9] D. Box, *Essential COM*. Addison-Wesley, 1997.
- [10] J. Murray, *Inside Microsoft Windows CE*, Microsoft Press, 1998.
- [11] D.S. Platt, *Introducing Microsoft .NET*, 3rd edition. Microsoft Press, 2003.
- [12] F. Lüders, D. Flemström, A. Wall, and I. Crnkovic, "A Prototype Tool for Software Component Services in Embedded Real-Time Systems." In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, 2006.
- [13] K. J. Åström and B. Wittenmark, *Computer Controlled Systems: Theory and Design*, 2nd edition. Prentice Hall, 1990.
- [14] J. Muskens, M. R. V. Chaudron, and J. J. Lukkien, "A Component Framework for Consumer Electronics Middleware." In C. Atkinson, C. Bunse, H. Gross, and C. Peper (editors.), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- [15] Space4U Project, "Space4U Public Homepage." <http://www.hitech-projects.com/euprojects/space4u/>, accessed on 28 April 2006.
- [16] Object Management Group, *Common Object Request Broker Architecture: Core Specification*. OMG formal/04-03-12, 2004.
- [17] P. Narasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: Support for Real-Time Fault-Tolerant CORBA." In *Concurrency and Computation: Practice and Experience*, volume 17, issue 12, 2005.
- [18] A. Tešanović, D. Nyström, J. Hansson, and C. Norström, "Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software." In *Journal of Embedded Computing*, volume 1, issue 1, 2004.
- [19] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao, "Real-Time Component-Based Systems." In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium*, 2005.
- [20] D. C. Schmidt, "Middleware for Real-Time and Embedded Systems." In *Communications of the ACM*, volume 45, issue 6, 2002.

- [21] F. Picioroaga, A. Bechina, U. Brinkschulte, and E. Schneider, "OSA+ Real-Time Middleware, Results and Perspectives." In *Proceeding of the 7th International IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, 2004.
- [22] C. D. Gill, R. K. Cytron, and D. C. Schmidt, "Multiparadigm Scheduling for Distributed Real-Time Embedded Computing." In *Proceedings of the IEEE*, volume 91, issue 1, 2003.

Chapter 8

Evaluation of a Tool for Supporting Software Component Services in Embedded Real-Time Systems

with Ivica Crnkovic and Per Runeson

Abstract

The use of software component models has become popular in the development of desktop applications and distributed information systems. The most successful models incorporate support for run-time services of general use in their intended application domains. There has been no widespread use of such models in the development of embedded real-time systems and much research is currently directed at defining new component models for this domain. We have explored the alternative approach of extending a mainstream component model with run-time services for embedded real-time systems. A prototype tool has been developed that generates code for a number of such services. To evaluate this tool, we have conducted a multiple-case study, where four teams of students were given the same development task. Two teams were given the tool while the remaining two were not. This paper describes the design of the study and our initial analysis of the results.

8.1 Introduction

The use of software component models has become popular over the last decade, in particular in the development of desktop applications and distributed information systems. The most successful component models in these domains include *JavaBeans* [1] and *ActiveX* [2] for desktop applications and *Enterprise JavaBeans* (EJB) [3] and *COM+* [4] for information systems. In addition to basic standards for naming, interfacing, binding, etc., these models also define stan-

standardized sets of run-time services oriented towards the application domains they target. This concept is generally termed software component services [5].

Software component models have not been widely used in the development of real-time and embedded systems. It is generally assumed that this is due to the special requirements such systems have to meet, in particular with respect to timing predictability and limited use of resources. Much research has been directed towards defining new component models for real-time and embedded systems, typically focusing on relatively small and statically configured systems. Most of the published research proposes models based on source code components and targeting relatively narrow application domains. Examples of such models include *Koala* for consumer electronics [6] and *SaveCCM* for vehicle control systems [7].

An alternative approach is to strive for a component model for embedded real-time systems based on binary components and targeting a broader domain of applications, similarly to the domain targeted by a typical real-time operating system. In our previous work we have explored the possibility of using a mainstream component model as the starting point and extending it with software component services for embedded real-time systems [8]. Specifically, we have investigated the use of the *Component Object Model* (COM) [9] with the real-time operating system *Windows CE* [10] and developed a prototype tool that generates code for a number of services.

To evaluate the usefulness of the prototype tool, we have conducted a multiple-case study where four development projects were run in parallel. Two of these used the tool and two did not. Before describing the study, we briefly present the prototype tool and its rationale in Section 8.2. Section 8.3 describes the design of the case study, Section 8.4 discusses the process of data collection in more detail, and Section 8.5 presents our initial analysis of the results. Some related work is briefly reviewed in Section 8.6 while Section 8.7 presents conclusions and our plans for future work.

8.2 Background

Software component models like EJB and COM+ include support for various services that are generally useful in the domain of distributed information systems. Examples of such services include transaction control, data persistence, and security. Our focus here is on services that address common challenges in

embedded real-time systems, including logging, synchronization, and timing control. Although the sets of services are different, the principles used to provide the run-time services are similar in many respects.

A prototype tool for supporting software component services in embedded real time systems was presented in [8]. The tool adds services to COM components on Windows CE through the use of proxy objects that intercept method calls. Figure 8-1 illustrates the use of a proxy object that provides a simple logging service. The object C2 implements an interface IC2 for which one wishes to apply a logging service. A proxy object that also implements IC2 is placed between C2 and a client that uses the operations exposed through IC2. The operations implemented by the proxy forward all invocations to the corresponding operations in C2 in addition to writing information about each invocation to some logging medium.

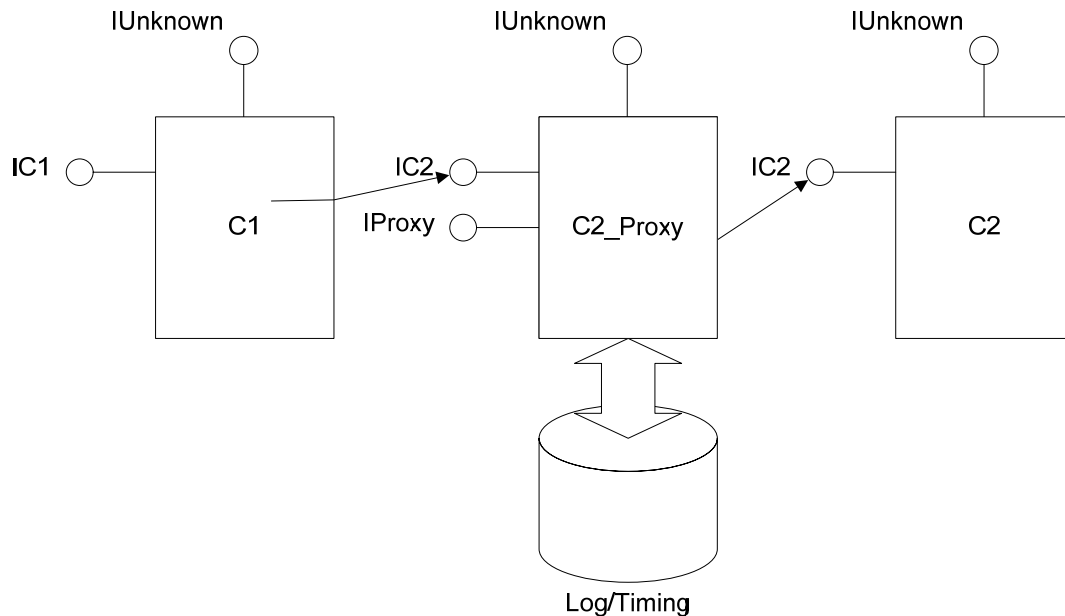


Figure 8-1 A logging service proxy

The tool takes as inputs a component specification along with specifications of desired services and generates source code for a proxy object. Component specifications may be in the form of Interface Definition Language (IDL) files or their binary equivalent Type Library (TLB) files. Desired services are either specified in a separate file using an XML-based format or in the tool's graphi-

cal user interface, described further below. Access to component source code is not required. Based on these inputs, the tool generates a complete set of files that can be used with Microsoft eMbedded Visual C++ to build a COM component implementing the proxy objects (i.e., the proxies are themselves COM objects). This process is depicted in Figure 8-2.

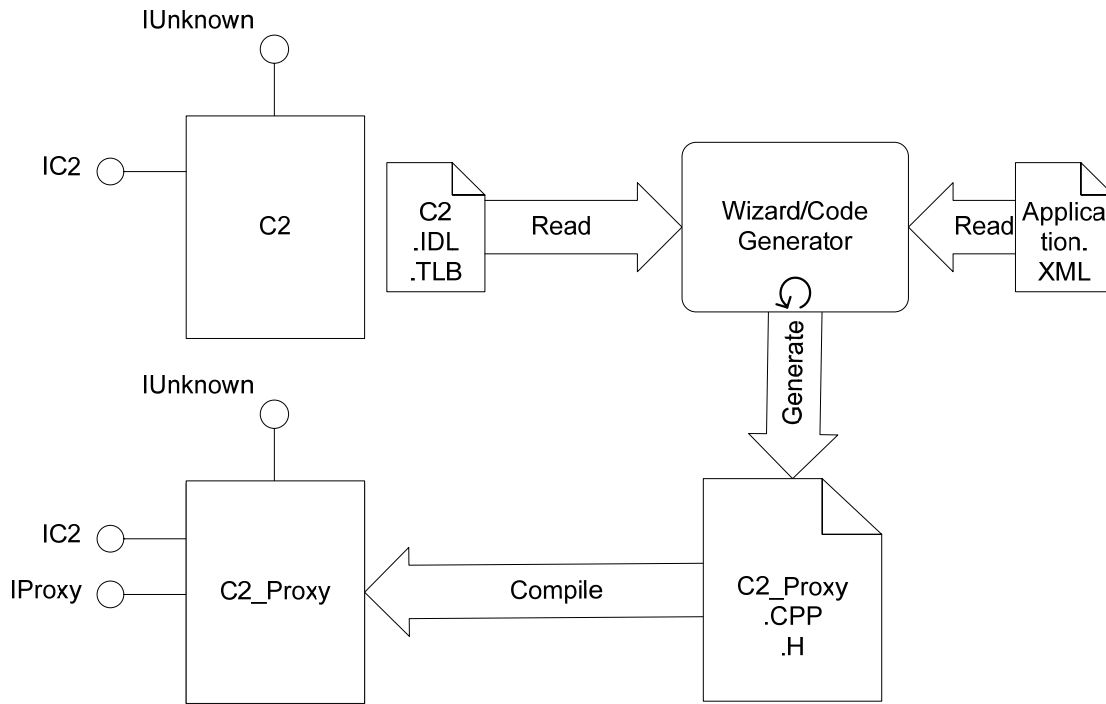


Figure 8-2 Proxy object generation

This use of proxy objects for interception is inspired by COM+. However, rather than to generate proxies at run-time, they are generated and compiled on a host computer and downloaded to the embedded system along with the application components. This process may occur when the software is initially downloaded to the system or as part of dynamic reconfiguration of a system that supports this. In the latter case, one can imagine updating or adding proxies without updating or adding any application components. The current version of the tool only generates proxy code and does not address the registration and run-time instantiation of components. This means that the client code must instantiate each proxy along with the affected COM object and set up the necessary connection between them.

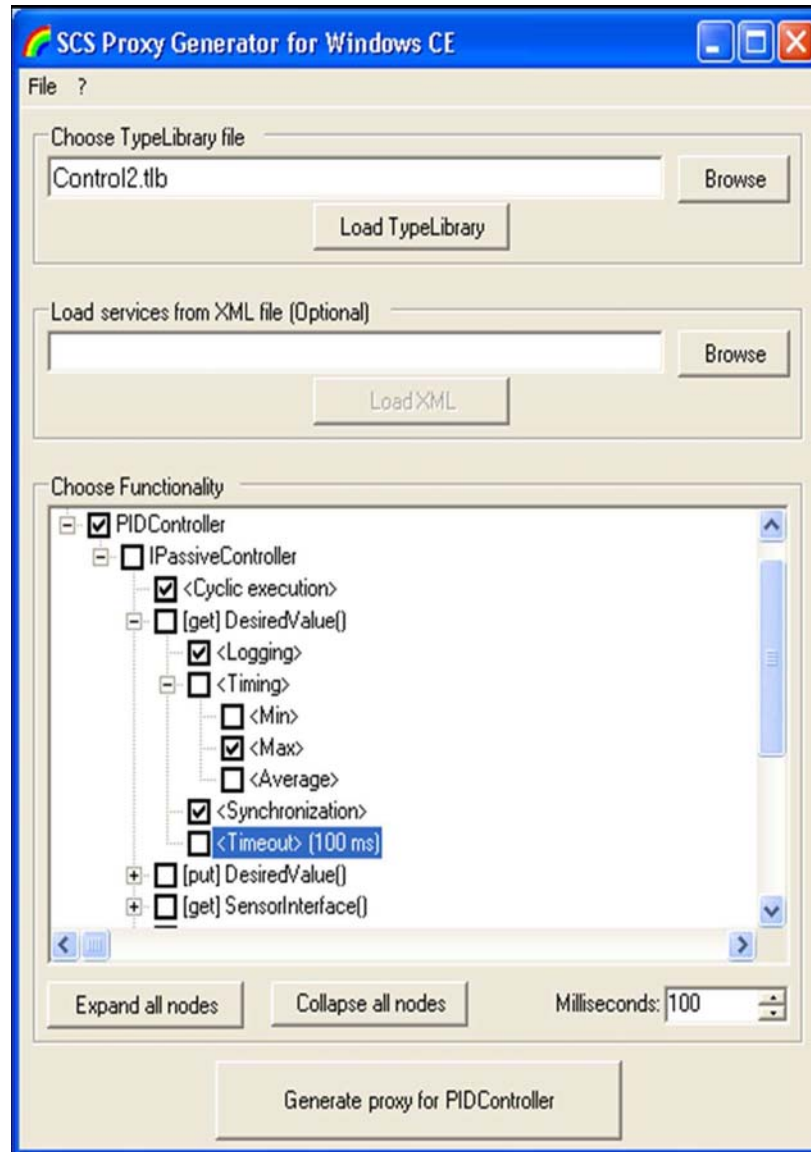


Figure 8-3 User interface of the prototype tool

Figure 8-3 shows the graphical user interface of the tool. After a TLB or IDL file has been loaded all COM classes defined in the file are listed. Checking the box to the left of a COM class causes a proxy for that class to be generated when the button at the bottom of the tool is pressed. Under each COM class, the interfaces implemented by the class is listed and, under each interface, the operations implemented by the interface. In addition, the available services are listed with their names set in brackets. Checking the box to the left of a service causes code to be generated that provides the service for the element under

which the service is listed. In the current version of the tool, a service for cyclic execution may only be specified for the `IPassiveController` interface while all other services may only be specified for individual operations. The `IPassiveController` interface is described further below.

Checking the logging service results in a proxy that logs each invocation of the affected operations. The timing service causes the proxy to measure the execution time of the operation and write it to the log at each invocation (if timing is checked but not logging, execution times will be measured but not saved). The synchronization service means that each invocation of the operation will be synchronized with all other invocations of all other operations on the proxy object for which the synchronization service is checked. The only synchronization policy currently supported is mutual exclusion.

The timeout service has a numeric parameter. When this service is selected (by clicking the name rather than the box) as in Figure 8-3, an input field marked `Milliseconds` is visible near the bottom of the tool. Checking the service results in a proxy where invocations of the operation always terminate within the specified number of milliseconds. In the case that the object behind the proxy does not complete the execution of the operation within this time, the proxy forcefully terminates the execution and returns an error code.

The cyclic execution service is particularly suited for components that implement process controllers [11]. If this service is checked, the proxy will implement an interface called `IActiveController` instead of `IPassiveController`. Both interfaces share a common set of operations for accessing control parameters, including the controller's set point. `IActiveController` includes operations for setting the period and threading priority of the cyclic execution. `IPassiveController` includes one operation for updating the controller's output and one for updating its internal state. The proxy invokes both these operations cyclically and the latter is synchronized with the operations for accessing control parameters.

8.3 Case Study Design

In order to evaluate the tool support, we launched an empirical study. The study is conducted using a multiple-case study design [12]. We prefer considering it a case study rather than an experiment, since from an experimental point of view, it is a quasi-experimental "post-test non-equivalent groups de-

sign” according to Robson’s terminology [13, pp. 133-146]. We observe four different project teams, solving the same problem with two different sets of working conditions – access to tool or not. We measure their results in quantitative terms of time consumption, problem reporting and a qualitative analysis of their technical solutions. We can not distinguish quantitatively between the effects of the tool and the teams’ capabilities, but seen as a case study, we may find indications and opinions regarding the value and contribution of the tool.

The study was conducted in the context of a project assignment for third year students in computer science that runs over 10 weeks with 50% workload – corresponding to 7.5 standard European credit units. There were 30 students, who were divided into four project teams of seven or eight members. During the early phases of the projects, some students dropped of from the course, such that the team sizes varied from five to eight members.

The assignment of the projects was to develop a component-based application to be run under Windows CE on a PC connected to two water tanks where the water level can be controlled by individual pumps. A requirement was that the software should include a component implementing a PID controller [11] able to control the pumps. The controller had to sample the current water level and update the pump voltage in a timely fashion. It should furthermore be possible to change the desired water level and control parameters during the operation of the controller in a thread-safe manner.

The detailed requirements of each project were elicited by the project teams through negotiation with a course instructor acting as customer. Thus, the requirements were not identical. Over the course of the projects, some changes in the requirements were introduced by the customer. This was in part based on each team’s achievements to avoid the task being too simple for some teams. In addition, two of the teams were given the additional requirement that they should use the prototype tool to implement multithreading, synchronization, and logging of process data,

The design used for the study is summarized as follows:

1. The subjects were divided into four teams by the course instructors with the intention of making the teams as equal as possible.
2. The team capabilities were assessed based on the earliest phases of the projects – requirements capture and user interface prototyping. We found that two teams were “strong” and two teams were “weak”.

3. All four teams were given (almost) the same task – implementation of the control system for a water tank. One strong and one weak team were given access to the tool, while the other two teams were not.
4. Data was collected during the course of the project from time sheets and weekly project reports, and the project deliverables were assessed – a project description, project final report, design description and code.

The case study teams are summarized in Table 8-1.

Table 8-1 Case study design overview

	Tool support	No tool support
Strong team	Team 1	Team 2
Weak team	Team 3	Team 4

Threats to the validity of a case study may be grouped into three categories; reactivity, respondent bias and researcher bias [13, p.172]. Reactivity means that the studied phenomenon behaves differently due to the fact that it is observed. The studied context is clearly artificial and observed in a teaching context, but all four teams are observed in the same way. Respondent bias means the risk that the respondents act based on expectations. The tool evaluation is a minor part of the study, and hence it is not clear to them what is expected. Further, the triangulation using both quantitative and qualitative measurements reduces the bias. Researcher bias means the risk that the researchers only see the positive signs pro their proposed tool. This is addressed by involving a third author for peer debriefing and negative case analysis. Triangulation also reduces researcher bias.

8.4 Data Collection

Each of the four project teams were charged with delivering a number of documents during the course of the project. In addition, the status of each project was presented orally at weekly meetings with a steering group, consisting of two course instructors for each project. Among the information collected was the number of working hours for each team member and activity. Table 8-

2 summarizes the reported working hours per activity for each group in number of hours as well as in percent of the total.

Table 8-2 Reported working hours

Activity	Team 1	Team 2	Team 3	Team 4
Project mgt.	80h 6%	37h 3%	13h 12%	12h 17%
Configuration mgt.	40h 3%	23h 2%	64h 6%	40h 6%
Requirements mgt.	400h 28%	210h 20%	78h 7%	70h 10%
Software design	280h 19%	160h 15%	131h 12%	80h 11%
Software coding	480h 33%	345h 32%	290h 26%	220h 31%
Software testing	160h 11%	160h 15%	115h 10%	131h 18%
Other activities	0h 0%	130h 12%	300h 27%	60h 8%
Total	1440h 100%	1065h 100%	1109h 100%	721h 100%

Obviously, each team was also expected to deliver a number of software components. At the end of the project, the executable software was demonstrated with the target equipment and all components – including source code – were delivered. Since the tool under evaluation is primarily intended to help with the implementation of cyclic execution and synchronization, we inspected the source code of all teams with respect to thread safety and timeliness. More specifically we studied the controller component and its relation to other components to determine if the following criteria were met:

- A timing mechanism is used to ensure that the control loop executes with the correct cycle time.
- A synchronization mechanism is used to prevent set-points and control parameters from being written by the application while they are being read by the control loop.

The properties of each team's controller component are summarized in Table 8-3 and described in more detail below.

Table 8-3 Control loop properties

	Team 1	Team 2	Team 3	Team 4
Timely	Yes	Yes	Yes	Yes
Thread safe	No	No	No	No

Team 1 had used the prototype tool and the cyclic execution service to generate a proxy that ensured correct timing of the control loop. Team 3 also used the tool to generate a proxy for the controller, but had failed to select the cyclic execution. Instead they had manually written code to execute the control loop in a separate thread, as had Teams 2 and 4, who did not use the tool at all. These three teams had all used appropriate timing mechanisms correctly.

Although Team 1 had used the tool with the cyclic execution service, they had failed to ensure thread safe execution of the control loop. As described in Section 2, the interface `IPassiveController` contains one operation for updating the controller output and one for updating its internal state, and only the latter is synchronized with other operations. Team 1's component was not thread safe, because the first operation updates the output as well as the internal state, while the implementation of the latter operation was left empty.

Of the remaining three teams, who had not used the cyclic execution service, Teams 2 and 3 had not used any synchronization mechanism at all in neither the control loop nor the operations for accessing the controller's data. Team 4 had used a mutual exclusion mechanism in the control loop but not in the other operations; the mechanism had been used in such a way that the control loops for the two water tanks were (quite unnecessarily) synchronized with each other. Consequently, none of these controller components are thread safe either.

8.5 Analysis

Based on the collected data, described in the previous section, we have performed a preliminary analysis to see whether there are any indications that the different conditions for the four project teams – i.e. use of tool or not – has resulted in any significant differences in the projects' results. The analysis is somewhat complicated by the fact that Team 3, who used the tool, failed to use

the cyclic execution service. Thus, with respect to implementation of the cyclic execution of the control loop, this team should be considered as not having used the tool, as indicated in Table 8-4.

Table 8-4 Overview of teams with respect to cyclic execution of control loop

	Tool support	No tool support
Strong team	Team 1	Team 2
Weak team		Teams 3 and 4

The reported worked hours for the four teams, summarized in Table 1, reveals no correlation between the use of the tool and the number of worked hours for the different activities. This is true both for the absolute number of hours as well as the percentages of the worked hours spent on the different activities. In particular, there are no significant differences with respect to the relative amount of work required for software coding. This can probably be attributed, at least in part, to the fact that the amount of code generated by the tool constitutes relatively small portions of the total amount of the code produced by the projects. Thus, a more detailed investigation of the working hours related to those parts of the software where the tool is most effective – i.e. the implementation of the control loop with multithreading and synchronization – would be desirable.

The properties of the four teams' controller components summarized in Table 8-3 shows a success rate of zero when it comes to thread safe execution of the control loops. Before analyzing this further, it should be pointed out that the subjects did not have prior knowledge of neither real-time systems in general nor computer-based control systems in particular. Although the necessity of using some synchronization mechanism to ensure thread safety was pointed out by the instructors at the start of the project, it seems that this was not made sufficiently clear, as at least two of the teams completely neglected to address the issue. This is not an unexpected mistake from someone without experience in concurrent system development, in particular since the error only occasionally results in failure and is likely to go undetected by testing.

Of the two teams whose control loops included some synchronization mechanism Team 1 had used the tool to generate the synchronization code. The fact that the team had only implemented the operation intended to update the con-

troller output prior to synchronization may be an indication that they too did not realize the need for synchronization, although an alternative scenario is that they were mistaken and believed that synchronization was provided. In any case, this observation shows that the way we have chosen to implement the tool to rely on two operations for updating the output and internal state respectively, is a potential source of error. This potential could easily be eliminated at the cost of removing the ability to generate the controller output in a way that is guaranteed not to be blocked by threads of lower priority. Another possible improvement may be to rename the operations from UpdateOutput and UpdateState to reflect that the former operation do not support thread safety.

Team 4 seems to have attempted to ensure thread safe execution of the control loop by using a mutual exclusion mechanism. The attempt failed because other operations that may update the controller's state did not use the same mechanism. A possible interpretation of this observation is that the team erroneously assumed that using the mechanism, called critical section in Windows CE, would prevent the thread executing the control loop from conflicting with any other threads in the system.

8.6 Related Work

The major source of inspiration for our approach and the prototype tool presented in this paper is COM+ [4], which is Microsoft's extension of their own COM model with services for distributed information systems. These services provide functionality such as transaction handling and persistent data management, which is common for applications in this domain and which is often time consuming and error prone to implement for each component. We use the same criteria for selecting which services our component model should standardize, namely that they should provide non-trivial functionality that is commonly required in the application domain. Since our component model targets a different domain than COM+, the services we have selected are different from those of COM+ as well.

We are furthermore inspired by the technique of providing services by interception. This mechanism is also used in other technologies and is sometimes called interceptors rather than proxies, e.g. in the *Common Object Request Broker Architecture* (CORBA) [14] and the *MEAD* framework for fault-tolerant real-time CORBA applications [15].

The approach presented in this paper is similar to the concept of aspects and weaving. The real-time component model *RTCOM* [16] supports weaving of functionality into components as aspects while maintaining real-time policies, e.g. execution times. However, *RTCOM* is a proprietary source code component model. Moreover, functionality is weaved in at the level of source code in *RTCOM* whereas in our approach, services are introduced at the system composition level.

Another effort to support binary software components for embedded real-time systems is the *Robocup* project [17], which builds on the aforementioned Koala model and primarily targets the consumer electronics domain. This work is similar to ours in that the component model defined as part of this project is largely based on the basic concepts of COM. Furthermore, the sequel of the project, called *Space4U* [18], also seems to use a mechanism similar to proxy objects, e.g. to support fault-tolerance.

8.7 Conclusion and Future Work

This paper describes a multiple-case study we have launched to evaluate the usefulness of a prototype tool that supports the concept of software component services in embedded real-time systems. The study is based on four parallel software development projects, where two of the project teams were given the tool. One of these only partly used the tool as intended, however, so in some important respects, three of the projects were conducted without tool support and only one with tool support.

The projects are completed and have resulted in delivery of documentation and software from each of the four teams. This paper presents our first analyses of some of this data – the reported number of working hours for different activities and the properties of the delivered software with respect to timeliness and thread safety. We have not been able to draw any conclusion from the reported working hours, except that it is desirable to study the required development effort related to certain parts of the software in more detail. The analysis of software properties has shown that the students participating in the projects were not well prepared for implementing the required functionality in a thread safe manner, neither with the support of the tool nor without it. However, we have identified possible changes to the tool that would probably make it easier to avoid such errors, even for developers without experience of multithreaded software.

In the immediate continuation of the work presented here we plan to expand upon our analysis of the differences between the four projects. In addition to the already identified task of analyzing the development effort in more detail, we expect to undertake a more comprehensive and systematic qualitative analysis of the delivered documentation and software. We also plan to launch further empirical studies to evaluate our approach for software component services and the prototype tool. For instance, it would be of great interest to investigate the use of the tool in connection with reuse of components across projects. One possibility is to conduct another study with students as participants, either as a multiple-case study again or as a controlled experiment. It would also be desirable to apply the prototype tool in an industrial case study, which would imply a lower level of replication and control but allow us to evaluate our approach in a more realistic setting. We plan to evaluate and extend the set of services supported by the tool. We hope to do this with the help of industrial partners in such domains as industrial automation, telecommunications, and vehicle control systems.

8.8 References

- [1] R. Englander, *Developing Java Beans*. O'Reilly, 1997.
- [2] D. Chappell, *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [3] R. Monson-Haefel, B. Burke, and S. Labourey, *Enterprise JavaBeans*, 4th edition. O'Reilly, 2004.
- [4] D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.
- [5] G. T. Heineman and W. T. Council (editors), *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.
- [6] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software." In *Computer*, volume 33, issue 3, 2000.
- [7] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren, "SaveCCM - A Component Model for Safety-Critical Real-Time Systems." In *Proceedings of the 30th EROMICRO Conference*, 2004.

- [8] F. Lüders, D. Flemström, I. Crnkovic, and A. Wall, "A Prototype Tool for Software Component Services in Embedded Real-Time Systems." In *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, 2006.
- [9] D. Box, *Essential COM*. Addison-Wesley, 1997.
- [10] J. Murray, *Inside Microsoft Windows CE*. Microsoft Press, 1998.
- [11] K. J. Åström and B. Wittenmark, *Computer Controlled Systems: Theory and Design*, 2nd edition. Prentice Hall, 1990.
- [12] R. K. Yin, *Case Study Research: Design and Methods*, 3rd edition. Sage Publications, 2003.
- [13] C. Robson, *Real World Research: A Resource for Social Scientists and Practitioner-researchers*, 2nd edition. Blackwell, 2002.
- [14] Object Management Group, *Common Object Request Broker Architecture: Core Specification*. OMG formal/04-03-12, 2004.
- [15] P. Narasimhan, T. A. Dumitras, A .M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava, "MEAD: Support for Real-Time Fault-Tolerant CORBA." In *Concurrency and Computation: Practice and Experience*, volume 17, issue 12, 2005.
- [16] A. Tešanović, D. Nyström, J. Hansson, and C. Norström, "Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software." In *Journal of Embedded Computing*, volume 1, issue 1, 2004.
- [17] J. Muskens, M. R. V. Chaudron, and J. J. Lukkien, "A Component Framework for Consumer Electronics Middleware." In C. Atkinson, C. Bunse, H. Gross, and C. Peper (editors.), *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Springer, 2005.
- [18] Space4U Project, "Space4U Public Home Page." <http://www.hitech-projects.com/euprojects/space4u/>, accessed 28 April 2006.

Chapter 9

Conclusion

9.1 Summary of Results

This dissertation investigates the use of a software component model based on binary components in the development of software for embedded real-time systems. Section 2.3.3 reviews a set of requirements for component models to be used in such systems and discusses how well a component model based on binary components, exemplified by COM, can satisfy these requirements. It is concluded that this model is well suited to most of the requirements, such as limited use of resources, support for standard modeling techniques, and ease of introduction. Other requirements can be satisfied by augmenting the model or putting restrictions on its use. For instance, analyzability can be supported by delivering models for this purpose along with the executable components. Similarly, white-box or testing and debugging are possible if components are built with debugging information and delivered with source code. Some requirements, such as those for reusability and maintainability, are highly dependant on the programming language and style used to implement components. Thus, these are orthogonal to the model itself, which defines a binary standard and is independent of programming language.

In addition to these requirements, an obvious requirement is that systems using the component model must exhibit predictable timing and be able to satisfy real-time constraints. Chapter 4 discusses the effect of using COM and DCOM on this ability as well as on the use of computational resources. A study of documentation leads to the conclusion that the models are not inherently incompatible with real-time requirements although their use introduces some additional sources of potential run-time errors and loss of predictability compared to traditional object-oriented programming. For instance, the dy-

dynamic binding between components that occurs transparently during object instantiation means that there is a potential that the component implementing a requested object is not present, an error that would be detected at compile-time in traditional object-oriented software. Also, the time required for instantiation can be unpredictable since it depends on whether the component is already loaded or not. These sources of potential errors and unpredictability can be handled by putting restrictions on the use of the models. For instance, system can be designed in such a way that all components are loaded during initialization. The study furthermore finds that the expected run-time overheads associated with the models are very modest. The run-time overheads of COM interfaces are the same as those typically incurred by using virtual methods in C++. The additional costs associated with using DCOM are dominated by the communication mechanisms themselves rather than the proxy and stub objects that provide location transparency.

The industrial case study reported in Chapter 5 demonstrates that a component-based software architecture, using the basic concepts of COM, can be beneficially used in the development of an industrial control system. More specifically, the study shows that the architecture effectively supports distributed development and leads to effort savings as well as quality gains related to making extensions to the software. After the study, technical management at ABB estimated that the investment in the framework was 15–20 person-years and that the savings in effort for extending the system with one new communication protocol was 1–3 person-years. The investment in the framework is dominated by the effort required for splitting the functionality into generic and protocol-specific parts rather than for adopting COM. The quality gains are related to the fact that the use of externally developed components has necessitated a well thought-through modularization of the system and that the identification of generic functionality that can be implemented in the framework has led to a simplification of protocol implementations. The performance of the system with the component-based architecture was found to be acceptable and the hard real-time requirements for scheduling of control programs were not affected, as it is not handled by the part of the system that was componentized. The additional time for instantiation of protocol handlers does not interfere with the normal execution of programs since all instances are created during program download. There are no potential run-time errors or timing unpredictability related to instantiation since static linking is used. An interesting aspect is that, while component-based software architectures are usually viewed as a way to support reuse of components across multiple

systems, this project demonstrates their usefulness in supporting integration of multiple components in a single system.

Three years after the study, a total of 15 protocol handlers have been developed and the effort estimates are still considered to be valid. The majority of the developed protocol handlers support protocols that are implemented on external communication interfaces rather than in the protocol handlers themselves. These protocol handlers may still contain a considerable amount for functionality, e.g. for managing hardware redundancy, reporting status, and performing diagnostics. The remaining protocol handlers also implement the actual protocols and communication is performed using either “dumb” external interfaces or the controller’s built-in serial or Ethernet ports. Most of the communication protocols that the control system supported already before the project have now been re-implemented as protocol handlers, which have been done in India. Thus, this effort has benefited from the architecture’s inherent support for distributed development. Due to the demonstrated usefulness of the component-based architecture for I/O and communication functions, ABB is now planning to adopt a similar approach for other types of functionality in the system. For instance, in the Control Builder, a framework to support the implementation of editors for different programming languages as independently developed components is being considered.

The framework has also been developed further to meet new requirements. As the number of protocols has grown and the majority of protocol handlers are not used in most controllers, it is no longer desirable that all protocol handlers are statically linked with the rest of the system. To address this, the system has been extended with a possibility for dynamic download of protocol handlers. This solution has been developed in-house rather than using the COM implementation available with VxWorks to avoid the additional licensing costs. This dynamism brings with it the potential for run-time errors and unpredictable timing, but this can still only occur during program download. An advantage of not using an off-the-shelf COM implementation is that it has been possible to keep the ability to link some protocol handlers statically. This is used for those protocols that are not optional and, consequently, the potential problems related to dynamism do not apply to these. Other new requirements on the framework are related to the use of the controller in safety critical systems. Effort estimates for the new developments of the framework are not available at the time of writing.

The dissertation furthermore investigates the possibility of extending the component model with software component services of general use for embedded real-time systems. Chapter 6 outlines a set of such services, including logging, synchronization, execution time measurement, invocation timeout, and cyclic execution. It presents an approach for supporting such services by the use of proxy objects, using automatic code generation based on declarative attributes, and a prototype tool that generates code for a number of services. Unlike in COM+ and the Trust4All project, described in Section 2.3.4, the code generation is performed off-line. This means that it does not use any computational resources of the target system and that the system does not need to store code for services that are not used. The prototype is evaluated by implementing an example application using a general PID controller component. Two versions of the application is implemented, one where generated proxies are used for logging, synchronization, and cyclic execution, and one where this is handled by the controller component. The amount of source code in the involved components is compared in the two versions, to provide an indication of the required development effort. The use of proxies substantially reduces the size of the controller component from 300 lines to 173 lines. On the other hand, for the executables that invoke the operations of the controller components, the source code grows from 81 to 157 lines, since the latter executable must also instantiate the proxy objects and set up the connections between them and the other objects. It is concluded that automatic generation of code for this task would be a valuable extension of the work and it is outlined how this can be quite simply achieved.

The prototype tool is evaluated further in the study reported in Chapter 7. The different versions of the example application of the previous chapter are tested on a PC running Windows CE and execution times are measured. In addition to the two component-based versions of the application, two non-component-based versions are also implemented and tested. In both of these, COM components are replaced by statically linked C++ classes. In one version, these classes are implemented with only virtual methods and, in the other, without any virtual methods. The use of virtual methods means that binding between objects occurs at run-time and operation invocation is performed via tables of function pointers. In the other version, binding is static and operations are invoked without indirection. Execution time measurements of an operation invoked repeatedly by the controller components show modest overheads related to the use of virtual methods, COM, and COM in combination with generated proxy objects. The average per invocation is approximately 7.5 nanoseconds when either virtual methods or COM is used and an additional 4

nanoseconds with the generated proxies. These measurements were performed 170 times and the variation was approximately 5% with virtual methods as well as COM and about 11% with proxies, meaning that timing is quite predictable.

Execution times were also measured during initialization of the different versions of the example application, which consists of instantiating COM or C++ objects and setting up connections between them. The average measurements were 0.4 milliseconds for the C++ application without virtual methods, 0.7 millisecond with virtual methods, one millisecond with COM, and nearly three seconds with COM and generated proxies. The study concludes that more measurements are desirable to investigate why the time required for initialization is so much larger in the last version of the application. The measurements were done as a part of Master thesis project and, after the project was completed, an inspection of the code used for testing found that the long times were caused by a flaw in the tests. New measurements with the Windows CE emulator after fixing this flaw showed that the time for initialization is of the same order of magnitude as for the other application versions.

Chapter 8 reports on a multiple-case study where the use of the prototype tool in a software development project is evaluated with respect to development efforts and software quality. The study makes use of a term project in software engineering where four teams of students were given the same software development task and only two of the teams were instructed to use the tool. Sources of evidence include reported working hours, implementations of control loops, and communication with project members. The reported working hours did not reveal any substantial difference between those teams that used the tool and those that did not. It is concluded that the part of the software for which the tool was used may be too limited for any saving in efforts to make a noticeable impact on the total and, therefore, that further studies are desirable. Concerning the implementation of control loops, it was found that only one team had made full use of the tool as intended. This team used the tool incorrectly to support cyclic execution and failed to produce a thread-safe system, as did the other three teams that hand-coded the cyclic execution. A possible modification of the tool was identified that would have prevented the incorrect use. Further studies should be performed with the modified tool and also giving the subjects clearer instructions for how to make full use of the tool. It was furthermore found that all teams successfully implemented the control loops in a timely fashion. Two teams achieved this by using the tool and the remaining two by hand-coding the timing control.

Based on an overall evaluation of the evidence, including discussions with project members, it is concluded that the approach underlying the prototype tool is promising with respect to both productivity and quality. For instance, the subjects that used the tool expressed that it was quite easy to understand and use. Making the tool even easier to use by adding automatic generation of code for instantiation and configuration of proxies and other objects was identified as a possible improvement already in Chapter 6. Given the study's lack of quantitative evidence, its most important results are the lessons it provides for designing further studies. Possible modifications to the tool were identified that should be made before new studies are conducted. Other lessons are that a system where a larger part of the software can benefit from the tool should be investigated and clearer instructions for the use of the tool should be given. A controlled experiment with smaller teams and a more limited development task is identified as an attractive option.

9.2 Research Questions Revisited

A number of research questions are formulated in Section 1.3, each of which is further decomposed into sub-questions. The first question and its sub-questions address the use of a software component model based on binary components in embedded real-time systems, in particular the use of COM and DCOM. These questions are quoted below for ease of reference.

Research Question 1

What are the advantages and liabilities of using a software component model based on binary components in the development of embedded real-time systems?

Research Question 1-1

Is it possible to use COM/DCOM in the development of software for systems with real-time constraints?

Research Question 1-2

What restrictions (if any) should be placed on the use of COM/DCOM in software for systems with real-time constraints to ensure predictability?

The following results are based on the study of COM and DCOM reported in Chapter 4:

Research Result 1-1

It is possible to use COM/DCOM in systems with real-time constraints.

Research Result 1-2

Placing restrictions on the use of COM/DCOM may be necessary to ensure predictability and satisfy real-time constraints. In particular, restrictions on when objects are instantiated and components are dynamically loaded may be necessary.

In addition to these results, which are related to the ability to meet real-time constraints, the discussion in Section 2.3.3 concludes that software component models based on binary components, such as COM, are well suited to most of the other requirements typically found in the domain of embedded real-time systems. The experiences from the industrial case study, discussed in the previous section, suggests that it may be desirable for a model targeting this domain to also allow static linking of certain components that must always be present. This is not supported by COM. The empirical study reported in Chapter 7 shows that the run-times overhead of using COM is very modest. In the particular system studied, the timing overhead per operation was found to be 7.5 nanoseconds $\pm 2.5\%$ while the memory overhead was not noticeable with the performed measurements. The absolute values for the overheads will of course depend on the underlying hardware.

The second question addresses the effects of adopting a component-based software architecture in the development of an embedded real-time system:

Research Question 2

What are the effects of adopting a component-based software architecture for an embedded real-time system?

Research Question 2-1

What are the effects on the effort required to make extension to the system?

Research Question 2-2

What are the effects on the real-time predictability of the system?

The following results are based on the case study described in Chapter 5:

Research Result 2-1

Adopting a component-based software architecture for an embedded real-time system may effectively support distributed development.

Research Result 2-2

Adopting a component-based software architecture for an embedded real-time system may reduce the effort required for making extensions to the system.

Research Result 2-3

It is possible to adopt a component-based software architecture for an embedded real-time system while maintaining real-time predictability.

Formulations like “may reduce the effort” are chosen because it would surely be possible to adopt a software architecture that was component-based but did not have the same positive effects. While demonstrating the positive effects in a single project is sufficient proof that they *may* be achieved, the scientific value of the results depends on the probability that the same advantages will result from adopting a component-based software architecture for other systems as well. It is reasonable to assume that the results can be generalized to other types of systems than industrial control systems and other types of functionality than I/O and communication. The major limitation of the results is that it only applies to extending a system with functions that have some commonality with other functions, such that these can be implemented by components complying with the same interfaces and sharing some generic functionality that can be provided by a framework. The result that real-time predictability can be maintained is furthermore only applicable to cases where those parts of the software that are subject to hard real-time constraints are not included in the componentization. The study’s observed effort savings came as a result of redesigning the software architecture and implementing a framework for providing generic functionality. This required investing some effort in the first place, and the savings were found to surpass the investment after 8–10 extensions. In general, a number of expected future extensions are required for this type of componentization to be cost-effective, but the exact number of such extension is of course difficult to determine a priori. Thus, the approach is most attractive when new extensions are expected to continue to be required for a long time.

The third question addresses the extension of a basic component model with automatically generated support for run-time services of general use for embedded real-time systems:

Research Question 3

What are the effects of using automatically generated support for software component services in the development of an embedded real-time system?

Research Question 3-1

What are the effects on the software’s size, resource usage, and predictability?

Research Question 3-2

What are the effects on the quality of the produced software?

Research Question 3-3

What are the effects on the software development effort?

The following result is based on the experiences with the prototype tool described in Chapter 6:

Research Result 3-1

The use of off-line generated COM objects to support software component services in the development of an embedded real-time system can be expected to result in at most a doubling of the size of the software.

Research Result 3-2

The use of off-line generated COM objects to support software component services in the development of an embedded real-time system may reduce the effort required for software component implementation.

The following result is based on the empirical study reported in Chapter 7:

Research Result 3-3

It is possible to use off-line generated COM objects to support software component services in the development of an embedded real-time system with predictable and only modest memory and timing overheads.

The first of these three results is based on the assumption that functionality is provided by code in either proxy objects or in other objects while housekeeping code will occur in all objects and components. Thus, the size of software that uses services will approach twice that of the equivalent software that does not use services as the size of code that provide application functionality approaches zero and is dominated by the housekeeping code. The second result is based on the fact that the use of proxy objects substantially reduced the amount of source code in components. Since it is taken from a single small example application and not based on direct effort measurements, this result should be viewed as preliminary and tested further in additional studies as discussed further below. In the third result, the formulation “It is possible” is chosen because it is of course possible to implement services in such a way that overheads are neither predictable nor modest. The possibility of achieving predictable and modest overheads is proven by demonstrating it in one system. Since it is based on execution time measurements related to object instan-

tiation and operation invocation, which are the basic ways for components to interact, generalizing the result to other systems that are developed using the same techniques is straightforward. The timing overhead observed in the study was 11.5 nanoseconds $\pm 5.5\%$ per operation invocation, including the 7.5 nanoseconds overhead of using COM. Again, the absolute value of overheads will depend on the hardware.

It is reasonable to assume that the above results are applicable to other operating systems than Windows CE while generalizing to other component models than COM is not necessarily possible. For instance, overheads will only be modest in models with modest overheads on operation invocation and the overhead on software size might be modest in a model without housekeeping code in components. Also, the results cannot be generalized to other techniques for automatically generating support for services than that of off-line generated code for proxy objects. For instance, run-time generation of proxy objects would incur more run-time overhead but no overhead on software size. Naturally, such run-time generation would require an extension of the run-time system that would counteract the effect of reduced software size.

The effects on software quality and development effort are investigated by the study reported in Chapter 8. As this study did not produce quantitative results and its main result is the identified possibilities for further studies, the following hypotheses that may be the starting point of such studies are formulated:

Research Hypothesis 3-3

The use of automatically generated support for software component services in the development of an embedded real-time system may improve the quality of the software.

Research Hypothesis 3-4

The use of automatically generated support for software component services in the development of an embedded real-time system may reduce the effort required for software development.

The first hypothesis is based on the fact that, although the only team that used automatically generated support for cyclic execution in the study did not achieve higher quality than the other teams, a possible modification of the tool was identified that would have resulted in higher quality in this case. The second hypothesis is based on the fact that the tool was considered easy to use by those participating in the study and that a possible extension of the tool to make it even easier to use has been identified.

9.3 Future Work

In this dissertation, the possibility of using a software component model based on binary components in the development of embedded real-time systems is investigated, first, by studying the documentation of COM and DCOM, which are some of the more popular models for non-real-time software. The use of a component-based component based software architecture in an industrial control system is furthermore investigated by a case-study. Since multiple-case studies are generally preferable to single-case studies, as discussed in Section 1.4, it would be desirable to study more cases. One way to strengthen the generality of the results would be to study an embedded real-time system from another domain than industrial control. For the same reason, it would be desirable to study a case where software components are directly involved in delivering functionality with hard real-time constraints. It has been noted that while .NET is increasingly being used instead of COM in the desktop and information system domains, it is not suitable for software with real-time constraints. An obvious research challenge is to make .NET usable for such systems through the use of predictable mechanisms for garbage collection etc. In the meantime, it would be helpful to study the combined use of .NET and COM in systems that include software with real-time constraints as well as software without such constraints. Empirical studies could be designed to investigate whether .NET can be used to increase the productivity of the development of those parts of systems that are not subject to real-time constraints without jeopardizing the predictability of real-time parts.

The dissertation furthermore presents an approach to software components services for embedded real-time systems and a prototype tool for supporting such services. An empirical evaluation of the tool shows that it results in predictable and only modest run-time overheads and at most a doubling of software size. Further evaluation could be conducted to obtain more detailed information, in particular about the timing overheads related to object instantiation and run-time memory overheads. Using a larger application in future evaluations would be useful to test the assumption that the proportional size overhead decreases with larger components. Several possible modifications to the tool are identified. For instance, it could be extended to also generate code for instantiation and configuration of proxy objects and other objects. A simple solution would be to let proxy objects instantiate the objects they act as proxies for, such that the client would only have to instantiate the same number of objects as when services are not used. The current version of the tool only supports generation of COM components that each provides a single COM class

implementing a proxy object. Allowing a single component to implement several proxy objects would help to reduce the overhead on software size. Another possible modification that may reduce the size overhead is to use proxy objects that are more light-weight than COM objects. To strengthen the generality of the approach, it would be desirable to use it with other operating systems than Windows CE and other components models than COM. A possible candidate is the Robocup model discussed in Section 2.3.3.

The use of the prototype tool in a development project is furthermore evaluated with respect to its effects on software quality and productivity in a multiple-case study. This evaluation did not produce qualitative evidence, and the need for further investigation has already been discussed. Prior to such investigation, it would be desirable to modify the tool. The probability of achieving higher quality could be improved by changing the tool to remove certain possibilities to use it incorrectly. Extending the tool with code generation to simplify the development of client code as described above, could raise the potential for reducing the software development effort. A lesson from the study already conducted is that it would be desirable to evaluate the tool in a project where a larger portion of the software can benefit from the automatic support for services. This is probably easier to achieve with a more limited development task, which would also allow the use of a higher number of smaller development teams. Thus, future studies could be designed as controlled experiments.

In addition to controlled studies, which can be conducted with students as subjects, it would be desirable also to evaluate the approach in industrial case studies. These two types of studies are complementary in the sense that the former allows a higher degree of control and replication while the latter provides a more realistic setting for evaluating the approach. A challenge related to testing the approach in an industrial setting is that the prototype tool, which is to a large part developed by students, cannot be expected to satisfy the quality requirements for use in industrial development projects. This is closely related to the challenge of technology transfer described at the end of this chapter. Another way to evaluate the approach with respect to its ability to meet the needs of industry is to use a survey and/or interviews within organizations developing software for embedded real-time systems to assess the usefulness of the services identified. This could also help to identify additional useful services for different application domains.

The approach to software component services presented in this dissertation intends to support the implementation of certain functionality in embedded real-time systems by standardizing a set of run-time services, much like software component models intend to support run-time interoperability by defining standards for (primarily) components and run-time environments. In addition to the challenges of identifying and implementing such services, there are many research challenges related to the impact of the approach on different software development activities. For instance, there are challenges related to modeling, specification, and documentation of component-based systems where software component services are used. Another example is compositional reasoning about such systems. Other research challenges include investigating the possible relationships between the approach and such quality attributes as testability and maintainability. A comprehensive approach to research on software component services should consider such challenges as well as implementation techniques and identification of services for different application domains.

It was noted in Section 1.5 that this dissertation provides both epistemic contributions, obtained through empirical investigations, and more practical contributions, in the form of a proposed approach to software component services for embedded real-time system and a prototype tool. A possibility for future work based on these practical contributions is technology transfer, which would involve developing the prototype into a production quality tool. This would require considerable efforts, however, as the tool would essentially have to be developed from scratch using proper quality assurance methods. If the tool were to be used for safety-critical systems in such a way that a failure of the automatically generated code might cause danger, the tool would itself be safety-critical. As noted in Section 2.3.3, the development of software for such systems is much more costly than software development in general. These potential costs strengthen the assertion made earlier that products to support software component services are most suitably provided by platform vendors, i.e. organizations that already supply such components as real-time operating systems and software development environments. This would provide more opportunities for using the products in a high number of projects and thereby regaining the costs invested in their development.