Mälardalen University Dissertations

No. 8

Predicting Quality Attributes in Component-based Software Systems

Magnus Larsson

March 2004

MÄLARDALEN UNIVERSITY
Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

Copyright © Magnus Larsson, 2004
ISBN: 91-88834-33-6
ISSN: 1651-4238
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

One of the major challenges to industry today is to provide products with high degrees of quality and functionality at low cost and short time to market. The cost and time to market requirements have quite successfully been addressed by the component-based approach. Unfortunately, satisfactory solutions for handling quality are not yet available. Hence, a still open challenge when building systems with software components is to accurately predict the quality attributes by the produced system.

Component technologies widely used in office, desktop and internet domains provide support for integration of components into a system via well-defined functional interfaces. However, the quality attributes of the final software system, such as its performance, scalability or reliability, is not easy to determine for such systems, since current component technologies lack support for managing quality. For this reasons, the component-based approach, although attractive for many reasons, is difficult to utilize in domains in which quality attributes are of primary importance.

This thesis demonstrates the possibility of developing component technologies that provide mechanisms for predicting quality attributes of software systems, given the quality attributes of their components. Moreover, a method that can be used to build prediction-enabled component technologies and validate the predictability theory is presented. The method is demonstrated by experiments and a discussion of two different attributes: latency and consistency.

There are quality attributes that cannot be predicted equally accurately, if at all, as they lack a formal specification, measurement possibilities, and as they depend on many different factors, not only on properties of components. The thesis proposes a classification of different attributes from a prediction perspective: distinguishing quality attributes that can be predict directly from component properties, from those that need more information, such as usage profile or architecture.

Having the means to reason about the qualities of a software design in the same way as one can reason about the qualities of a mechanical design is a dream of software engineers. By introducing predictability capabilities in component-based systems, this thesis is a small step towards fulfilling this dream.

“Prediction is very difficult, especially about the future”.
Niels Bohr, 1885-1962

To
Christina
Emmy, Ida, Jacob and Thea

Acknowledgements

First my greatest thanks go to professor Ivica Crnkovic who has motivated, encouraged and laughed with me during the years of study. Without the support, valuable feedback and the opportunities created by Ivica, I would not have finished this thesis. In addition I want to thank Hans Hansson for being assistant supervisor, always ready with good and precise feedback, during my years at Mälardalen University.

Love goes to my parents and parents in laws, whom have been extremely supportive in innumerous different ways. Without you I would have been torn apart between the work and the family.

For the challenging work and the sponsoring from ABB, I am grateful to my master Charlotte Brogren. The work started with Erik Gyllenswärd who made this journey possible and with him, I always have a mentor and a friend.

Many fellow colleagues have reviewed my work and shared my time and I want to thank them for their valuable feedback. They are, among others: Fredrik Ekdahl, Holger Hofmann, Esther Gelle, Stig Larsson, Frank Lüders, Christer Nortström, Kristian Sandström and Anders Wall.

During my stay at the Carnegie Mellon University/Software Engineering Institute it has been great to work with Kurt Wallnau and his team; many good ideas and laughs have been shared.

Thanks to all my friends that always listen to me, and my questions about life, the universe and everything.

Last but not least, thank you Christina for sharing my life!

Published Material

I have authored or co-authored the following publications:

Books
Building Reliable Component-Based Software Systems Artech House publisher 2002 ISBN 1-58053-327-2 Editors: Ivica Crnkovic, Magnus Larsson
Journal Articles and Book Chapters
Concerning Predictability in Dependable Component-based Systems: Classification of Quality Attributes Submitted for publication to special issue on Architecting Dependable Systems II, Lecture Notes in Computer Science, 2004, Springer Verlag Authors: Ivica Crnkovic, Magnus Larsson
Challenges of Component-based Development Journal of Software Systems, April 2002, Elsevier Science Home Authors: Ivica Crnkovic, Magnus Larsson
Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues In Databases and Information Systems, Selected Papers, Fourth International Baltic Workshop, Baltic DB&IS, pp.237-252, 2001. Kluwer Academic Publishers ISBN 0-7923-6823-1 Authors: Ivica Crnkovic, Magnus Larsson, Kung-Kiu Lau, Juliana K. Küster Filipe
Thesis
Applying Configuration Management Techniques to Component-based Systems Licentiate Thesis MRTC 00/24, December 2000 Author: Magnus Larsson

Conferences and workshops
Towards an Impact Analysis for Component Based Real-Time Product Line Architectures Euromicro Conference on Component Based Software Engineering, September 2002. Authors: Anders Wall, Magnus Larsson, Christer Norström
Using Prediction Enabled Technologies for Embedded Product Line Architectures In 5th ICSE Workshop on Component-Based Software Engineering, May 2002. Authors: Anders Wall, Magnus Larsson, Christer Norström, Ivica Crnkovic
Component-based Software Engineering - New Paradigm of Software Development, Invited talk & Invited report, MIPRO (Microprocessor systems, Process control and Information Systems) 2001 proceedings Opatija, Croatia, May 2001 Authors: Ivica Crnkovic, Magnus Larsson
Managing Complex Systems - Challenges for PDM and SCM In Software Configuration Management SCM 10, 23rd, ICSE Toronto, Canada, May 2001. Authors: Annita Persson Dahlqvist, Ivica Crnkovic, Magnus Larsson
Configuration Management for Component-based Systems In Software Configuration Management SCM 10, 23rd ICSE, Toronto, Canada, May 2001 Authors: Magnus Larsson, Ivica Crnkovic
Implementation of a Software Engineering Course for Computer Science Students In Proceedings, APSEC, Asia-Pacific Software Engineering Conference Singapore, December 2000 Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders
Component Configuration Management In ECOOP Conference, Workshop on Component Oriented Programming, Nice, France, June 2000 Authors: Magnus Larsson, Ivica Crnkovic
Software Process Measurements using Software Configuration Management In Proceedings, The 11th European Software Control and Metrics Conference Munich, Germany, May 2000. Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders
The Different Aspects of Component Based Software Engineering In Proceedings, MIPRO (Microprocessor systems, Process control and Information Systems) Conference Opatija, Croatia , May 2000. Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders
A Case Study: Demands on Component-based Development In Proceedings, 22nd International Conference of Software Engineering, Limerick, Ireland, May 2000, ACM, IEEE Authors: Ivica Crnkovic, Magnus Larsson
Development Experiences of a Component-based System In Proceedings 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems Edinburgh, Scotland, April 2000, IEEE Authors: Magnus Larsson, Ivica Crnkovic

Conferences and workshops
Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues In Proceedings, fourth IEEE international Baltic workshop on databases and information systems, Vilnius, Lithuania, January 2000. Authors: Ivica Crnkovic, Juliana K. Küster Filipe, Magnus Larsson, Kung-Kiu Lau
State of the Practice: Component-based Software Engineering Course In Proceedings, Workshop ICSE 2000 conference, 3rd International Workshop on CBSE, January 2000. Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders
Component Configuration Management for Frameworks In Proceedings, Asia-Pacific Software Engineering Conference, Workshop on Software Architecture and Components Takamatsu, Japan, December 1999. Authors: Ivica Crnkovic, Magnus Larsson, Kung-Kiu Lau
Processing Requirements by Software Configuration Management In Euromicro 99, proceedings of the 25th EUROMICRO conference, Milan, Italy, September 1999, IEEE Computer Society Authors: Ivica Crnkovic, Peter Funk, Magnus Larsson
New Challenges for Configuration Management In System Configuration Management, SCM-9, proceedings Toulouse, France, August 1999, LNCS nr: 1675, Springer Verlag Authors: Magnus Larsson, Ivica Crnkovic
Managing Standard Components in Large Software Systems In Proceedings on 2nd workshop on Component Based Software Engineering, Los Angeles, USA, May 1999. Authors: Ivica Crnkovic, Magnus Larsson

Technical reports
Predictable Assembly of Substation Automation Systems: An Experiment Report Technical Report, CMU/SEI-2002-TR-031, September 2002. Authors: Magnus Larsson, Kurt Wallnau, Scott Hissam, John Hudak, James Ivers, Mark Klein, Gabriel Moreno, Linda Northrop, Daniel Plakosh, Judith Stafford, William Wood
PDM and SCM: Similarities and differences Technical Report, Sveriges Verkstadsindustrier, January 2002 Authors: Ivica Crnkovic, Annita Persson Dahlqvist, Ulf Asklund, Magnus Larsson, Daniel Svensson
Component-Based Development - a New Approach in Software Development, Technical Report, May 2001 Authors: Ivica Crnkovic, Magnus Larsson
Component Based Software Engineering - State of the Art Technical Report, Internal, January 2000 Authors: Magnus Larsson, Ivica Crnkovic

Contents

1 Introduction

Software is at the heart of many industrial systems in use today. Added value to products is to a large extent provided by the software. Furthermore, production cost reduction is imperative and is often achieved by introducing software that permits the use of less complex hardware. Domains in which the use of software is now essential include the automotive, medical systems, process control, and manufacturing industries. Industrial products are often systems consisting of software and hardware, the software part being referred to as a *software system* incorporating many software programs or applications that must cooperate without fail.

The demands that companies in these industries must satisfy include low production costs, short time to market and high quality. The cost and time to market issue is addressed by means of the rapidly emerging *Component-based Development* (CBD) approach. Adding new functionality to existing products must have low cost and high quality. In CBD, software systems are built as an assembly of components already developed and prepared for integration. The many advantages of the CBD approach include effective management of complexity, reduced time to market, increased productivity, a greater degree of consistency, and a wider range of usability [16].

New functionality is frequently added to industrial products to meet market requirements. The new functionality is often introduced by adding components to the system. This is the reason why the component-based development approach is attractive to industry.

Different classes of components are used in different industrial settings, and different component technologies target different application domains. Industrial component technologies such as IEC 61131-3 [49] and Koala [111] are used to build applications. Other component classes are used to provide the infrastructure in systems for examples, databases, real-time operating systems, user interfaces and communication components. This diversity means that it is possible to use components at different levels in a system.

The quality aspects of software products are not, however, addressed adequately by component-based development. Component-based applications are sensitive to evolution of the system. As components and applications have separate lifecycles and different kinds of requirements, there is some risk that a component will not completely satisfy the particular requirements of certain applications or that it may have characteristics not known to the application developer. When introducing changes on the application level (changes such as updating the operating system, updating other components, changes in the application), there is a risk that the change introduced will reduce the reliability of the system due to different unforeseen mismatches [66].

Components are in general considered as black boxes with little or no information easily accessible. The information needed from the software components relates to their quality attributes. If the components' attributes are not known, the attributes of the system of which they are part of is even more difficult to reason about.

Industrial products are required to meet high functional and quality standards. Customers often pay for the functionality they require and take the quality of the product for granted. This business model influences the way software is developed. Unfortunately, software developers tend to focus on delivering the functionality without giving the quality requirements the same priority

It is possible to ensure the quality characteristics and the functionality of a product by means of extensive testing, but this is generally an expensive and not very efficient approach. A much more effective approach would be to predict the quality attributes of the product on the basis of the quality of the components of which it is constructed.

A software program computes functions with certain functional attributes. There are other attributes that we care for than the functional ones, these attributes are expressed in terms of quality attributes. A *quality attribute* is originally the characteristic of a system but today we also recognize quality attributes of software. Security, safety, performance and dependability are examples of system characteristics. Quality attributes are primarily attributes of systems which more and more are achieved by software. For example consider an industrial robot, which is an industrial product of high quality with respect to reliability. The robot, e.g. from ABB Robotics, is when delivered specified to function for 50,000 hours between failures [2]. The usual requirement is that non-scheduled downtime must be kept to an absolute minimum. The main challenges in obtaining this attribute are in software, not in hardware components.

Components have interfaces used to access the functionality of the component but it is seldom possible to access or determine the quality attributes. A problem encountered is that even if the interfaces comply with a particular standard, a component-based system may fail due to mismatch of component and application behavior. It is not sufficient to satisfy the functional requirement of an application without taking its quality behavior into consideration.

Quality attributes are often referred to as properties, non-functional or extra-functional attributes because they relate to the quality of the component and not explicitly to the component functionality [5]. The quality attributes describe the characteristics of a component. The *properties* of a component are the concrete accessible values that represent the characteristics. A component can express its quality attributes in terms of its properties. Not all quality attributes can be expressed as properties at the component level, some are better described at the application level. In this thesis, we use the terms component property and quality attribute interchangeably and in certain circumstances, we use the terms non- and extra-functional attributes all to denote the same.

It is important to reason about the quality of the product and the impact of functionality added due to a change in the system. We want to predict the quality attributes of the new or updated product and not to discover reduced quality in the late and expensive test phase of product development, or even worse at customer installations.

Predictability in the component-based development approach is the ability to reason about application behavior from the quality attributes of the components and the components' interconnections. The inability to date, to predict the application behavior from the components' properties is a serious disadvantage of the component-based development approach.

Achieving predictability calls for methods for obtaining hard data about the properties concerned. To measure many properties, it is necessary to measure other properties and to deduce, or estimate, the required values from the results of these measurements.

In a CBD approach data about properties of components and systems must be obtained. Data on the component level are used as input for predicting the system behavior. Data on the system level are required for predictability validation.

To obtain predictability of a quality attribute of a system we must have:

1. A property theory describing the theory behind the desired quality attribute;
2. A component technology that will be able to specify components in a way to be interpreted by the theory. This specification will include both the specification of the function and component properties;
3. An interpretation of the component model in the property theory;
4. Ability to measure the component and system properties.

This thesis addresses the problem of designing predictable component-based systems. Primarily, it focuses on a method of building predictable component technology supporting these systems. Further, the thesis discusses methods for validating the predictability obtained.

The remainder of this chapter contains descriptions of the research questions addressed in this thesis and presented in Section 1.1, and the methods used to obtain answers in Section 1.3. The main contributions of this thesis are presented in Section 1.4. Work directly related to the research questions is listed in Section 1.5. A description overview of the thesis organization 1.6

1.1 Key Research Questions: The Property Prediction Problem

As pointed out above, many challenges are encountered in the development of component-based systems. Of crucial importance with respect to quality attributes is the problem of obtaining predictability of the software quality before application development and use. Given the system quality attributes required, which properties of the involved components are required? If there are reasoning techniques for a particular class of quality attribute, it is of importance that these be used where possible.

The determination of the accuracy of the prediction of the quality attributes in a software system remains a problem. This and similar challenges have been addressed at a series of CBSE workshops [26,101], and particular models of certain properties have been analyzed [42,100], but so far very little work has been performed in the systematization and classification of quality attributes in relation to the above. Some system quality attributes could be derived directly from the component properties; others might require a complex model, related to the component model and the system architecture. Some system properties do not exist on the component level and might be the result of a complex combination of the system interaction with its environment, system architecture and component model. This leads to the main research question for this thesis:

Can the quality attributes of an application or a system be predicted using a component-based approach, given the attributes of the components?

This overall question may have a yes or a no answer but even if it is possible to achieve predictions of quality attributes, the complications might be too great and the cost might be too high to be practical. In order to answer the question, several sub-questions are considered.

1. *Is it possible to develop a component technology that will support reasoning about system quality attributes from component properties?*
2. *How can we verify the predictions to gain objective trust?*
3. *Which types of quality attributes are suitable for prediction?*

One possible approach to obtaining predictability is to design a component model or use one existing, that supports prediction. If it is possible to build or extend existing component models to support prediction then the question “How are prediction-enabled component models or technologies to be built? “ follows

The objective of this thesis is not to build component technologies in general but rather to develop a method that can be used to build prediction capabilities into a component technology. Such a method for designing a technology, under certain preconditions, is presented in Chapter 4, as a possible answer to this question

What kinds of component properties and technologies are suitable for prediction and how can we obtain information about component properties? For certain types of components, it might be possible to determine that prediction of certain attributes is not practical. It would be interesting to know if the definition of a component model can determine if it is suitable or not suitable for prediction. If the component can be found to be not suitable for prediction, no more effort need be spend to introduce prediction. In addition, the quality attribute itself determines the means of predictability; by their nature, certain quality attributes are easy to predict. Some are very difficult or even impossible to predict, for instance non-measurable attributes derived from different types of quality attributes. A discussion concerning these questions can be found in chapters 2 and 3

To obtain predictability of a component-based system several factors must be available: (i) a component technology that provides the means for specification of component properties, (ii) a property theory that identifies and specifies the property, and provides a means for its calculation, and (iii) interpretation rules which makes it possible to translate the component specification to the property theory.

If the factors are available and predictability is obtained, how can we verify that predictions made are? If the prediction is demonstrated valid, can we base critical design decisions on this information? This question is discussed in Chapter 7.

To be able to measure the desired attributes makes it possible to verify predictions, but does this mean that all non-measurable properties are not suitable for prediction? A series of questions regarding the ability to detect, estimate or measure properties arises and are addressed in chapters 3, 4 and 5.

1.2 Key Assumptions

The questions outlined in the previous section are dealt with in several of the following chapters. Each chapter addresses one or more of the questions and answers, either by experiment or by discussion.

Software engineering as approach – The objective of this thesis is to study the problem of designing predictable components. It attempts to reason about assembly properties in an empirical manner rather than in a formal way. Instead of taking a more formal approach, it concentrates on software engineering from a pragmatic point of view. It aims at solving problems in an empirical manner by designing and developing a model problem and then evaluation the solution.

Component technology as a base for predictability – There exist many different approaches to reason about quality attributes of a software system. One approach is to design the component technology to support the reasoning of quality attributes. We take the approach that the component model and technology are the basic parts for which predictability is required.

Model problem as research setting – The models and experiments presented in Chapter 5 and 6 are not obtained directly from industrial settings but are rather idealized for research purposes. The idealized models are used to simplify an understanding of the principles of quality attribute predictability and the problems involved. However the experiments are strongly related to industrial cases. Experiments are based on models of industrial products and in cooperation with industry

Idealized components and architecture used for validation – For the validation experiments, synthetic components have been used with a random architecture. A real world system with real components can behave in an unforeseen manner which will affect the accuracy of the predictions. Since the focus of the thesis is the process of achieving predictability rather then predictability theory, this simplification has no significant impact on the results of the thesis.

Quality attributes are measurable – The experiments address a quality attribute that is well defined and measurable but there are other attributes that are difficult to quantify and are not measurable which it may be necessary to address with the methods and technologies adopted.

Research scope is runtime attributes – In the thesis we demonstrate the methods on quality attributes visible at runtime, and in particular we have implemented a solution for the latency quality attribute. Different quality attributes may require different method. For example, lifetime attributes may require measurements during the entire life cycle of a system. These types of attributes have not been considered in the theses. Also other objectives of prediction, such as prediction of success, development time, usability and cost of development are not covered in this thesis.

1.3 Research Method

In addressing the research questions, we have adopted a research method defined by Shaw [95] (shown in Figure 1). The method begins with a real world practical problem and the ultimate goal is to provide a solution to this practical problem. The solution is obtained in a research setting – by identifying the idealized problem and producing a research result which leads to a solution (or solutions) of the idealized problem. It should then be possible, by transforming the solution of the idealized problem, to obtain a solution to the real problem. In using this research method, the validation of the result is crucial in both research and industry settings.

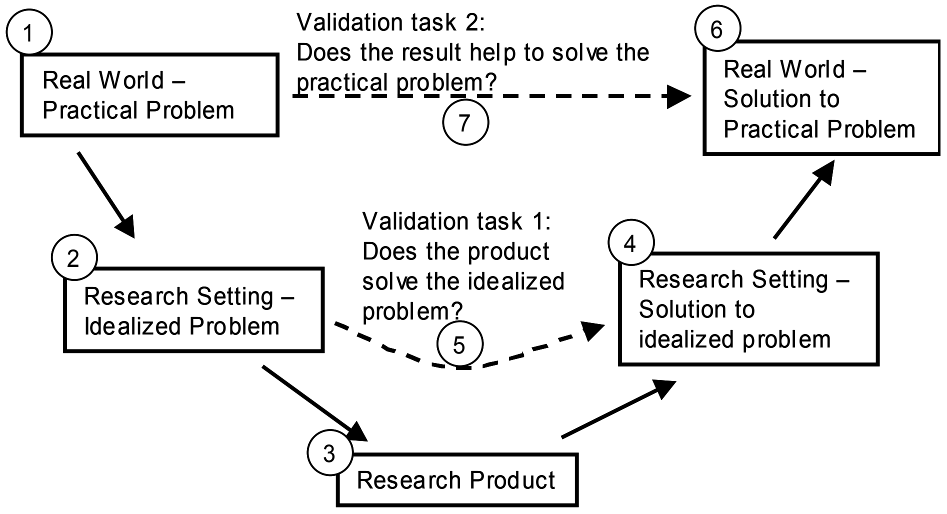

1
Real World – Practical Problem
Validation task 2: Does the result help to solve the practical problem?
7
6
Real World – Solution to Practical Problem
2
Research Setting – Idealized Problem
Validation task 1: Does the product solve the idealized problem?
5
4
Research Setting – Solution to idealized problem
3
Research Product

Figure 1: A method for doing research targeting real world problems [95]

We have considered the real world problem from case studies [23,67,69] and converted it to an idealized research problem.

There are three main phases in the research setting work: Determination of the research questions, production of the research result and validation of the result obtained. The research questions can be of different types, for example feasibility, characterization, methods, generalization or selection. Similarly the results can be obtained in different ways, for example by developing a qualitative model or a technique by building a system, developing an empirical model or an analytic model. Validation can be performed by persuasion, implementation, evaluation, analysis, or on the basis of experience.

Our approach was different for research questions of different types.

The answer to the main research question, *Can quality attributes be predicted?,* is obtained by demonstrating the feasibility of the concept with the strategic objective of implementing a system with the capability to predict a quality attribute. The research result is validated by evaluating the implementation and experimental results.

Building two component technologies demonstrates that it is possible to obtain predictability of system properties and this answers the first research sub-question, *is it possible to develop a component technology supporting reasoning about quality attributes?*. The first sub-question is similar to the main question i.e. a question of feasibility. The research result is in the form of a system implementation and validated by its evaluation through industrial related experiments.

The second sub-question, *how can we verify the predictions made?,* is answered by developing a method that can be used to verify predictions and thereby validate the results of the experiments. The research method is validated during the application of the method when building certain examples of component technologies.

The third sub-question, *which types of quality attributes are suitable for prediction?,* is addressed by developing a classification model of the quality attributes and by assembly of the results of a survey. The validation phase includes the implementation of the model for particular quality attributes and an analysis of the survey results.

When performing experiments and drawing empirical conclusions, there are many traps that can be avoided by careful preparation and implementation [34]. For examples, the experimental design should allow others to reproduce the experiment and the conclusions drawn must be objective. Is the experiment a theoretical exercise or a real world situation?

The experiments described in this thesis, target an idealized version, in a research setting, of a practical problem from the real world. Our aim is to first find a solution to the idealized problem and then to apply the results to the practical problem. The importance of experimental evaluation is emphasized in [108,109]

We have performed several research experiments in the field of component-based software engineering to prove the validity of the research results. These experiments address the present research questions in direct and indirect ways. Some have contributed to the field of prediction while others have contributed more to the component-based software engineering field in general and the fields of product data management and software configuration management in particular.

To broaden the discussion about prediction, we participated in a research project investigating predictable assembly from certifiable components with the objective of studying how prediction can be applied to an industrial problem.

1.4 Contribution

The main contributions are:

- the development of a method for building prediction-enabled component technologies;
- the development of a method for validation of such component technologies;
- a framework for classification of quality attributes from the prediction perspective.

The methods are demonstrated by several experiments relating to problems derived from industrial settings. Magnus Larsson has actively participated in the research project in which the theory was built and the experiments were made. He contributed in the project by development of the method to achieve and validate predictability. This method is presented in Chapters 4 and 5. A classification framework results from the evaluation of the experiments and is presented in Chapter 3.

The answer to the main research question is that, with certain reservations, it is possible to predict quality attributes of components when assembled. For example, it is shown in the validation phase of this thesis that it is possible to calculate the overall latency of an assembly of components knowing the execution time of the components and certain information about their quality attributes.

There is no unqualified "yes" answer to this research question, since prediction might mean imposing too many restrictions on a component technology to make it practically useful.

Component properties can be of many different types and classes and certain quality attributes cannot be predicted for different reasons. It is difficult, for example, to say that a component has certain scalability. It is not until the components are assembled that we can determine the scalability of the system or the assembly. Those properties more difficult to predict have been analyzed and the results presented in this thesis.

When we know what properties we can predict and that it is possible to incorporate prediction into a component technology, a method to accomplish this is required. From laboratory work, we have extracted a method that guides the prediction-enabled component technology builder to reach a ready-made framework for prediction. This part is described in detail in Chapters 4 and 5.

In addition to this, the contributions of this work are:

- Chapter 2:
  - Identification of reuse challenges based on a case-study performed in an industrial setting.
  - Demonstration of a component model with capabilities for predicting consistency between components.
- Chapter 3:
  - Classification of quality attributes with respect to prediction.
- Chapter 4:
  - Development of a method used to build prediction-enabled component technologies (PECT).
  - Development of a validation method for a PECT.
- Chapter 5:
  - Implementation of a PECT according to the defined method for a particular quality attribute.
  - Validation of the implemented PECT using the defined method.
- Chapter 6:
  - Demonstration of how to achieve predictability for another quality attribute.
- Chapter 7:
  - An analysis of the suitability of predictability in industrial settings.

Several publications serve as a base for this thesis. Each publication has been peer-reviewed and provides self-contained contributions. Magnus Larsson has also authored other publications related to the design of predictable software components, these being referred to directly in the text. The selected main publications for this thesis are presented below with a short summary of their respective contribution and a presentation of the specific contribution of the author.

- **Building Reliable Component-Based Software Systems**

This book, edited jointly by Ivica Crnkovic and Magnus Larsson, and published by Artech House (2002 ISBN: 1-58053-327-2) presents different issues in developing reliable component based systems.

Contribution: Magnus Larsson contributed to this book in his capacity as coeditor with Ivica Crnkovic. As coeditor, Magnus defined certain chapters and commissioned their writing by scientists, specialists in the fields concerned. He coauthored some chapters and reviewed and quality-assured others. Chapter 2 in this thesis is based on the book and on those parts, which Magnus contributed most.

Abstract: This is a book about CBSE - Component-Based Software Engineering. CBSE is the emerging discipline of the development of software components and the development of systems incorporating such components. Component-based systems are built by assembling components developed independently of the systems. To assemble components, a proprietary code, which connects the components, is usually needed. This code is often referred to as "glue code". In an ideal world of components, the assembly process is smooth and simple: the effort required to obtain the glue code is practically negligible; a system incorporating components knows everything about them - their operational interfaces and their non-functional properties and the components are exactly what the system needs; in short, components can be assembled as easily as Lego blocks. In the real world, the component-based development process is complex and often difficult; systems are built from pre-existing components when appropriate and possible and by developing a new code specific to the particular system. The system may know about the syntax of the operational interfaces of the components, but not necessarily their other properties. Developing the glue code can be costly - it may take a longer time to develop it than the components concerned. Software components are in fact much harder to assemble than Lego blocks. “Constructing software systems from components is more like having a bathtub full of Tinkertoy, Lego, Erector set, Lincoln logs, Block City, and six other incompatible kits - picking out parts that fit specific functions and expecting them to fit together" (Mary Shaw: Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging, Presentation at the Symposium on Software Reusability SSR'99). CBSE tries to make the real world as close as possible to the ideal world of component-based development. There is a long way to go to achieve this goal. In spite of many difficulties, the component-based approach has achieved remarkable success in many domains. A majority of the software programs we use everyday take advantage of component-based technologies. There are however many classes of software in which the utilization of the component-based approach is rudimentary. For these classes of software the specification of "how" is at least as important as the specification of "what". Example of these classes of systems are reliable systems, safety-, business- or mission- critical systems, (also known as dependable systems), embedded systems. The general-purpose component technologies currently available cannot cope with the non-functional (or more correctly extra-functional) requirements of such systems. These additional requirements call for new technologies, new methods and a specific approach of component-based software engineering. This book describes the basic principles, trends in research and practice of CBSE with emphasizes on dependable systems.

- **Predictable Assembly of Substation Automation Systems: An Experiment Report**

  Technical report describing the experiment about latency prediction and is published at Carnegie Mellon University the Software Engineering Institute, CMU/SEI-2002-TR-031, Authors: Kurt Wallnau, Scott Hissam, John Hudak, James Ivers, Mark Klein, Magnus Larsson, Gabriel Moreno, Linda Northrop, Daniel Plakosh, Judith Stafford, William Wood

  Contribution: As a member of the SEI PACC team, Magnus Larsson was active in the design, implementation and reporting of the experiment carried out. Magnus Larsson was responsible for the operator PECT and the PECT method, including parts of the empirical validation. The descriptions of the method and experiment implementations of the PECT are the main contribution from this report. Magnus Larsson was also involved in the design and construction of the validation infrastructure for the controller PECT. Chapters 4 and 5 are heavily based on this work.

  Abstract: The Predictable Assembly from Certifiable Components (PACC) Initiative at the Software Engineering Institute is developing methods and technologies for predictable assembly. A software development activity that builds systems from components is predictable if the runtime behavior of an assembly of components can be predicted from known properties of components and their patterns of interactions (connections), and if these predictions can be objectively validated. A component is certifiable if these known properties can be obtained or validated by independent third parties. The SEI's technical approach to PACC rests on prediction-enabled component technology (PECT). At the highest level, PECT is a scheme for systematic and repeatable integration of software component technology, software architecture, technology, and design analysis and verification technology. This report describes the results of an exploratory PECT prototype for substation automation, an application area in the domain of power generation, transmission, and management. This report focuses primarily on the methodological aspects of PECT; the prototype itself was only a means to expose and illustrate the PECT method.

- **Concerning Predictability in Dependable Component-based Systems: Classification of Quality Attributes**

  Submitted to a special issue on Architecting Dependable Systems II, Lecture Notes on Computer Science, December 2003, Springer Verlag

  Authors: Ivica Crnkovic and Magnus Larsson

  Contribution: Magnus Larsson co-authored this article together with Ivica Crnkovic and was responsible for the classification and assembly part of the article. Main contribution from this article is the classification framework for quality attributes. Chapter 3 consists of parts of the original version of this paper and additional statistical and experimental validation analysis.

  Abstract: One of the main objectives of developing component-based software systems is to enable building systems by integration of components which are perceived as black boxes. While the construction part of the integration using component interfaces is a

standard part of all component models, the prediction of the quality attributes of the component compositions is not fully developed. This decreases significantly the value of the component-based approach to building dependable systems. When it is not possible to predict the value of a particular attribute of a system based on the specifications of the system components, the system must be subjected to other procedures, often costly, to determine this value empirically. However, not all quality attributes have the same characteristics; nor is it possible to predict the behavior of all the proper-ties of a composition from the properties of the components. This chapter classifies different types of relations between the quality attributes of components and those of their compositions. The types of relations are classified according to the possibility of predicting the properties of compositions from the proper-ties of the components and a9ccording to the impacts of other factors such as system environment or software architecture. Such a classification can indicate the efforts which would be required to predict the system attributes that are essential for system dependability and in this way, the feasibility of the component-based approach in developing dependable systems.

- **Challenges of Component-based Development**

  The article presents the background to and experiences with the problems encountered in developing component-based systems. Published in Journal of Software Systems in December 2001 by Elsevier Science.

  Authors: Ivica Crnkovic and Magnus Larsson

  Contribution: Magnus Larsson describes and analysis an ABB case study and presented conclusions to be drawn from the different reuse challenges in component based software engineering. The case study and the experiences shown in this article are the main contribution. Chapter 2 is partly based on this article.

  Abstract: It is generally understood that building software systems with components has many advantages but the difficulties of this approach should not be ignored. System evolution, maintenance, migration and compatibilities are some of the challenges met with when developing a component-based software system. Since most systems evolve over time, components must be maintained or replaced. The evolution of requirements affects not only specific system functions and particular components but also component-based architecture on all levels. Increased complexity is a consequence of different components and systems having different life cycles. In component-based systems, it is easier to replace part of system with a commercial component. This process is however not straightforward and different factors such as requirements management, marketing issues, etc., must be taken into consideration. In this paper, we discuss the issues and challenges encountered when developing and using an evolving component-based software system. An industrial control system has been used as a case study.

- **Towards an Impact Analysis for Component Based Real-Time Product Line Architectures**

  The paper addresses the predictability of latency and consistency. Published in Euromicro

Conference on Component Based Software Engineering, September 2002.

Authors: Anders Wall, Magnus Larsson and Christer Norström

Contribution: Magnus Larsson applied a consistency property theory to the component model presented and provided reasoning underlying the achievement of predictability in product line architectures for embedded systems. The results from this article about component dependencies and consistencies served as a base for chapter 6.

Abstract: In this paper, we propose a method for predicting the consequences of adding new components to an existing product line in the real-time systems domain. We refer to such a prediction as an impact analysis. New components are added as new features are introduced in the product line. Adding components to a real-time system may affect the temporal correctness of the system. In our approach to product line architectures, products are constructed by assembling components. By having a prediction enabled component technology as the underlying component technology, we can predict the behavior of an assembly of components. We demonstrate our approach by an example in which temporal correctness and consistency between versions of components is predicted.

- **Applying Configuration Management Techniques to Component-based Systems**

  A Licentiate thesis covering the different issues with configuration control of software components. This thesis is published by Mälardalen Real-Time Center (MRTC) as a technical report, December 2000.

  Author: Magnus Larsson

  Contribution: Magnus Larsson wrote and presented this thesis to achieve a licentiate degree in computer science. Main contribution from this thesis is the approach presented how to analyze and keep track on dependencies in a component-based software system. Techniques from the configuration management field have been used as input for the results. This work is based on several peer-reviewed papers. Ideas from this thesis serve as base for chapter 6.

  Abstract: Building software from components, rather than writing the code from scratch has several advantages, including reduced time to market and more efficient resource usage. However, component based development without consideration of all the risks and limitations involved may give unpredictable results, such as the failure of a system when a component is used in an environment for which it was not originally designed. One of the basic problems when developing component-based systems is that it is difficult to keep track of components and their interrelationships. This is particularly problematic when upgrading components. One way to maintain control over upgrades is to use component identification and dependency analysis. These are well known techniques for managing system configurations during development, but are rarely applied in managing runtime dependencies. The main contribution of this thesis is to show how Configuration Management (CM) principles and methods can be applied to component-based systems. This thesis presents a method for analyzing dependencies between components. The method predicts the influence of a component update by identifying the components in a system and constructing a graph describing their dependencies. Knowledge of the possible influences of an update is important, since it can be used to limit the scope of testing and be a basis for evaluating the potential damage of the update. The dependency graphs can also be used to facilitate maintenance by identifying differences between configurations, e.g., making it possible to recognize any deviations from a functioning reference configuration. For evaluation of the method, a prototype tool which explores dependencies and stores them under version control has been developed. The prototype has been used for partial analysis of the Windows 2000 platform. Preliminary experiments indicate that most components have only a few dependencies. The method has thus given an indication that the analysis of the effects of component updates may not be as difficult as might be expected.

1.5 Related work

For each of the research questions there are related work performed by other research groups studying the building of a component technology with predictability capabilities.

Developing a component technology supporting reasoning for quality attributes – The most related work is conducted by Carnegie Mellon University/ Software Engineering Institute (SEI) in the area of predictable assembly from certifiable components [20,42,44,46,57,77,119]. The approach is to achieve predictability by construction. Restrictions on the constructive component model allow analytic reasoning frameworks to operate on components and applications created. We worked jointly with SEI on the predictable assembly and our contribution to that work is presented in chapters 4 and 5. Our explicit contribution is the design method, predictability evaluation and implementation of the presented approach.

The work by Schmidt et al [89] is targeting the building or use of component technologies to support prediction. The intention of this promising work is to solve real world practical problems, with prediction of reliability, through intense collaborations with industry. They uses a test bed for evaluating the prediction made, however a difference with our work is that measured reliability is not performed on randomly generated assemblies of components. One or a few defined component assemblies have been used for the validation.

The work by Dolbec and Shepard [29] proposes a model to obtain software system reliability estimates from the reliability of software components and their usage profiles. The work is related as it describes an analytic model for reliability of components but there is no evidence that implementation or measurements have been performed.

Gaining objective trust in predictions made – The second research sub-question is addressed in the work of Moreno [77]. Statistical methods, which are used in this thesis, are presented and tested in a research experimental setting.

The work of Wohlin [124] and Voas [112] outlines models and methods for certifying software components. Although this thesis does not focus on approaches to certify components, it is possible that certain of their ideas can be adopted when building a framework for certification of prediction theories or prediction- enabled component technologies.

Quality attributes suitable for predictions – Properties suitable for prediction are addressed for certain specific attributes in the area of predictability. Below is a list of different types of quality attributes that are addressed by other research groups. Each type of attribute and its corresponding approach to achieve predictability is related to our work.

Reliability of component-based software architecture is addressed in the papers [89,91,92] using parameterized contractual specifications based on state machines. The research is demonstrated with an e-commerce example and a report from experiments in a reliability test bench. Reliability depends largely on the environment of a component and this research advocates a reliability model parameterized by required component reliability in a deployment context. Other research in reliability of component-based software includes the work done by Stafford and McGregor [100]. This work applies software reliability theories to component-based software.

Scalability and performance prediction for component-based systems is addressed in [38,125] in which an empirical investigation has been conducted on several COTS middleware products. This research presents scalability metrics dependent on the performance of the system. Validation of the scalability predictions has been performed with a measurement infrastructure not unlike our approach.

Memory usage and its prediction are presented in [31]. This research uses the Koala component model [111] for experimenting with the prediction of the memory demand from component compositions. Static memory evaluation techniques are used and a method for estimating the memory usage is proposed. This work demonstrates the power of having support for prediction built into the component technology and addresses both the first and the third sub-research question.

Another attribute is freedom of deadlock. Deadlock detection between components and a method for avoiding deadlock can be found in [54]. This work presents a technique that enables synthesized connectors to prevent deadlock between components in a COM/DCOM [75] setting. In difference to our work this approach takes software architecture into consideration.

Work that addresses quality attributes at a more general level serves as a basis for reasoning about the possibility of prediction. For this there are several classifications, McCall proposes quality factors in [74], the ISO 9126 [56] standard lists certain quality attributes and Bertoa analyses quality attributes for COTS components [8]. The different classifications are a base for our classification of quality attributes from the predictability perspective discussed in Chapter 3.

There are several approaches to the question of how to obtain the properties of components and thereby the property of their assembly [5,65,87]. These works focus mainly on quality attributes as such and not on the quality attributes of software components and how to reason about assemblies of such components.

1.6 Organization of the Thesis

This thesis is organized in eight chapters of which Chapter 1 is an introduction. Each of the other chapters targets the research questions where applicable. Chapter 2 outlines the challenges of component-based software engineering and discusses three component models with respect to predictability. Chapter 3 contains a classification of different quality attributes with their possibility of their prediction. The underlying work of developing a prediction- enabled component technology is described in Chapter 4 while an example of a technology is provided in Chapter 5. Another quality attribute used to analyze the impact of installing or changing a component in a system is described in Chapter 6. The ethical aspects of basing design decisions on predicted values are discussed in Chapter 7 with a focus on the possibly catastrophic consequences of using incorrectly predicted data. The thesis is concluded in Chapter 8 with our own analysis, a sketch of future work to be performed and a summary of the lessons learnt.

2 Component-Based Software Engineering and Prediction

Both customers and suppliers have expected much from component-based development, but their expectations have not always been fulfilled. Experience has shown that component-based development requires a systematic approach to focus on the component aspects of software development [23,69]. Traditional software engineering disciplines must be adjusted to the new approach, and new procedures must be developed. *Component-based Software Engineering* (CBSE) has become recognized as a new sub-discipline of Software Engineering.

The major goals of CBSE are [24,40]:

- To provide support for the development of systems as assemblies of components;
- To support the development of components as reusable entities;
- To facilitate the maintenance and upgrading of systems by customizing and replacing their components;
- To provide software architecture that better support the desired quality attributes.

Significantly, there is nothing about component and quality attributes of the system. The building of systems from components and the building of components for different systems requires established methodologies and processes not only in relation to the development/maintenance aspects, but also to the entire component and system lifecycle including organizational, marketing, legal, and other aspects [59,105]. In addition to specific CBSE subjects such as component specification or composition and technologies, there are a number of software engineering disciplines and processes, which require specific methodologies for application in component-based development. Many of these methodologies are not yet established in practice, some not theoretically sufficiently refined. The progress of software development in the near future will depend very much on the successful establishment of CBSE and this is recognized by both industry and academia.

The remainder of this chapter is organized as follows. In the first section we present the scope and goals of this chapter. Section 2.2 identifies different challenges of CBSE and these are taken further in Section 2.3 where the different reuse challenges of components are discussed. Section 2.4 discuses components in dependable systems and the quality attributes addressed in this area. Various definitions of components and assemblies of components are outlined in Section 2.5. The Section 2.6 presents three component models in the light of prediction. The chapter is then concluded in the last section.

2.1 Scope and Goal of Chapter

The scope of this chapter is to provide a wider background for component-based software engineering and the challenges faced. To provide these challenges is also a goal of the chapter since that supports the understanding of the practical problems faced in dealing with CBSE. We performed industrial case studies and literature research to identify the challenges [23,67,69].

An additional goal is to focus, the wider scope of CBSE, on concrete definitions of components and assemblies of components. This is to provide a base for Chapter 3 and onwards. Moreover, the chapter discusses the strengths and weaknesses from a prediction perspective of three different component models.

2.2 Challenges of Component-Based Software Engineering

Component-based development (CBD) and CBSE are only in the starting phase of their expansion. CBD is recognized as a powerful new approach that will significantly improve - if not revolutionize - the development of software and software use in general. We can expect components and component-based services to be widely used by non-programmers in building their applications. Tools for building such applications by component assembly will be developed. Automatic updating of components over the Internet, present already today in many applications, will be a standard means of application improvement. Another trend we can see is the standardization of domain-specific components on the interface level. This will make it possible to build applications and system from components purchased from different vendors.

The standardization of domain-specific components requires the standardization of domain-specific processes. Widespread work on standardization in different domains is already in progress, (a typical example is the work of the OPC Foundation [81] on a standard interface to enable interoperability between automation/control applications, field systems/devices and business/office applications). Support for the exchange of information between components, applications, and systems distributed over the Internet will be further developed. Current technologies such as XML [1] are already used to exchange information over the Internet and between applications, and well integrated with several component technologies.

However, CBSE is far from being mature and it facing many challenges today [24]; some of which are summarized here

Component specification – Although this problem has been addressed since the very beginning of development of component models, there is still no consensus about what a component is, and how it should be specified. Component specification is an important issue as the basic concepts of component-based development rely on. The following research address this problem [17,71,94,96].

Component Models – Even though existing development models demonstrate powerful technologies, they have many ambiguous characteristics, they are incomplete, and they are difficult to use. The relations between system architecture and component models are not precisely defined. The basic principles of component models, their relations to software architecture and descriptions of the most commonly used models are presented in [13,32].

Component-based software lifecycle – Lifecycle of the component-based software is becoming more complex as many phases are separated in unsynchronized activities. For example, the development of components may be completely independent of the development of systems using those components. The process of *engineering requirements* is much more complex as the possible candidate components usually lack one or more features which the system requires. In addition, even if some components are individually well suited to the system, it is not obvious that they function optimally in combination with others in the system. These constraints may require another approach in requirements engineering – an analysis of the feasibility of requirements in relation to the components available and the consequent modification of requirements. As there are many uncertainties in the process of component selection there is a need for a strategy for managing risks in the component selection and evolution process [40,63]. Similarly, there are many open questions in the late phases of component-based software lifecycles. As component-based systems include components with independent lifecycles, the problem of system evolution becomes significantly more complex. There are many questions of different types not yet solved: technical issues (can a system be updated technically by replacing components?), administrative and organizational issues (which components can be updated, which components should be or must be updated?), legal issues (who is responsible for a system failure, the producer of the system or of the component?), etc. CBSE is a new approach and there is little experience yet of the maintainability of such systems. There is a risk that many such systems will be troublesome to maintain [78].

Composition predictability – Even if we assume that we can specify all the relevant attributes of components, it is not necessarily known how these attributes will determine the corresponding attributes of systems of which they are composed. The ideal approach, to derive system attributes from component attributes is still a subject of research. The question remains - "Is such derivation at all possible? Should we not concentrate on the determination of the attributes of component composites?" [121].

Trusted components and *component certification* – Because the trend is to deliver components in binary form and the component development process is outside the control of component users, questions related to component trustworthiness become of great importance. One way of classifying components is to certify them. In spite of the common belief that certification means absolute trustworthiness, it in fact only gives the results of tests performed and a description of the environment in which the tests were performed. While certification is a standard procedure in many domains, it is not yet established in software in general and especially not for software components [79,112,115].

Component configurations – Complex systems may include many components which, in turn, include other components. In many cases, compositions of components will be treated as components. As soon as we begin to work with complex structures, problems involving structure configuration appear. For example, two compositions may include the same component. Will such a component be treated as two different entities or will the system accept the component as a single instance, common to both compositions? What happens if different versions of a component are incorporated in two compositions? Which version will be selected? What happens if the different versions are not compatible? The problems of the dynamic updating of components are already known, but their solutions are still the subject of research [22,67]. One way to handle such complex systems with many components is to make use of product line architectures [11,12] to impose rules for component configurations.

Tool support – The purpose of Software Engineering is to provide practical solutions to practical problems, and the existence of appropriate tools is essential for a successful CBSE performance. Development tools, such as Visual Basic, have proved to be extremely successful, but many other tools are yet to appear – component selection and evaluation tools, component repositories and tools for managing the repositories, component test tools, component-based design tools, runtime system analysis tools, component configuration tools, etc. The objective of CBSE is to build systems from components simply and efficiently, and this can only be achieved with extensive tool support.

Managing quality attributes and CBSE – The use of CBD in safety-critical domains, real-time systems, and different process-control systems, in which the reliability requirements are especially rigorous, is particularly challenging. A major problem with CBD is the limited possibility of ensuring the quality and other non-functional attributes of the components and thus our inability to guarantee specific system attributes. The need to state information about what these attributes are of a system and their values is exactly the goal when we design predictable component based systems.

2.3 Different Reuse Challenges using Components

Reuse principles place high demands on reusable components. The components must be sufficiently general to cover the different aspects of their use. At the same time, they must be concrete and simple enough to serve a particular requirement in an efficient way. Developing a reusable component requires three to four times more resources than developing a component, which serves a particular case [104]. Some of the challenges faced trying to accomplish and achieve reuse are described below.

2.3.1 System Evolution

One of the most important factors for successful reusability, in an evolving software system, is the compatibility between different versions of the components. A component can be replaced easily or added in new parts of a system if it is compatible with its previous version. The compatibility requirements are usually essential for industrial products with long lifetime, since smooth upgrading of systems, running for many years, is required. Compatibility issues are relatively simple when changes introduced in the products are of maintenance and improvement nature only. Using appropriate test plans, including regression tests, functional compatibility can be tested to a reasonable extent. Problems that are more complicated occur when new changes introduced in a reusable component eliminate the compatibility. In such a case, additional software, which can manage both versions, must be written.

However, apart from the compatibility problem with evolving systems there are other kinds of evolution that affect long-life products:

- *Evolution of system requirements, functional and non-functional*. A consequence of a continually competitive market situation is a demand for continually improved system performance and functionality. The systems controlling and servicing business, industrial, and other processes should permanently increase the efficiency of these processes, improve the quality of the products, minimize the production and maintenance costs etc.
- *Evolution of technology used in software products*. Evolution in computer hardware and software technology is so fast that an organization manufacturing long-life and complex products must expect significant technology changes during the product life cycle. From the reliability and risk point of view, such organizations prefer not to use the latest technology, but it appears that they are forced to adopt new technology because of the demands of a highly competitive market. The unpredictable changes often made in products cause delivery delays and increased production costs. Also the use of standard components implies less control on them [19,67,68], especially if the components are updated at runtime. A software system is usually configured once only during the build-time when known and tested versions of components are used.
- *Evolution of technology related to different domains*. The advance of technology in the different fields in which software is used requires improved software. The improvements may require a completely new approach to or new functions in software.
- *Evolution of technology used for the product development*. As in the case of products themselves, new technology and tools used in the development process appear frequently on the market. Manufacturers are faced with a dilemma – to adopt the new technology and possibly improve the development process at the risk of short term higher costs (for training and migration), or to continue using the existing technology and thereby miss an opportunity to lower development costs in the long run.

- *Evolution of society*. Changes in society (for example environmental requirements, or changes in the relations between countries - as in the EU) can have a considerable impact on the demands on products (for example new standards, new currency, etc.) and on the development process (relations between employers and employees, working hours, etc.).
- *Business Changes*. We face changes in government policies, business integration processes, deregulation, etc. These changes have an impact on the nature of business, resulting, for examples, in a preference for short-term planning rather than long-term planning and more stringent time-to-market requirements.
- *Organizational Changes*. Changes in society and business have direct effects on business organizations. We can see a globalization process, more abrupt changes in business operations and a demand for structures that are more flexible and management procedures, "just-in-time" deliveries of resources, services and skills. These changes require another, fast and flexible approach to the development process.

All these changes have a direct or indirect impact on the product life cycle. The ability to adapt to these changes becomes the crucial factor in achieving business success [16].

The development of reusable components would be easier if functional requirements did not evolve during the time of development. Because of new requirements for the products, new requirements for the components will be defined. The more reusable a component is, the more demands are placed on it. A number of the requirements coming from different products, may be the same or very similar, but this is not necessarily the case for all requirements passed to the components. This means that the number of requirements of reusable components grow faster than of particular products or of a non-reusable piece of software. The relation between component requirements and the requirements from the products is expressed with the following equation (1):

$$R_C = R_{C0} + \sum a_i R_{pi} : a_i = [0,1] \tag{1}$$

R_{C0} denotes direct requirements of the component,

R_{pi} is requirements of the products P_i ,

a_i is impact factor to the component and

R_C is the total number of the component requirements.

To satisfy these requirements the components must be updated more rapidly and the new versions must be released more frequently than the products using them.

The process of the change of components is more dynamic in the early stage of the components lives. In that stage, the components are less general and cannot respond to the new requirements of the products without being changed. In later stages, their generality and adaptability increase, and the impact of the product requirements become less significant. In this period, the products benefit from combinatorial and synergy effects of components reuse. In the last stage of its life, the components are getting out-of-date, until they finally become obsolete, because of different reasons: Introduction of new techniques, new development and runtime platforms, new development paradigms, new standards, etc. There is also a higher risk that the initial component cohesion degenerates when adding many changes, which in turn requires more efforts.

This process is illustrated in Figure 2. The first graph shows the growing number of requirements for certain products and for a component being used by these products. The number of requirements of a common component grows faster in the beginning, saturates in the period [t_0– t_1], and grows again when the component features become inadequate. Some of the product requirements are satisfied with new releases of products and components, which are shown as steps on the second graph. The component implements the requirements by its releases, which normally precede the releases of the product if the requirements originated from the product requirements.

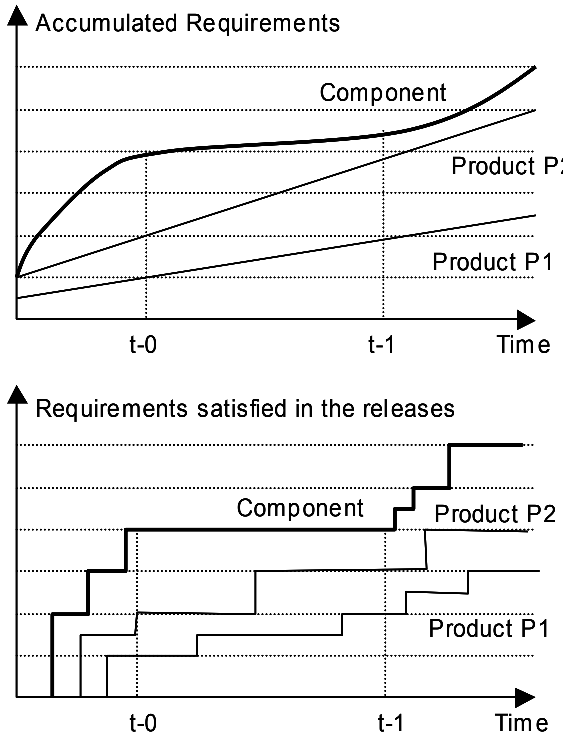


Figure 2: To satisfy the requirements the reusable component must be modified more often in the beginning of their life.

The development time for these components is significantly shorter than for products: While new versions of a product are typically released every six months, new versions of components are released as least twice as often. After several years of intensive development and improvement, the components become more stable and require less effort for new changes. In that period, the frequency of the releases is lowered, and especially the effort is significantly lower.

New efforts for further development of components appear with migration of products on different platforms and newer platforms versions. Although the functions of the products and components do not change significantly, a considerable amount of work is done on the component level.

2.3.2 Development of Reusable Components

When developing reusable components several dimensions of the development process must be considered, support for

- development of components on different platforms;
- development of different variants of components for different products;

- development and maintenance of different versions of components for different product versions;
- independent development of components and products.

To cope with these types of problems, it is not sufficient to have appropriate product architecture and component design. Development environment support is also essential. The development environment must permit an efficient work in the project - editing, compiling, building, debugging and testing. Parallel and distributed development must also be supported, because the same components are to be developed and maintained at the same time on different platforms. This requires the use of a powerful Configuration Management (CM) tool, and definition of an advanced CM-process.

The CM process support exists on two levels [21,83]. Firstly, on the source-code level, where source-code files are under version management and binary files are built. Secondly, configuration management exists on the product integration phase level. The product built must contain a consistent set of the component versions. For example, Figure 3 shows an inconsistent set of components. The product version *P1-V2* uses the component versions *C1-V2* and *C2-V2*. At the same time, the component version *C1-V2* uses the component version *C2-V1*, an older version. Integrating different versions of the same component may cause unpredictable behavior of the product.

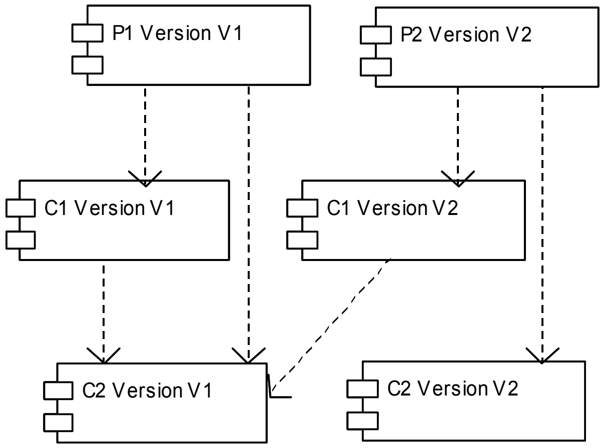


Figure 3: An example of inconsistent component integration

Another important aspect of CM in developing reusable components is Change Management. Change management keeps track of changes on the logical level, for example error reports, and manages their relations with implemented physical changes (i.e. changes of documentation, source code, etc.). Because change requests (for example functional requirements or error reports) come from different products, it is important to register information about the source of change requests. It is also important to relate a change request from one product to other products. The following questions must be answered: What impact can the implemented change have on other products? If an error appears in one product, does it appear in other products? Possible implications must be investigated, and if necessary, the users of the products concerned must be informed.

Component development independent of the products gives several advantages. The functions are broken down in smaller entities that are easier to construct, develop and maintain. The independent component development facilitates distributed development, which is common in large enterprises. Development of components independently of product or other component development introduces also a number of problems. The component and product test become more difficult. On the component level, a proper test environment must be built, which often must include a number of other components or even maybe the entire product.

Another problem is the integration and configuration problem. A situation shown on Figure 3 must be avoided. When it is about complex products, it is impossible to manually track dependencies between the components, but a tool support for checking consistency should exist to support the developer of component-based systems.

The whole development process is complex and requires organized and planned support, which is essential for efficient and successful development of reusable components and of applications using these.

2.3.3 Maintaining Reusable Components

The maintenance process is also complex, because it must be handled on different levels: On the system level, where customers report their problems, on the product level, where errors detected in a specific product version are reported, and finally on the component level, where the fault is located. The modification of the component can have an impact on other components and other products, which can lead to an explosion of new versions of different products which already exist in several versions.

One approach to minimize this cumbersome process of having diverse component configurations as the installed base is to adopt a policy of avoiding the generation of and supply of specific patches to selected customers. Instead, revised products incorporating sets of patches are generated and delivered to all customers with maintenance contracts, to keep customer installations consistent. A caveat is that this approach is not wanted in the cases where there is no need or demand for updating already working functionality, in certain critical systems it is not allowed to update working functionality unless the update can be proven correct.

The relations between components, products and systems must be carefully registered to make possible the tracing of errors on all levels. A systematic use of software configuration management has a crucial role in the maintenance process.

Another important subject is the maintenance of external components. It can be assumed that external components must be treated in the same way as internal components. All known errors and the complete error management process for internal and external components are treated in similar way. The list of known and corrected errors in external components is important for developers, product managers and service people. The cost of maintaining components, even those maintained by others must be taken into consideration.

2.4 Components in Dependable Systems

In many domains, the component based development approach has been very successful. CBD, and software reuse in general, has been extensively used for many years in desktop environments, graphical and mathematical applications. The components used in these areas are, by their nature, precisely defined and they have intuitive functionality and interfaces. On the other hand, the extra-functional characteristics and constraints are not of the highest priority since the function is what is used to solve the problem or need of the customer.

While component-based models successfully deal with functional attributes (although still being far from the ideal solutions), they provide no support for managing quality attributes of systems or components. Component-based software engineering faces two types of problems in dealing with quality attributes. The first type, one common to all software development, is the fact that there are many and often imprecise definitions of these properties. The second, specific to component-based systems, is the difficulty in relating system quality attributes to component attributes. Let us take *reliability* as an example.

An intuitive definition of the reliability of a system is the probability that a system will behave as intended**.** The formal definition of reliability is “the ability of a system or component to perform its required functions under stated conditions for a specified period of time” [51]. There are several points in this definition which must be considered. To predict or calculate the reliability of a system correctly, we must state precisely the relevant conditions under which it is to apply. This definition does not apply to system behavior in unexpected situations, but experience teaches us that problems occur mostly when a system is exposed to unexpected conditions. Uncertainty in the specification of conditions leads to uncertainty in any specification of system reliability.

To include unexpected (or expected, but not “normal”) conditions, we introduce the property *robustness*. We distinguish these two properties but cannot precisely define their relation. There are also other properties which are closely related to these two. The a*vailability* of a system is the probability that it will be up, running, and able to provide useful service at any given time. *Trustworthiness* denotes a user’s confidence that the system will behave as expected. There are systems in which the *safety* (i.e. the ability of the system to operate without catastrophic failure) and *security* (the ability of the system to protect itself against accidental or deliberate intrusion) are of the main importance. In such systems the quality attributes reliability, robustness, availability, etc. must be very precisely specified. These systems are often designated as *dependable* [99].

The specific problems and challenges involved in component-based development when dealing with quality attributes are the determination of the relations between component properties and system properties. Which properties should be considered when evaluating components, when composing them into assemblies, when testing them? Can we predict the quality attributes of a system from the specifications of its components? Let us again consider reliability as an example. The first question, which arises, is how to define the reliability of a component? It depends on specified conditions, which might be only partially defined, as these conditions are determined not only by the component itself but also by its deployment and runtime environment. The second question is how to predict the reliability of a system from the known reliabilities of its components?

Thane [107] presents a model for determining the confidence and trustworthiness of components. To acquire confidence in a component it must be supplied with a contract and be tested with a certain input. A contract specifies the functionality and the runtime conditions for which the component has been designed, i.e. assumptions about inputs, outputs and environment. If the component supplier provides such a contract, it can be used to calculate the probabilities of the occurrence of errors. Evidence based on the component’s contracts and the experience accumulated must be obtained. The environment must be considered when components are integrated in new systems; the input domain may differ considerably from the input domain for which it was tested. Confidence in a component’s reliability is only warranted when the component is used in the environment for which it is intended.

Component-based development usually decreases the development time and effort, but also the possibility of guaranteeing quality attributes. For example, the main problem when using commercial components in safety-critical systems is the system designer who has limited insight into the safety-critical properties of components. Increasing the number of test cases may decrease this uncertainty. We also need specific test methods to be applied to components. One way of performing tests is to use fault injection which can reveal the consequences of failures in components to the rest of the system [9,113,114]. As in general, the trustworthiness of commercial components is less than that of software developed in-house, we must perform tests as much as needed, but not more.

If a component is extensively tested in one configuration, do we need to repeat all the tests performed or can we assume some of the results from previous tests? Must we add new tests? This depends on the system requirements and on the system configuration. By reasoning about changes in requirements, changes in the system environment and changes in the entire environment in which the system is performing, we can to some extent, ascertain which test cases are already covered by the previous tests.

Component-based real-time systems (systems in which the correctness is also determined by time factors), and hence real-time components, must take into consideration timing constraints. Very often, these systems are dependable systems (i.e. reliable, robust, safety-critical, etc.). General-purpose component models do not provide real-time support. There are many open questions how to build component-based real time systems: what is real-time component, what are its properties, how a real-time component can be specified, etc.?

2.5 Components and Assemblies

Experience has shown that it is not easy to agree on a definition of a software component. A precise definition of a component is needed in order to understand the basics of CBSE.

It is possible to find several definitions of a component in literature, most of which fail to give an intuitive definition of a component, but rather focus on the general aspects of a component. For example, in a COM technical overview from Microsoft [75], a component is defined as "a piece of compiled software, which is offering a service". Everyone agrees that a component is a piece of software, and it obviously offers a service but this definition is too broad because, for example, even compiled libraries could be defined in this way.

The following definition by Szyperski is used in this thesis. Szyperski [104] defines a component precisely by enumerating the characteristic properties of a component: *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 party.*

The implication of these properties is as follows: For a component to be deployed independently, a clear distinction from its environment and other components is required. A component communicates with its environment through interfaces. Hence, a component must have clearly specified interfaces while the implementation must be encapsulated in the component and not directly reachable from the environment. This is what makes a component a unit of third-party deployment.

The most important feature of a component is the separation of its interfaces from its implementation. This separation is different from those which we can find in many programming languages, such as ADA or Modula-2, in which declaration is separated from implementation, or those in object-oriented programming languages in which class definitions are separated from class implementations. What distinguish the concepts in CBSE from these concepts are requirements of integration of a component into an application: Component integration and deployment should be independent of the component development lifecycle and there should be no need to recompile or re-link the application when updating with a new component.

When components are put together, using connectors or glue code, they form an *assembly*. Such assemblies can be treated as units of software and even as components that have desirable properties. Assemblies can also be abstract, i.e. they only define what components are included and their interconnections. An abstract assembly cannot be treated as a deployable component or a software unit. Similar to components, the assemblies provide particular functions or service and are characterized by quality attributes. An assembly can be part of a software application or an abstraction of such an application.

An important characteristic of a component is its visibility exclusively through its interface. An important implication of this is a need for a complete specification of a component including its functional interface, non-functional characteristics (performance, resources required, etc.), use cases, tests, etc. Unfortunately, the specification of a component is far from being complete. The current component-based technologies successfully manage only functional interfaces partially. Functional specifications are limited to syntactic lists of operations and attributes and current technologies fall short of addressing the semantics of functional properties. Further, there is no satisfactory support for specification of non-functional properties.

D'Souza and Wills in [28] define a component as a reusable part of software, which is independently developed, and can be composed with other components to build larger units. It may be adapted but may not be modified. A component can be, for example, "compiled code" without a program source (so that it may not be modified), or a part of a model and/or a design. Components are divided into two major kinds: general components and implementation components. General components are, for example, user-interface widgets dropped onto a canvas, C++ list templates, or class frameworks. Implementation components are, for example, any executable code, source code, interface specifications, or code templates.

Even though the reusability concept is familiar to us from object-oriented technologies, CBSE takes an approach to reusability different from conventional software reuse. Aoyama in [3] explains this difference as follows: Firstly, components can be composed at runtime without the need for compilation. Secondly, a component detaches its interface from its implementation, and conceals its implementation details, hence permitting composition without need to know the component implementation details. The interface of a component shall be standardized to enable reuse and allowing components to inter-operate in a predefined architecture.

Components are often understood in different ways in academia and in industry [10]. The academic view of a component is that it is a well-defined entity, often small and with easily understood functional and non-functional features. It is a black box because it has an explicit encapsulation boundary which restricts any external access. Industry follows this concept in general, and there exists many domain-specific component models, some of them used as standards (for example, *IEC 61131-3* [53], some of them developed internally by companies (for example *Koala*,[111], *AspectObjects* [39,85], or *Object Modeller* [33]). However, in many cases, industry sees a component as a large piece of software which is reusable and has a complex internal structure. It does not necessarily have well-understood interfaces, and it does not have an explicit encapsulation boundary preventing access to its internal entities. This is especially true for product-line architectures, in which different concepts and component-models are used within the same systems [23].

There are many other definitions of components. These definitions begin from the consideration of CBSE from different viewpoints and they focus on different aspects of software engineering: Different phases (in design phase: components as reusable design parts, in implementation phase: components confirmed to a specific component model, at runtime phase: binary packages, distributed components), business aspects (business components, service components, COTS components), architectural issues (UML components), etc.

What is then common to components? As previously defined, it is a unit of composition, and it must be specified in such a way that is it possible to compose it with other components and integrate it into systems in a predictable way.

To be able to describe a component completely and to ensure its correct integration, maintenance, and updating, the component should consist of 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.
- Executable code that can be coupled to the code of other components via interfaces.

As interfaces exist and are used to provide functionality it is also possible to have analytic interfaces that provide information about quality attributes as properties. These analytic interfaces can be part of the component in a form of a runtime accessible interface, or the information can be annotated to the specification of the component. Having extra information about a component is valuable since it provides a mean for reasoning about the component. To improve the reasoning capabilities about component quality, the following elements can be included in the specification of a component:

- The specification of quality characteristics which are provided and required.
- The validation code, which confirms a proposed connection to another component.
- Additional information including, documents related to the fulfilling of specification requirements, design information, use cases, etc.

A typical difficulty is how to deal with quality aspects of communication, co-operation, and co-ordination included in a componentized architecture. In a same way as components themselves, these quality attributes should be possible to compose and easy to control.

A clear separation of quality attributes gives a component more context-independence, and possibly permits reuse of the component across a wide range of contexts. Another serious problem is the *syntactic* fragile base class problem, which arises due to the incompatibility of different versions of a component. This problem should not be confused with the *semantic* fragile base class problem. If client components dependent on a component rely on a particular behavior of the objects in a component but are unaware of updates which change the inner workings of the component, the client components' function may cease, perhaps causing a system failure. This is designated the semantic fragile base class problem [76].

As we target for predictability in component-based software systems it is important to know what is required from a component model to reason about quality attributes. Can any component model be used for prediction of assembly properties?

2.6 Component Models from the Prediction Point of View

A component model defines what a component is and how components are assembled together. An implementation of a component model is designated a component technology. Component technologies include runtime and execution model while a component model defines rules on a more abstract level. There are several component technologies available and we will discuss in particular if .NET [106] and Koala [111] are suitable as a base for making predictions.

In order to enable an analysis of properties of component-based software systems, we must have means for specifying analytical properties of components and identify synchronization and communication between them. Different component models specify this to different extent. However, most of them do not treat quality attributes in an explicit manner. In the following sections we look more at three different component models from the prediction perspective.

2.6.1 Prediction in .NET

.NET is a fairly recent component technology from Microsoft that includes the notion of assembly. However, an assembly in .NET is not the same as an assembly of components; the .NET assembly is in fact the component. There is a risk of misinterpretation of the word assembly in this section and we should refer to it as a .NET assembly. There might be several reasons for calling a component an assembly, and one might be that the word component is actually overloaded with too many definitions.

A .NET *assembly* is the basic unit of deployment and versioning consisting of a manifest, a set of one or more modules, and an optional set of resources.

The .NET component technology called .NET framework and consists of a component runtime, several class libraries and technologies, such as, web services and windows forms. A central part of the .NET framework is the common language runtime (CLR) which checks, compiles and executes all assemblies in the system upon need.

.NET assemblies contain one or more classes/types and are self-describing by means of their manifest which is included in each assembly. The manifest

- establishes the identity, version, culture and digital signature of the assembly;
- defines what files are included in the assembly;
- specifies the types and resources that make up the assembly, included those that are exported from the assembly;
- itemizes the compile-time dependencies on other assemblies;
- specifies the set of permissions required for the assembly to run properly.

As .NET assemblies are self describing it is possible to obtain version information and dependencies to other assemblies. This is actually very helpful for prediction of consistency in an application using many assemblies. However, the component model allows multiple versions of the same assembly to be installed and run at the same time by the CLR. By having this capability of simultaneous execution of multiple .NET assembly versions provides the possibility to predict and achieve consistency to a higher degree.

The capability to annotate information to components/assemblies is built into .NET. There are several techniques, such as, the notion of attributes and the possibility to emit code, allowing extra proprietary information to be added. This information can describe extra-functional properties or quality attributes, that later can be utilized for reasoning about the components.

.NET does not have a notion of assembly of components and that means it is hard to build automated reasoning frameworks. A reasoning framework takes an assembly of components and looking at the parts reason about the whole. A .NET application using many .NET assemblies almost resemble an assembly of components but is actually more a set of components. The interactions between the .NET assemblies used by an application are not explicitly defined; they are implicit in the program code of the application.

The deployment model of .NET is very much a component oriented one; all .NET assemblies are deployed as single partly compiled entities into the framework. When an application is run and a component is accessed, an instance of the deployed component class in the assembly is created. All these instances are the actual components providing the application logic.

A concluding remark on the .NET component technology is that there are several extension capabilities that can be used to build prediction enabled component technologies or applications. Components can be annotated with information that was not present or thought of during design of the component. A definite advantage is that all added information on the components can be accessed programmatically at runtime allowing external programs, such as prediction tools, to access the information. A down side from the predictability perspective is that the notion of assembly of components is missing, there are only applications using components that are denoted .NET assemblies. Components are not assembled together when creating applications.

2.6.2 Prediction in Koala

Koala is a proprietary component model and architectural description language used in Philips to build consumer electronics such as televisions and DVDs [111]. Koala is designed to meet three fundamental requirements:

- It should be possible to compose components freely into products to deal with diversity in product population.
- The component technique must work in a resource-constrained environment with little computing power.
- The product architecture should be as explicit as possible to manage complexity.

Koala has a clear notion of components and assembly of components. A component is a piece of code that can interact with its environment through explicit interfaces only. Each component expresses both provided and required interfaces and can have multiple interfaces to deal with diversity. An interface resembles in many ways an interface in for instance COM, with a description of in and out parameters.

An assembly of components is called configuration and consists of a list of components and a list of connections between components. A configuration of components is shown in Figure 4 where also certain binding, gluing and switching components are shown.

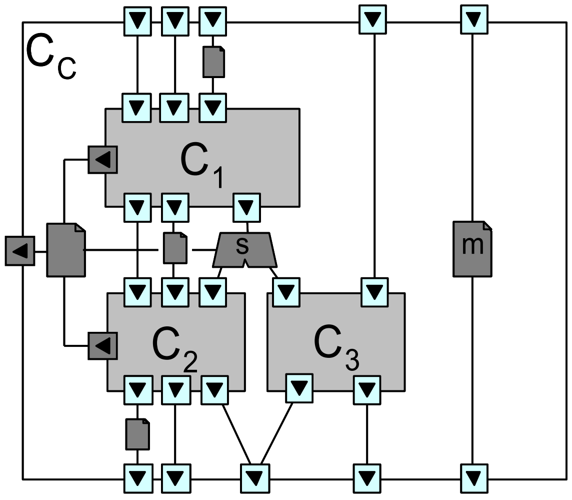


Figure 4: A Koala component example

Neither components nor configurations (assemblies) are binary and deployed as such. Components are source code with explicit interfaces and configurations are the blue print of how to connect the components. When a configuration is ready it actually compiles all components and their connections into a single executable.

The explicit architecture of components in a configuration provides very good means for creating reasoning frameworks for prediction. For instance memory consumption of the components can be predicted by analyzing the components and their interconnections [31,35]. The approach was to add a new, analytical, interface that provided the information of the memory consumption property. This interface was then used to determine the usage of resources such as RAM, ROM or stack. It was proven that this approach was very effective and accurate but the down side is that the interface had to be added to all components at design time. It was not possible to annotate already existing components with the information.

From a predictability perspective Koala has many useful features in its component model. The explicit architecture and the clear notion of components and assemblies provide very good capabilities to add reasoning frameworks. Another very important feature is that all components must communicate with the surrounding environment through explicit requires interfaces only. By having this restrictive view on components and assemblies a huge degree of determinism is achieved. The maybe only possible downside of a static component model is that it is hard to add new reasoning technologies for other quality attributes. If such technologies have to be added the component implementations must be changed, although the component model does not have to be changed. Maybe aspect oriented programming can be used for separation of concern but to our knowledge this approach has not been implemented in Koala.

2.6.3 Prediction in ReFlex

This section describes a component model that has version information built into it. The model is presented in detail in [117] and is developed as part of the ReFlex component technology [116]. This component model extends the expressiveness of port-based objects and it is presented in a simplified manner hereinafter. For a more detailed description, we refer to [116,118]. We have extended the ReFlex model with dependency information to address the consistency problem discussed in Chapter 6.

In Figure 5, the example component meta-model is depicted in UML-fashion. Components have in and out ports, which resemble the data interface. In addition, a component encapsulates services, which provide the actual functional behavior. Besides having data interfaces, defined by their ports, components in the framework have two additional interfaces, *control interface*, and *parameterization interface*. The execution of, and synchronization among components is controlled through its control interface by associating a task to the interface. A task provides a thread of execution that is defined and restricted by a set of attributes, e.g. priority, frequency. A task in the framework can be based on any task model defined by the used real-time operating system (RTOS). A task is a runtime mechanism and hence, it is a constructive part of a component. However, note that some of the attributes of a task are required when, together with some analytical properties, analyzing temporal properties of an assembly. The parameterization interface defines the points of variation of a component's behavior.

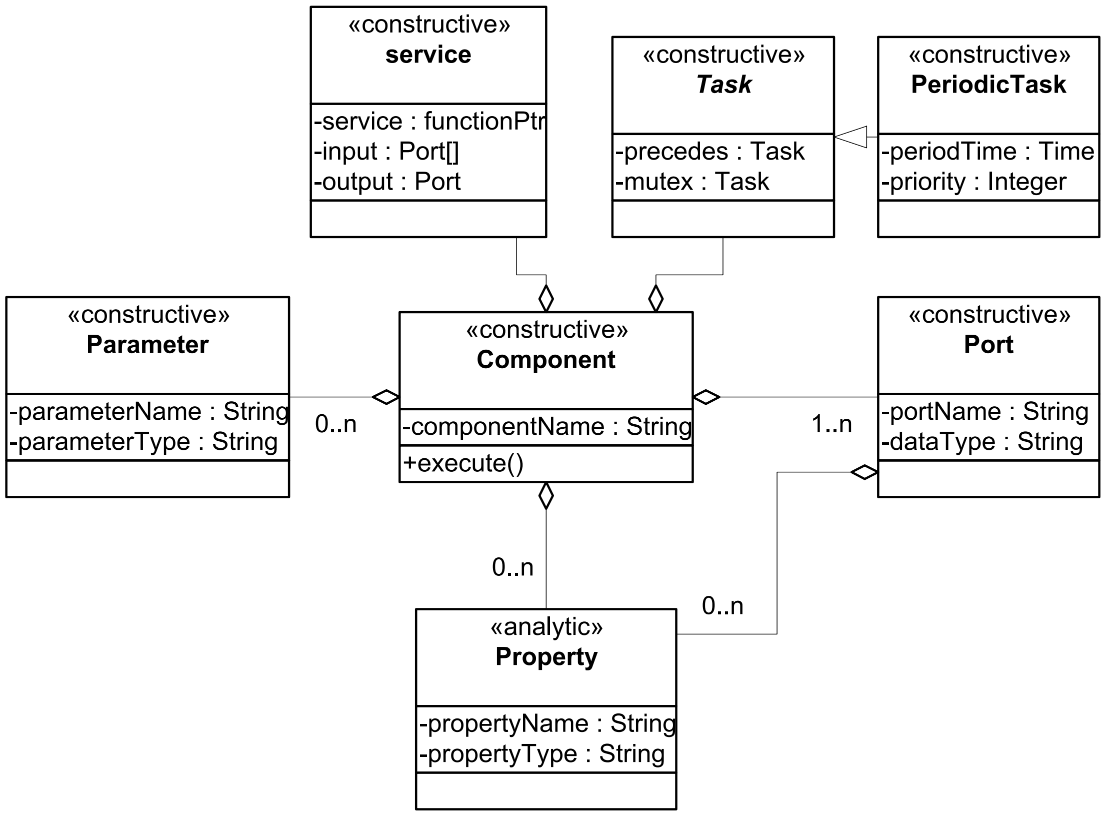


Figure 5: ReFlex – A port-based component model

The property class that is stereotyped as analytic provides the information needed by the different analyses we are interested in performing on an assembly. We will refer to such a property as an *analytical property* or quality attribute. An analytical component property usually does not have a correspondence in a component instance. A typical example of such a property, different from the consistency property, is the execution time of a service of a component. The execution time is derived from the source code, or by measurements, for the purpose of modeling and analysis of a system and has no correspondence as such in the runtime. The *analytical model* of a component is defined by its analytical properties.

For further discussions, definitions of certain terms in the component model are listed. This model emphasizes the real-time properties. A formal definition of the constructive part of the component model depicted in Figure 5 is as:

A component c is a tuple $\langle f, P, I, O, C, s_c \rangle$, where f is the service encapsulated

by c, P is the set of parameters, I is the set of in-ports, O is the set of out-ports, (2)

C is the control interface and s_c is the state of component c.

A component's state is updated by the service within a component and remains in between consecutive executions of a component.

An assembly is a specific configuration of a set of components that also defines the components interconnections. The union of all its components' states gives the state of an assembly. Formally, we define an assembly as:

An *assembly* A is a tuple $\langle C(A), R^* \rangle$, where $C(A) \subseteq C$ is the set of components in A, and R^* is the set of relations valid between $C(A)$ in A, and C is a set of all components encapsulated in the product. (3)

Note that an assembly does not necessary correspond to a product. While in some cases, we are interested in properties of the product, in some cases we may want to analyze properties of a sub-part of the complete product. In both cases, we will refer to an assembly. An assembly is only a conceptual- and analytical view of a complete product that exists for the analysis of a particular property, and has not necessarily a constructive correspondence.

In order to construct an assembly, we must be able to connect components with each other via some relation. In our definition of an assembly, we have three kinds or relations among components that belongs to the set R, *precedence*, *mutual exclusion* (mutex), and *data-flow connections*.

Precedence and mutual exclusion specify the synchronization among tasks that controls the execution of components. Formally, we define precedence and mutual exclusion as:

A precedence relation, $\rightarrow$, is a binary, transitive relation among tasks such that if $\tau_i \rightarrow \tau_j$, then τ_j may start its nth execution earliest at the end of τ_i's nth execution where $i \neq j$. (4)

A mutual exclusion relation, $\otimes$, is a binary, symmetric relation tasks such that if $\tau_i \otimes \tau_j$, then neither τ_i nor τ_j is permitted to execute while the corresponding party is executing and $i \neq j$. (5)

Besides synchronization, we can also specify data-flow relations among components in an assembly. Data-flow connections specify the data that are exchanged between components in an assembly through their ports. We define the data-flow relation as:

A data flow connection =, is a binary, anti-symmetric relation among ports on components such that if $c_i.i_x = c_j.o_y$ then c_i's in port i_x is connected to c_j's out port o_x. (6)

Figure 6 shows an example where four components have been instantiated from the model presented in Figure 5. The infrastructure in which those components will execute (the RTOS) has a scheduling policy based on fixed priorities. The task model consequently specifies the level of priority and the frequency of each task. When defining an assembly it is of importance to specify how the assembly is build. There are not only the properties of the components that determine the properties of an assembly, but also the assembly architecture. For example, in a pipe-filter architecture the data flow between components (i.e. the precedence relations) must be specified. In this example, there are definitions of the precedence property and ports connections. An analytical property is also added that specifies how many times components are supposed to be executed and for dependencies to other components.

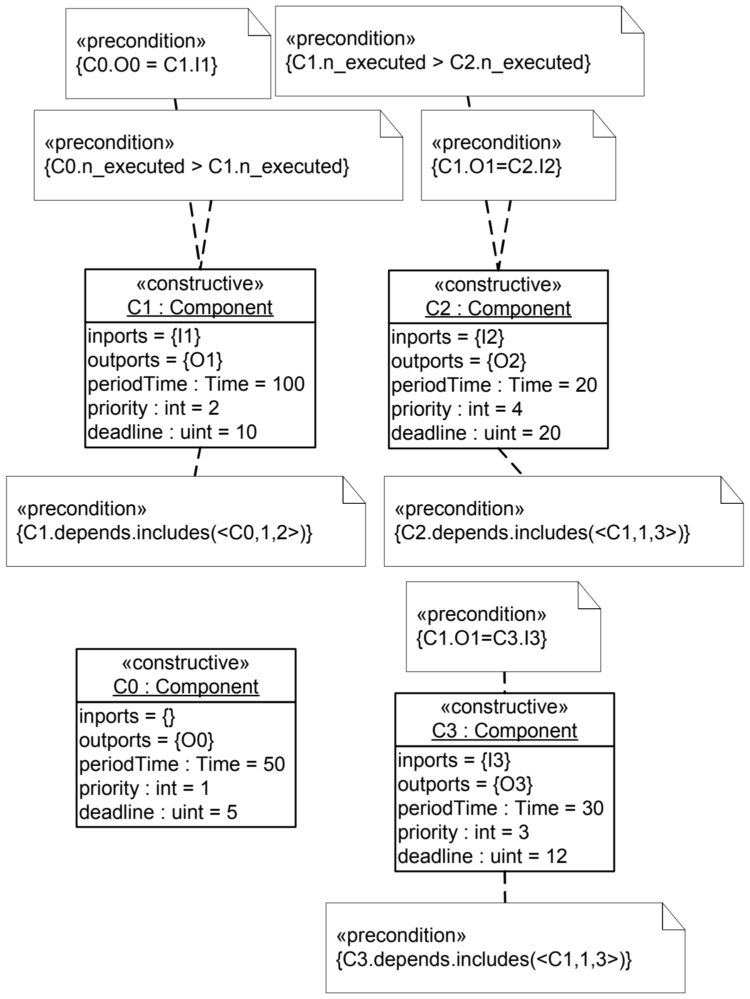


Figure 6: Four components with precedence, connections and version dependencies specified using constraints

This component model is very well suited for predictable reasoning about performance and consistency quality attributes. This model does not tie the implementation of analytical information to the components. It is up to the designers that implement the component technology to decide where and how to annotate components.

2.7 Summary and Conclusion

In this chapter we identify and discuss different challenges of component-based software engineering, based on a case-study performed in an industrial setting and literature search. The chapter provides a deeper discussion about the challenges of reusing components facing system evolution, development and maintenance. Components in dependable systems and their quality attributes are discussed in addition.

Three different component models .NET, Koala and ReFlex have been presented with strengths and weaknesses from a prediction perspective. In addition, a component model with capabilities for predicting consistency between components is defined and presented. This model will later be used as an example in Chapter 6 to illustrate reasoning about consistency between component versions and variants in an assembly.

3 Classification of Quality Attributes

The focus for the various existing component technologies is on providing mechanisms that support component deployment into a system through the component interface. In the domains in which these technologies are widely used, the quality attributes have not been of primary interest and have not been explicitly addressed in the component technologies; they have instead been treated separately from the applied component-based technologies.

In many other domains, for example dependable systems, component-based development is utilized to a lesser degree for a number of different reasons. One is the difficulty of implementing the same component technologies in different domains because of different system constraints. Another reason is the unclear distinction between system components which include both hardware and software parts and software components which may be encapsulated in system components or distributed through several system components.

Finally, an important reason is the inability of component-based technologies to deal with quality attributes as required in these domains. For dependable systems, a number of quality attributes are at least as important as the functions they provide, and the development efforts related to them are most often greater than the efforts related to the implementation of particular functions. If the advantages of component-based technologies are limited to the functional domain only and cannot be utilized in the domain of quality attributes, or, even worse, introduce difficulties in the management of quality attributes, these technologies cannot be successfully utilized in the development of dependable systems.

It is interesting to know which quality attributes are suitable for prediction. In addition it is interesting to know other capabilities of quality attributes such as how to compose attributes and how to quantify them. This chapter lists the basic conditions that each quality attribute has from the prediction perspective.

The chapter is organized as follows: The first section sets the scope and goal of the chapter. A short introduction is given in Section3.2. Section 3.3 identifies the types of attributes according to the principles for predicting the properties of component assemblies. Section 3.4 discusses the possibility of defining component composition in a recursive way. Section 3.5 lists different quality attributes and classifies them according to the compositional principles discussed in the previous Sections. A survey of certain quality attributes and their classification is presented in Section 3.6. Finally, Section 3.7 exemplifies the reasoning by showing quality attributes for safety-critical systems and a conclusion is presented in Section 3.8.

3.1 Scope and Goal of Chapter

In this chapter, our intention is to demonstrate the diversity of quality attributes and the different assumptions which should be considered for predicting system quality attributes from the properties of the components involved. These properties can be classified according to the ability of component-based technologies to specify them and provide methods for expressing their compositions, and the ability to predict the properties of component assemblies.

An assumption is that quality attributes of a component might be expressed with component properties. In many cases the attributes are the same as the properties.

The goal of the chapter is to present a classification framework for quality attributes. The classification framework is tried out on certain quality attributes by careful consideration and a small survey.

3.2 Quality Attributes in Component-based Systems

Some of the main advantages of component-based development are reusability, higher abstraction level and separation of the system development process from the component development process. These advantages have, however, implications related to other aspects of software and system development. The final success of the utilization of CBD depends not only on its advantages but also on these implications – the degree to which they are positive and negative. Since for dependable systems, particular quality attributes are of the greatest importance, a question which arises is to what extent does CBD influence the achievement of these properties: CBD can introduce new difficulties, it can be irrelevant for those properties, or can have a positive effect. For this reason, it is of interest to analyze the ability of CBD to cope with requirements related to quality attributes.

Component-based software engineering (CBSE) faces two types of problems in dealing with quality attributes. The first, common to all software development, is the fact that the quality attributes are often imprecisely defined or very difficult to measure. The second, specific to component-based systems, is the difficulty of relating system properties to component properties. In CBD, one requirement is that components should be selected and integrated in an automatic and efficient way. This goal is achieved for the functional part; components are selected and integrated through their interfaces. The question is if a similar approach can be applied to quality attributes.

For component-based systems, crucial questions in relation to quality attributes are the following:

- Given the system quality attributes required**,** which types of properties are required of the components?
- Given a set of component properties, which system properties are predictable?
- How can the quality attributes of a system be accurately predicted, from the quality attributes of components which are determined with a certain accuracy.
- To which extent, and under which constraints are the emerging system properties (i.e. the system properties non-existent on the component level) determined by the component properties?

These and similar questions have been addressed at a series of CBSE workshops [27], and particular models of certain properties have been analyzed [77,100], but so far very little work has been done in the systematization and classification of quality attributes in accordance with the questions above. Although there are other classifications of quality attributes such as [30,55,87,103], these have not considered the predictability aspect of the quality attributes.

Some system properties can be derived directly from the component properties; others might require a complex model, related to the component model and the system architecture. Some system properties, such as safety, do not exist on the component level and might be the result of a complex combination of the system interaction with its environment, system architecture and component model.

3.3 Classification of Properties

A great number of quality attributes are encountered in Software engineering. They are classified in many different ways, frequently in a non-orthogonal manner. One example of classification is related to the system life cycle: Runtime properties (visible and measurable during the program execution) and life-cycle properties (those that characterize different phases in a development and maintenance process).

The classification we consider here is related to composability. We classify properties according to the principles applied in deriving the system properties from the properties of the components involved. Instead of the term "system", we will use a generic term *Assembly (A)* which simply denotes a set of interacting components. Some properties, however, cannot be related only to an assembly, but are explicitly related to the entire system and its interaction with the environment. In such cases we refer to a *System (S)*.

According to composition principles, we can distinguish the following types of properties:

a. *Directly composable properties.* A property of an assembly that is a function of and only of the same property of the components involved.

b. *Architecture-related properties*. A property of an assembly which is a function of the same property of the components and of the software architecture.

c. *Derived properties.* A property of an assembly which depends on several different properties of the components.

d. *Usage-depended properties.* A property of an assembly which is determined by its usage profile.

e. *System environment context properties.* A property which is determined by other properties and by the state of the system environment.

Let us discuss these cases and give examples in the following subsections.

a. Directly composable properties

Definition: A directly composable property of an assembly is a function of, and only of the same property of the components.

$$\begin{aligned}
&P = \text{property}, A = \text{assembly}, c = \text{component} \\
&A = \{c_i \mid 1 \le i \le n\} \\
&P(A) = f(P(c_1), P(c_2), \ldots, P(c_n))
\end{aligned} \tag{7}$$

Note that the property of the assembly is the same as the component property. Further, the component technology is not explicitly specified in the relation (7). However, it is obvious that the function *f* itself is dependent on the technology since the mechanisms to assemble components are provided by the component technology.

An example of a property of this type is the static memory size of a component or an assembly, this is also known as the memory footprint. The simplest composition model is the calculation of the static memory of an assembly as the sum of the memories used by each component:

$$M(A) = \sum_{i=1}^{n} M(c_i)$$
$$M = \text{memory size}, A = \text{assembly}, c_i = \text{components} \qquad \textbf{(8)}$$

The function $M(c_i)$ is different for different technologies. For example in the case of the separation of composition time from runtime which is usually used in embedded systems, $M(c_i)$ will be a constant, possibly parameterized by configuration factors. In such cases, the static memory size of an assembly will be a constant. A more complicated model can be found in the Koala component model [111], in which additional parameters, such as size of glue code, interface parameterization and diversity are taken into account (i.e. the parameters determined by the component technology used).

The equation (8) is also valid for a dynamic memory, with the difference that $M(c_i)$ is not a constant, but a function which may depend on the usage profile. When using a particular technology, this function may be limited or budgeted. In such a case, the total amount of memory can be calculated.

$$M(A) \le \sum_{i=1}^{n} M_{\max}(c_i) \qquad \textbf{(9)}$$

The properties of this type can be calculated directly from the component properties and the particular technology. There are no other assumptions and therefore these properties are the easiest to specify and calculate. This does not mean that the composition functions are easy or even possible to express formally. However, the fact that the property is visible on component and assembly level, and that the assembly property is dependent only on the component properties simplifies the prediction procedure. This is valid either for measurements or for identifying restrictions, which will make it possible to reason about the composition.

b. Architecture-related Properties

Definition: An architecture-related property of an assembly is a function of the same property of the components and of the software architecture.

$$A = \{c_i | 1 \le i \le n\}$$
$$P(A) = f(P(c_1), P(c_2), \ldots, P(c_n), SA) \qquad \textbf{(10)}$$
$$SA = \text{software architectureture} \subseteq A \times A$$

In this case, the assembly properties depend not only on the component properties but also on the architectural structure. The software architecture is often used as a means for improving particular properties without changing the component properties. These types of properties can be tuned by different architectural solutions or variations.

One example of such an attribute is reliability, which depends a lot on redundant fault tolerant architectures. Another example of such a property is a performance predictability model for J2EE (Java 2 Platform, Enterprise Edition) application. A typical application implemented in this technology would be a distributed web-based application in which the variability in scalability is achieved by it being possible to add new clients and new computational (business) components to the server as illustrated in Figure 7. To achieve concurrency the components are executed in different threads. A possible extension variation of this architecture is the possibility to include several nodes with web servers and business applications.

The performance of the system shown in the Figure 7 is related to the number of clients, the number of server components, and the network utilization, which will not be considered here in our example. A typical requirement for such applications is the performance and scalability, i.e. the dependencies between the performance and number of clients and active business components.

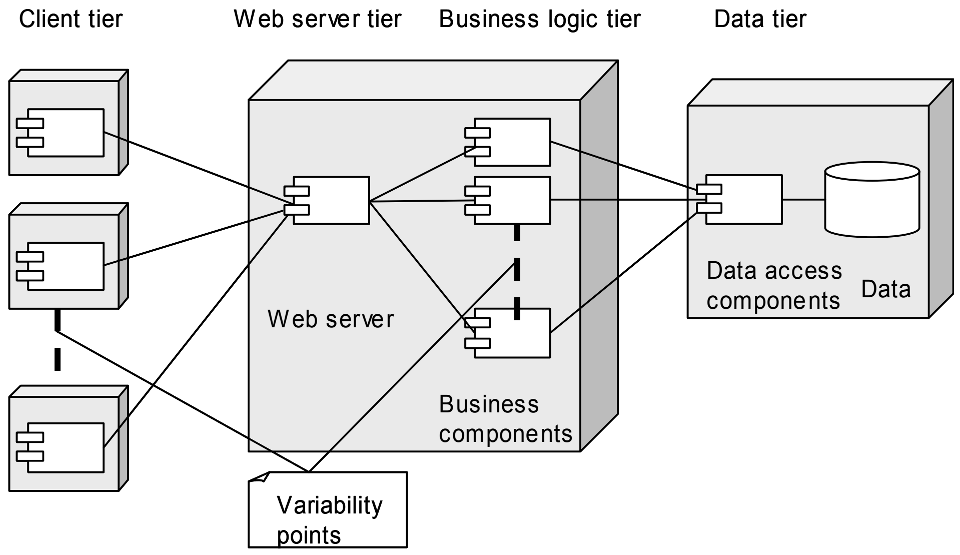


Figure 7: A typical multi-tier architecture with client and servers variability points affecting the performance quality attribute

According to [58,125] the time per transaction T/N expressed in (11) depends on several factors related to the system architecture: The first factor comes from the concurrent requests that compete for service from the server component. This includes the network bandwidth and underlying transport mechanisms. The second factor describes a case in which accepted requests compete for a thread to execute the business components. The third factor results from concurrent access to the database by the concurrent server threads.

The first factor is proportional to the number of clients, the second to the number of clients and inversely proportional to the number of threads (i.e. number of components on the server) and the third factor is proportional to the number of threads.

$$T / N = ax + b\frac{x}{y} + cy$$

$$T / N = \text{execution time per transaction} \qquad (11)$$

$$x = \text{number of clients};\ y = \text{number of components}$$

$$a, b, c = \text{proportional factors for a particular implementation}$$

The form of the equation shows that it is possible to calculate the optimal number of threads in relation to the number of clients to achieve a minimum response time per transaction.

c. Derived Properties

Definition: A derived property of an assembly is a property that depends on several different properties of the components.

$$A = \{c_i | 1 \leq i \leq n\}$$

$$P(A) = f\begin{pmatrix} P_1(c_1), P_1(c_2), \cdots, P_1(c_n), \\ P_2(c_1), P_2(c_2), \cdots, P_2(c_n), \\ \vdots \\ P_k(c_1), P_k(c_2), \cdots, P_k(c_n) \end{pmatrix} \qquad (12)$$

$$P = \text{assembly property}$$

$$P_1 \ldots P_k = \text{component properties}$$

In the same way that a function of an assembly is more than the sum of the component functions, there are properties that are the result of the composition of different component properties.

An example of such a property in a real-time system is the end-to-end deadline (a maximal response time) that is a function of different component properties, such as worst-case execution time (WCET) and execution period as shown in the following example. Let us consider real-time port-based component models with provided and required interfaces and interfaces to an underlying operating system or I/O devices, as discussed in [24,43,117]. In these models, components are implemented as tasks, parts of a task or a set of tasks. An assembly consisting of two components, where every component is realized as a task is shown on Figure 8. Each basic component includes properties such as WCET and execution period. A composition of this simple model is achieved by connecting ports and identifying provided and required interfaces.

The question is whether we can calculate the WCET for an assembly of components executing with different periods. In a case in which the execution periods are the same, this would be possible. In a case in which these periods are different, we cannot specify WCET of the assembly, but we can specify an end-to-end deadline and a period. An end-to end deadline is the maximum time interval between the start of the first component in an assembly and the finish of the last component in the assembly. The assembly period will be a number to which the components periods are divisors.

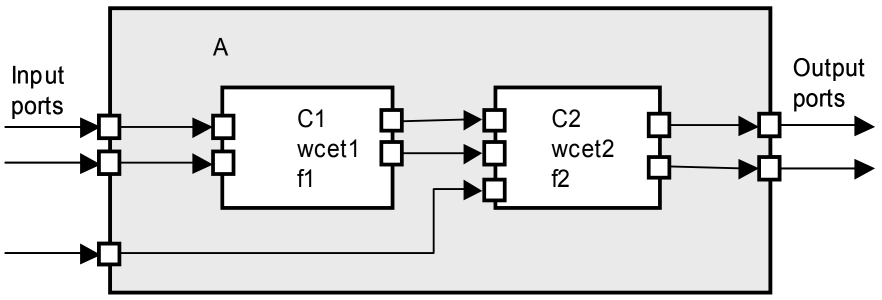


Figure 8: Composition of port-based components

Emerging properties, i.e. properties that are pertinent on a system (or an assembly) level but are not visible on the component level are of special interest in this category. For such properties, the major challenge is to identify the properties of the components that have impact on them.

d. Usage-dependent Properties

Definition: A Usage-dependent property of an assembly is a property which is determined by its usage profile.

$$\begin{aligned}&P(A,U_k) = f(P(c_i,U'_{i,k})) : \; i,k \in N\\&P = \text{property for a particular usage profile}\\&U_k = \text{assembly usage profile}\\&U'_{i,k} = \text{component usage profile for}\\&\text{component } c_i \text{ resulting from applying } U_k \text{ to the assembly}\end{aligned} \tag{13}$$

The quality attributes of an assembly and consequently of a system depends not only on the internal properties of the components and their composition but also on the particular use of the system. A usage profile U_k which determines a particular property P_k must be transformed to the usage profile $U'_{i,k}$ to determine the properties of the components.

Properties of this type introduce particular problems as they depend on the use of the system. This means that the component developers must predict as far as possible the use of the component in different systems – which may not yet exist. A second problem is to transfer the usage profile from the assembly (or from the system) to the component. Even if the usage profile on the assembly level (U_k) is specified, the usage profile for the components ($U'_{i,k}$) is not easily determined, especially when the assembly (and the system) configuration is not known.

A particular problem with this type of property is the limited possibility of reusing measured and derived properties. If the usage profile is changed, the properties must be re-calculated or re-measured. An example of such a property is reliability which is based on a usage profile. The question arising here is the possibility of reusing previous specifications of the property [24]. The first thought would be that this is possible if the domain of the new usage profile is a sub-domain of an old usage profile. In this case, the value of a property will be within the range of possible values of the property for the old usage profile, the local maximum and minimum value being in the range of values for the old usage profile (see Figure 9).

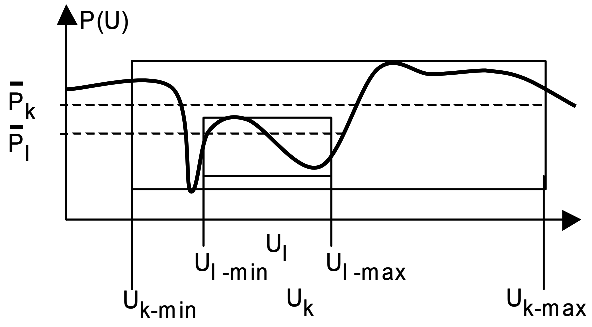


Figure 9: Property for different usage profiles

If the new requirements are equal of or less stringent than the old requirements, we can use the property value from the old usage profile. This means, for example, that we do not need to measure the component properties.

$$U_l \subseteq U_k \Rightarrow P_{k-\min}(A,U_k) \le P_l(A,U_l) \le P_{k-\max}(A,U_k) \qquad \textbf{(14)}$$

In a case in which a property is expressed as a statistical value (such as a mean value), the property value in a particular interval can be changed in an unwanted direction. Figure 9 illustrates an example in which the mean value of the property *P(U)* in the interval *[$U_{l\text{-min}}$, $U_{l\text{-max}}$]* is lower than in the entire interval *[$U_{k\text{-min}}$, $U_{k\text{-max}}$]*, although the minimum value is higher. For certain properties (such as availability) in certain domains (for example, multimedia with streaming of video files), the average plays a more important role than min or max values.

e. System Environment Context Properties

Definition: A System Environment Context property is a property which is determined by other properties and by the state of the system environment.

$$\begin{aligned}
&P_k(A,U_k,E_l) = \\
&f(P_k(c_1,U'_{1,k}),P_k(c_2,U'_{2,k}),\ldots,P_k(c_i,U'_{i,k}),E_l): \quad i,k,l \in N \\
&A = \{c_i \mid 1 \le i \le n\} \\
&U_k = \text{Assembly usage profile for property } k\,; \\
&E_l = \text{Environment context} \\
&U'_{i,k} = \text{Component usage profile}
\end{aligned} \qquad \textbf{(15)}$$

The property depends not only on the system property determined by the usage profile, but also on the environment in which the system is used. An example of such a property is safety. As the safety property is related to the potential catastrophe, it is obvious that in different circumstances, the same property may have different degrees of safety even for the same usage profile. We can argue that these properties are out of the scope of the predictable assembly, but as such properties are also dependent on component properties, this relation is important. The analysis approach for such properties is opposite to the composition; the system environment and the system properties define the requirements for component properties.

A system can exhibit numerous properties and certainly not all of them have the same characteristics; some are easy to perceive and measure while others are very difficult to analyze, or measure (for instance administrability). Analyzable properties, which can be measured, are potential candidates for automatic reasoning about the quality of a system. Properties that depend on the environment in which a system is deployed are generally hard to derive from the component properties.

3.4 Assemblies and Systems

We have used the generic term “assembly” for a set of integrated components. In the previous section, it is shown that a specification of some properties should distinguish between an assembly and a system. A system is in general much more than a set of components; a system is a set of software programs and their inter-dependencies deployed on computer hardware. A system is usually seen as a complete program or many programs and computers working together to provide a function and interacting with its environment. In several models the assembly is interpreted by a runtime framework that instantiates and executes the application or system. Very seldom is an assembly of components an application or a system of its own, a tool or environment that can interpret run the assembly must be provided.

Nevertheless, a software system may consist of a set of assemblies, which turns out to be a set of components. Several questions arise when composing assemblies: Can the composed assemblies be treated as components in the new assembly, or are they treated in the new assembly as a set of the original components loosing the assembly identity. An ideal situation would be to have a means of using a hierarchical and recursive model which permits the same reasoning on all levels of the hierarchy. In most of the existing component-based technologies, this is not possible to achieve.

There are two kinds of assemblies supported by existing component technologies. The first is the first order assembly, which is not treated as a component in the component model. This type of assembly is merely a set of components integrated together, creating an application or a part of an application. In this case, an assembly is seen as a virtual boundary of the component set and not as a separate entity. The second type of assembly is hierarchical which means that the assembly, created from components, is treated as a new component inside the component model.

There are different criteria which must be satisfied if an assembly is to be treated as a component. The basic criteria are the following:

- Operational (construction) interface;
- Component deployment;
- Component quality attributes.

Operational Interface: The operational interface is the simplest part of the hierarchical composition. A component model should provide a means of constructing an interface for the assembly from the integrated components. Several component technologies provide support for managing assemblies. One example is port-based architectures for components [24]. In such models, components have specified source and sink ports which are attached to the ports of other components. It is easy to imagine that the result of the composition of these components can be treated as a new component. Although it is important that the new interface of the assembly must be specified and implemented in the same manner as those of the components themselves, the assembly is not otherwise defined in the same component model.

Some component models provide support in creating assemblies on the architecture definition language (ADL) level. Other component models use composition languages that have composition and component notations. The Koala component model [111] is one example that includes an ADL and programming language which describes the components and connections between them. Koala, however, never creates the real assemblies, source code components are used instead and an application/assembly is generated by the description in ADL, i.e. there are no binary components as in, for instance, the .NET technology. Another example is COM which, by means of its aggregate facility, provides support by bundling several components into a new one. This new component does not have a binary representation but from the users point of view it behaves as any other COM component.

Component deployment: To obtain independent deployment of components, the components must be packaged independently and in a uniform manner conforming to the component model. In the case of an assembly of independent components, it is not easy to package the complete assembly and to treat it as a component because there is no component type which matches the assembly. Every component has a component type or unique identifier specified in a type repository. For instance, COM uses the windows registry to store the component identifiers and the defined interface identifiers.

The component deployment and the component instantiation are two separate activities; the component type is deployed and a component is subsequently instantiated from the type. This approach differs from hardware components for which the actual component is wired together with other hardware components onto the board. The hardware components are deployed and instantiated at the same time.

Once installed, the software component can be instantiated many times. To instantiate an assembly, is to instantiate components which might be already deployed. This fact makes it impossible to treat assemblies as independent units of deployment.

In the Koala component model [111], a new component, which may include other components, can be specified, so in that respect there are hierarchical assemblies of components. When a complete system of components is designed, Koala uses a compiler which, given the description of the assembly, produces the actual binary output. This output is however not treated as a component in the model, as it is the final product. The model was designed to produce applications, not to produce new binary components. New components are added to the system in the form of source code, and component specifications.

Component quality attributes: With respect to quality attributes and hierarchical composition, one may ask if it is possible to treat quality attributes of assemblies in the same way as quality attributes of components. The question is partially answered in the first section – it depends on the type of the property. In many cases, the property and the way to use it in compositional reasoning will be the same.

The way to obtain the property value might be different. Theoretically, the property value of an assembly can be derived from the component properties. In this way, a property of an assembly of assemblies will be a composition of assembly and component property functions. For example, the properties of type (a) from the Section 3.3 will be derived in the following way:

$$\begin{aligned} &P(A_a) = f(P(A_1), P(A_2), \ldots, P(A_i)) : i \in N \\ &A_a = \{A_1, A_2, \ldots, A_i \mid 1 \le i \le n\} \\ &P = property, A_a = assembly\ of\ assemblies \end{aligned} \tag{16}$$

For the memory consumption case in equation (8), we have:

$$M(A_a) = \sum_{i=1}^{k} M(A_i) = \sum_{i=1}^{k} \sum_{j=1}^{n} M(c_{ij}) \tag{17}$$

For emerging properties, it is in general not possible to make recursive definitions. The same is valid for component properties which are not relevant on the assembly level.

We can conclude that the recursive compositions of components are very restricted, which is understandable from the definition of a component. A component is defined as a unit of composition and treated as a black box. Internal structure or internal composition can completely differ from the assembly or system structure and composition of components.

3.5 Composition of Quality Attributes

This section outlines several different quality attributes and explains how they are classified according to the five types outlined in the first section.

Quality attributes can be complex and at a higher level or they can be more tangible and concrete. The attributes at higher level are, for example, dependability or performance while tangible attributes could be, for example, memory footprint, scalability or availability. These higher-level attributes are harder to classify in one category only as they consist of several more tangible attributes. Apart from the five categories presented in this section, there are other means of classification. For instance, attributes can be placed in two classes, namely, whether they can be observed at runtime or not. Preiss et al show such a mapping of quality attributes from the runtime observable perspective [87].

To demonstrate that there are different quality attributes in the various categories we have assembled a table (Table 1) showing the classification of the attributes. We did the classification of quality attributes that are mainly taken from [103]. A short description of each attribute is to be found in the appendix (Table 21) and [4,7,60,87]. According to [55] certain attributes grouped together constitute attributes at a higher level of abstraction. Many of these attributes are vaguely defined, and we have in some cases slightly modified them to achieve better consistency. Some inconsistencies remain, for example, “maintainability” is defined as a quality attribute as well as a higher level attribute. Such inconsistencies are the result of having different requirements, priorities, development processes and focus when performing the categorization.

Table 1 shows the dominant type of composition and possibly another related composition type. For each attribute, there is one prevailing character of composition, marked with “xx”, which has the greatest impact on the particular attribute. Other related composition types, not as dominant but still clearly visible, are marked in the table with a single “x”.

When identifying the most important types of compositions for each quality attribute we have asked the following questions with the possible answers no, yes, or yes significantly:

a. Directly composable attributes - Is it possible to analyze this assembly quality attribute given the same quality attribute of the components involved?

b. Architecture Related attributes - Is it possible to analyze this assembly quality attribute given the assembly software architecture and the same quality attribute of the components involved?

c. Derived attributes - Is it possible to analyze this assembly quality attribute from several different component attributes of the components involved?

d. Usage-dependent attributes - Is it necessary to know the usage profile of the assembly to analyze this quality attribute?

e. System environment context dependent attributes - Is it necessary to have system environment information to analyze this quality attribute?

Table 1: The different quality attributes and their classification

		Classification				
Higher level attribute	**Quality attribute**	**Directly composable**	**Architecture-related**	**Derived**	**Usage-dependent**	**System environment context**
Usability	Accessibility		xx			
	Administrability				xx	
	Understandability			x		
	Generality			x		
	Operability				xx	
	Simplicity				xx	
Portability	Mobility			xx		
	Nomadicity			xx		
	Hardware independence	xx				
	Software independence	xx				
Performance	Accuracy	xx				
	Footprint	xx				
	Responsiveness				xx	
	Scalability	x	xx			
	Schedulability		x	x		
	Timeliness		x	x		
	CPU utilization			x	x	
	Latency		x	x		
	Transaction Throughput		xx		x	
	Concurrency			xx		
	Efficiency	xx	x			
Maintainability	Flexibility	x	xx			
	Evolvability	x	xx			
	Extensibility	x	x			
	Modifiability	x	xx			
	Upgradeability	x	x			
	Expandability	x	x			
	data consistency	xx				
	Version consistency	xx				
Integrability	Adaptability	xx	x			
	Composability	xx				
	Interoperability	xx				

		Classification				
Higher level attribute	Quality attribute	Directly composable	Architecture-related	Derived	Usage-dependent	System environment context
	Openness	xx				
	Heterogenity		x	x		
	Integrability	xx				
	Audibility	x	xx			
	Completeness	x	x		x	
Design	Conciseness	xx				
	Correctness		x		x	
	Testability	x	x			
	Traceability	x				
	Coherence	xx	x			
	Analyzability	x	xx			
	Modularity		xx			
	Reusability	xx				
	Configurability	x	xx			
	Distributeability		xx			
Deployability	Ease of creation		xx			
	Availability		x	xx		
	Confidentiality			xx		
Dependability	Integrity			xx		
	Maintainability	x	x			
	Reliability		x		xx	
	Safety				x	xx
	Security				xx	x
	Cost			xx		x
	Projected Lifetime	x	x	xx		x
Business	Targeted Market			xx		x
	Time to market	x	x	xx		x
	Affordability			xx		x
	Development time	x	x	xx		

Table 1 and Figure 10 show clearly that the greater proportion of the attributes are dominated by or related to the architecture, which is not surprising. As concluded in the book by Bass et al, the design of the software architecture is the primary instrument in achieving certain qualities [7]. The components are assembled together according to certain architectural rules. The importance of software architecture is obvious, which implies that the choice of component model and technology is of equal importance as a model and technology is partly defined by the architectural rules.

The second result, which shows that a large proportion of attributes are directly composable is indicative; it shows that many attributes of a system can be determined by reasoning about the composition of these attributes on the component level. The large number of such attributes can be explained by the fact that most of the attributes in the table are concrete and related to technologies, while fewer are composed. Further, knowing that CBD and software architecture are much related, it is not surprising that many component models have impact on the number of quality attributes. Indeed, from the table we can see that more than 50% of attributes belong to the category of directly composable and architecture-related attributes.

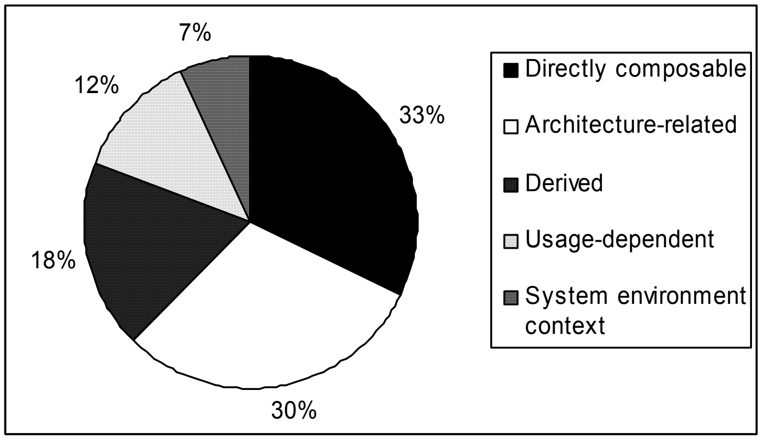


Figure 10: Distribution of the various attributes types from Table 1

It can also be noticed that all attributes related to direct composition are strongly influenced by the component model. A desired directly compositional attribute can be obtained or at least supported by a wise choice of component model. On the other hand, if there is an existing component model, which does not support the particular desired attribute, there might be a problem. The component model might actually oppose the compositional aspects of the attribute.

Directly composable properties and those related to architecture are dependent on the choice of component model. The other types, i.e. derived, usage and system environment dependent properties are dependent on an underlying component model or technology to a lesser degree.

From the distribution graph we can conclude that many properties, first of all those tangible and related to technologies are influenced by a component-based approach. This implies that a deployment of a component-based approach in a development process may affect many quality attributes.

3.6 Validation of the classification

We have evaluated the classification by performing a survey on twelve researchers. Our intention was to get input on the difference interpretation of the classification and to see if the classification is intuitive. A possible additional objective is to see if there are many different perceptions about the meaning of the quality attributes. Each researcher answers the questions for the quality attributes in Table 1. Answers for each attribute is marked with the confidence of the researcher. The confidence is marked from 0 to 5. The survey is performed without verbal discussion with the researchers and it is remarked that the definitions and descriptions of the various quality attributes are not sufficient to provide a clear classification.

As each researcher can indicate the confidence in the answers, we decided to remove all answers that indicated low or no confidence indicated with 0 or 1. To see if an attribute belongs to a category we take the result if 60% of the researchers are in agreement. For disagreement the result could be either part or not part of the classification. Figure 11 illustrates the distribution of the various attributes in the classification according to the result of the survey.

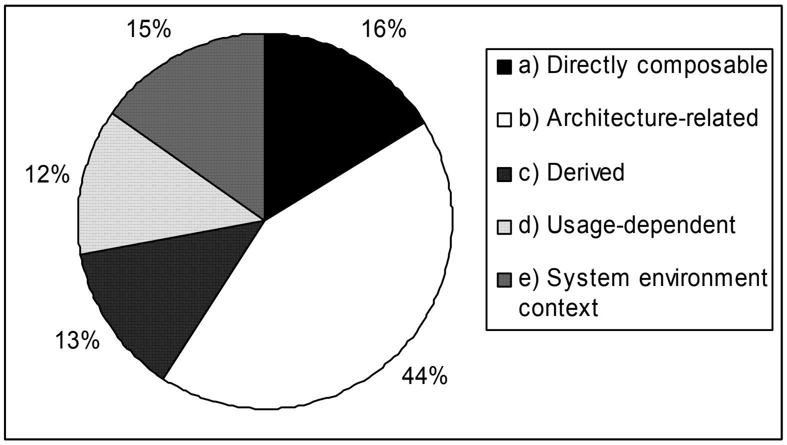


Figure 11: Distribution of the various attributes according to the survey (Table 22)

The level of agreement, for each category, between researchers is shown in Figure 12. A high level indicates that the researchers are of the same opinion when it comes to classification of the attributes. For the architecture classification it is 95% agreement and that indicates that the researchers are consistent in classifying an attribute in the architecture depending class.

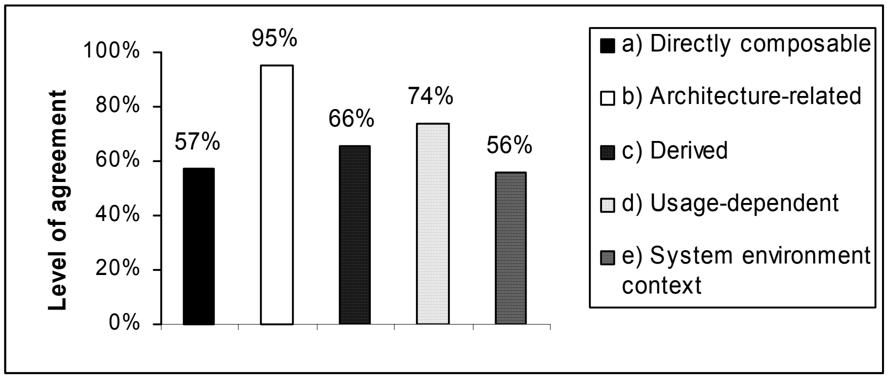


Figure 12: Level of agreement for the classification

We also compared our classification in Table 1 with the research answers. In 67% of the answers we have the same classification as the researchers. Our classification is performed under careful consideration with all the definitions in mind. The researchers did not have the same background information and were performing the survey more out of the context. The researchers results are presented in Table 22. It can be concluded that the researchers are not overly confident in their answers as around 60% answers that they have confidence on the low side, 2, 1 and 0. Figure 13 illustrate the distribution of the confidence of the answers. The descriptions of the quality attributes might have been insufficient and led to different opinions about how to classify the attributes.

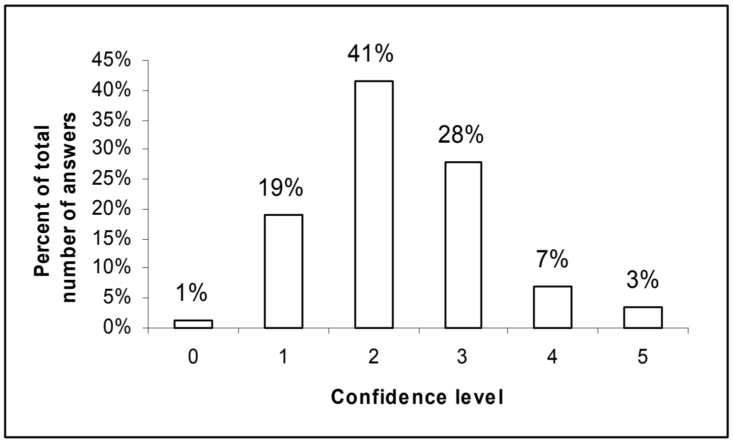


Figure 13: The confidence distribution of the answers

To see if the confidence level for each attribute is related to the level of agreement a regression analysis is performed. The result is that there is no such correlation. There are many attributes with high agreement with low confidence in the answers and vice versa.

3.7 Composition of Dependability Attributes

To illustrate the attribute classification, we take dependability as an example. *Dependability* is defined as the ability of a system to deliver service that can be trusted and the ability of a system to avoid failures that are more severe and frequent than are acceptable to the users [4]. Dependability is the main quality attribute when building safety-critical systems, i.e. in systems where failure can cause human casualties or great economical loss. In most cases, safety-critical systems are hard real-time systems (i.e. systems in which the timing requirements must be met), and embedded systems (i.e. a combination of software and hardware), which implies a number of timing and resource constraints.

Such systems require rigorous development procedures and software architecture which can make them as predictable as possible. Dependability is a quality attribute consisting of six main quality attributes, namely, availability, reliability, safety, confidentiality, integrity and maintainability.

The questions of interest to component-based software engineering or development are:

- Which of the dependability attributes are emerging or derived system attributes, which are both system and component attributes?
- In a component-based system how are these attributes related to component attributes?
- To which extent (and how) can these attributes be determined from component attributes?
- To which extent can the uncertainty of the predictability of these attributes be minimized and how much is it related to the uncertainty of the component attributes?

The following is a short analysis and classification of the dependability attributes.

3.7.1 Reliability

The basis for the definition of reliability is the probability that a system will fail during a given period. One definition is that reliability is inversely proportional to this probability as mean time to failure (MTTF) and it is defined for a specified period under stated conditions [4]. The probability of failure is directly dependent on the usage profile and context of the module under consideration.

$$MTTF(A) = \frac{1}{P_f(A)}$$
$$A = \text{module (component, assembly, module)} \qquad \textbf{(18)}$$
$$P_f = \text{probability that module } A \text{ fails per time unit}$$

The module can be a system, or an assembly or a component, so the relation between the reliability of an assembly and a component can be expressed by (10) and, because of the fact that the reliability is measured under stated conditions, a usage profile must be specified, which is expressed by (12). The dominant type of impact on reliability is the usage profile but reliability is also depending on software architecture. A fault-tolerant redundant architecture improves the reliability of the assembly of components. One possible approach to the calculation of the reliability of an assembly is to use the following elements [91,92]:

- Reliability of the components – Information that has been obtained by testing and analysis of the component given a context and usage profile.
- Path information or usage paths – Information that includes usage profile and the assembly structure. Combined, it can give a probability of execution of each component, for example by using Markov chains.

It is noteworthy that even if the reliability of the components are known it is very hard to know if there are side effects that will affect an assembly of the components, e.g. if the components write in a memory space used by another component thereby causing a failure. A model based on these assumptions needs the means for calculating or measuring component reliability and an architecture which permits analysis of the execution path. Component models that specify provided and required interface, or implement a port-based interface make it possible to develop a model for specifying the usage paths. This is an example in which the definition of the component model facilitates the procedure of dealing with the quality attribute. One known problem in the use of Markov chains in modeling usage is the rapid growth of the chain [92,124]. The problem can be solved because the reliability permits a hierarchical approach. The system reliability can be analyzed by (re)using the reliability information of the assemblies and components (which can be derived or measured).

3.7.2 Availability

Availability is defined as the probability of a module being available when needed. Formally, it is defined as the mean time to failure divided by the mean time between failures (MTBF), which in turn is the sum of the MTTF and the mean time to repair (MTTR):

$$Availability(A) = \frac{MTTF(A)}{MTTF(A)+MTTR(A)} \qquad \textbf{(19)}$$
$$A = \text{module (component, assembly, module)}$$

From (19) we can see that availability is related to reliability. In the same way as reliability, availability can be obtained by measurements through the usage profiles. It is however not the same property; For example a module can be considered to have good availability if it has relatively poor reliability but requires little time to repair. Also, an assembly, or a component can be reliable but not available, for example due to lack of resources (the component may need to wait for the resource to start). MTTR is related to a means to attain the dependability, i.e., it is related to fault tolerance, fault forecasting and fault removal. All these terms are not formally specified and cannot be directly measured. The difference between reliability and availability is that availability is dependent on the dynamic state of the system – the availability of an assembly cannot be derived from the availability of the components in the way that its reliability can be derived from the reliability of its components.

Further, if availability is treated in a larger context, non runtime properties or quality attributes must be taken into a consideration. Availability is related to the maintenance and support of the components constituting the assembly. Factors such as these are very important in determining if or not a component is to be selected for inclusion in a system. It is important to reduce the mean time to repair.

3.7.3 Safety

Safety is a property involving the interaction of a system with the environment and the possible consequences of the system failure. It is a system property, neither a component nor an assembly property. Safety depends on where and how the system is deployed. For example, a system controlling a robot could cause harm to a human being if a fault leads to an uncontrolled robot movement. Safety is increased by installing the robot in a cell which cannot be entered during operations.

Since safety is a system property that is dependent on the system's environment, a means for analyzing safety is a top-down approach, a decomposition rather than composition. In the analysis process, the components' attributes are used as selection criteria or are identified as demands that should be met. For this reason, a component-based approach might not have the apparent advantage – on the contrary, if the starting idea is a reuse of existing components, the components' attributes cause new constraints and in this way might decrease the system safety. However, when the constraints are identified and unambiguously related to the constraints on the system level, the system safety can increase. In addition, some properties, such as reliability, might improve the accuracy of the system safety prediction, especially if known or measured when used in other applications.

3.7.4 Confidentiality and Integrity

A system would not be dependable if unauthorized access or, even worse, unauthorized alterations of the system data, were easy or even permitted. Hence, security aspects, confidentiality and integrity, defined as follows [4] apply to dependable systems.

Confidentiality is defined as a measure of the absence of unauthorized disclosure of information.

Integrity is defined as the absence of improper system state alterations.

From the definitions, it is apparent that these properties are not directly measurable, and this is the main obstacle to the development of a theory for their prediction.

Integrity is a prerequisite for availability, reliability and safety, but need not be so for confidentiality or maintainability. Confidentiality and integrity are emerging system properties that can be tested and analyzed on the system and architectural level but not on the component level. Usage profiles can be used for testing and analysis, but it is impossible to derive these properties from the component properties in an automatic manner.

3.7.5 Maintainability

Maintainability is defined as the capacity to undergo repairs and modifications. Maintenance is of three different types, corrective, adaptive and perfective maintenance [4]. Corrective maintenance is the removal of faults, thereby improving the reliability of components under the condition that no new faults are introduced. Adaptive maintenance is performed when a component is modified to meet modifications in the target environment. Perfective or preventive maintenance addresses improvements of the components in response to users or designers' input.

Maintainability is related to the activities of people and not of the system itself. Component technologies might provide support for dynamic upgrading/deployment of components which can improve the maintainability of a system. In this case, the maintainability is much a matter of component technology, and not of the component itself. The system architecture thus has an impact on maintenance.

Component technologies certainly affect corrective and adaptive maintenance but not perfective maintenance to the same degree. A component-based architecture might be easier to adapt to environmental changes because of its natural modularity.

A modular architecture might facilitate, depending on the relationships between the components, the removal of faults in corrective maintenance but for efforts to perfect the implementation, a component model is of lesser importance. For instance, a need to replace all macros in the code with templates does not depend on the component model.

As the classification in Table 1 indicates, most maintainability attributes are probably directly composable or architecture-related.

Many parameters can be measured and used to estimate the maintainability of a code (for example McCabe's metrics for complexity [73]). These parameters can be identified for each component. It is however not clear how these parameters can be defined on the assembly level. One possibility is to define a mean value of all components normalized per lines of code.

3.8 Summary and Conclusion

A full advantage of the component-based approach will be achieved when not only the functional parts are reused, but also when this approach leads to easier and more accurate predictability of the system quality. Systems designed and built from components have many system attributes that can be derived from the component properties, this being more accurate if a support for defining and measurements of the attributes are built in the component technologies. However, a predictability of attributes does not depend only on component models but also on the attributes themselves. For each property, a theory of the property, its relation to the component model, composition rules and an overall context related to the requirements must be known.

The quality attributes can be classified with respect to types of composition, in which each type is characterized by the required input for obtaining predictability on the system level. Some types show clear composable characteristics, while others are not directly related to compositions.

The existing component models differ considerably and how the assemblies and components' attributes are treated will be highly dependent on these models, especially for those properties that are directly composable or are related to the architecture. For example, if the component model has independently deployable components with a 1st order assembly model, it is likely that the properties of the components cannot be propagated further than the assembly level without considering the environment.

In spite of diversity of properties, technologies, and theories, it should be possible to create reference frameworks that by identifying the type of composability of attributes can help in estimation of accuracy and efforts required for building component-based systems in a predictable way.

4 Predictable Assembly of Components

In the previous chapter, we have looked at certain component models from the perspective of prediction. This chapter presents methods how to define or extend a component model to support predictable assembly of components.

Predictable assembly is to take components and their properties and reason about resulting assembly quality attributes. One of the main challenges in constructing and maintaining software is to express and verify quality attributes on applications. By introducing predictable assembly of components, we provide means for expressing and predicting quality attributes derived from the properties of the individual assembled components.

Application properties specify the quality attributes and in most of the cases they cannot be expressed as part of the functional interface. The same is valid for components and their properties, which cannot be expressed through the functional interfaces. For this reason there are many research attempts, such as [84], to extend the component specification.

If components can specify their quality attributes as properties, this information can be used to reason about assemblies of components. This chapter presents a method how to build a component technology that has predictability features built in. The chapter also presents a method how to empirically validate such an implementation.

Section 4.1 sets the scope and goal of this chapter. Predictable assembly is described in Section 4.2 and the more concrete implementation a prediction enabled component technology is presented in Section 4.3.The method how to build a component technology supporting prediction is given in Section 4.4 and the validation procedure of such a technology in Section 4.5. The chapter is concluded in Section 4.6.

4.1 Scope and Goal of Chapter

The scope of this chapter is to outline what predictable assembly and component technologies supporting prediction are. The chapter is based on the work at the software engineering institute (SEI) in which we participated [44,47]. The SEI has developed the concepts of predictable assembly from certifiable components and prediction enabled component technologies. We have extended these concepts by developing a method that builds and validates prediction enabled component technologies. Our contribution with the methods is found in Sections 4.4 and 4.5.

These methods are the main goal of this chapter. The second research question, how to verify predictions to gain objective trust is answered by providing the validation method. An additional goal is to provide a base for answering the first research question if it is possible to develop a component technology that will support reasoning about system properties from component properties.

4.2 Predictable Assembly

From the predictability point of view, obtaining new functional features of the application is relatively simple as they come directly from the functional properties of components; on the opposite, the quality attributes of applications are hard to predict. For example, adding components with new functional features may degrade the quality of services of a product or affect the temporal correctness.

There are various approaches that can be taken to introduce predictability in a software application and below we list three of them.

1. The first approach is to build a software application for one purpose only and with very much "hard coded" logic to calculate, monitor and assume the quality attributes. Since the application is designed, from the beginning, with the goal to be able to show certain quality attributes it gives also the most possible freedom to the developers. However, this one-time development effort resulting in a component technology will be very hard to reuse and modify to incorporate new quality attributes if wanted. In addition, it is possible that the various implementation components are hard to reuse because of the cross cutting quality aspects. With this approach, the application will probably work and the desired quality attributes will be achievable. However, when later change occurs, it will be a tiresome exercise to adapt, because of probably poor maintainability and modifiability.
2. A second approach is to improve modifiability by having component-based software architecture. If the first approach gives the most freedom to the developer but least modifiability then the second improves the modifiability. In this approach, there is only one target application as in the first but this time we implement it by having a component-based architecture. By designing the application and the components so that the components can provide what is needed to get the quality attribute of the application we can possibly hide some of the complexity inside the components. The components are later easier to exchange and modify without touching the whole application. Usually also a software architecture with defined functionality in modules is easier to maintain and modify.

   The cross cutting aspects of the application is possible to achieve with aspect oriented programming. As an example, we consider the work done by Schultz et al that introduces fault tolerance quality in a .NET component based environment. They use the attribute mechanisms in .NET to express the quality attributes for fault tolerance [93]. This second approach is probably most used in practice today, but there is still little capability for achieving reuse of components between applications. In addition, there are no strict rules forcing the use and implementation of components to achieve predictability, the approach is more ad hoc and an extension of new quality attributes is hard.
3. The third approach is to have a software architecture that can be used for many applications and products. We would like to have a technology that supports the underlying mechanisms needed to deal with quality attributes. This third approach designs a technology that serves as a framework for predictable assembly of components. The technology also defines architectural rules. To increase flexibility we put much effort into producing the component technology infrastructure. The components themselves can carry the information needed about the quality attributes or they can rely on an external specification. The clear benefit with this approach is that the infrastructure provides means to achieve predictability of the quality attributes. This approach is taken by the predictable assembly from certifiable components (PACC) group at the software engineering institute (SEI) and related work is found in references [43,45,47,119].

To be able to predict the application properties from the component properties, SEI defines both a *component model* and an *analytical model*. An example of such analytical model used on a component level is a model used to calculate the version dependency among components. While a component model deals with operational (functional) properties, the analytical model is used to define and reason about quality attributes.

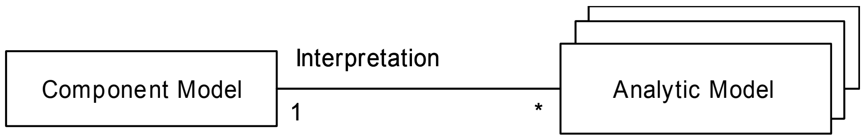


Figure 14: A component model can have many interpretations as analytic models

Each component model can be interpreted into several analytic models, where each analytic model is used for particular quality attributes. An instance in the component model can thus be represented in many analytic models (see Figure 14). The component model does not have to be consistent with the analytical model. It is sufficient to have the possibility transform a concrete assembly into an analytical assembly, there is no need to do vice versa.

As information might be abstracted away from the component model when a translation into the analytic model is done, we do not have a reflexive relation between the models. It is not possible to take a representation from the analytic model and create an instance of it in the component model if vital information is lacking on the component model side.

If the component model is implemented in a component technology to produce software applications, the analytic models might be in a similar manner implemented in various analytic technologies. Although it is not necessary to do this implementation if manual work is acceptable to calculate the predictions. In many cases, integration of the analytic technology with the component technology is desired, to achieve maximum degree of automation. A prediction enabled component technology (PECT) is an example of how such an integration of analytic and component technologies can be done.

The fundamental principle of predictability is the ability to predict properties of assemblies from properties specified for components. The premise to achieve predictability is to have the possibility to restrict and limit both what is possible to predict and the information needed from the components.

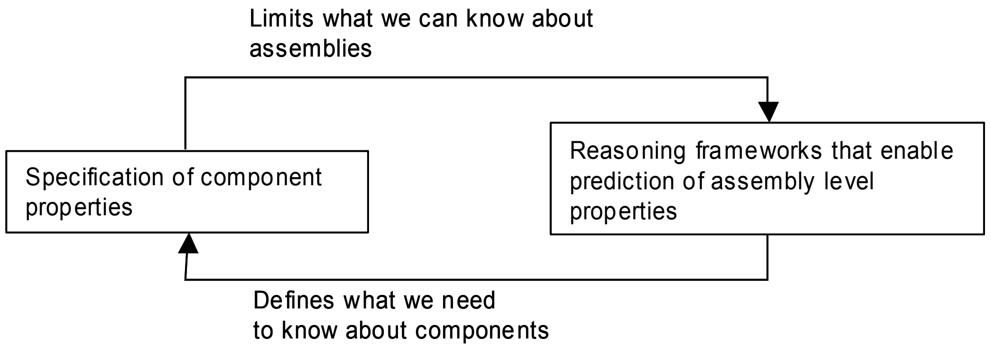

Limits what we can know about assemblies
Specification of component properties
Reasoning frameworks that enable prediction of assembly level properties
Defines what we need to know about components

Figure 15: What we want to know about assemblies affect what information is needed from the components and vice versa [119]

To achieve predictable assembly of components, a framework for reasoning about quality attributes is needed. Figure 15 displays that the information needed from the components define and limits what we can know about the assemblies. In addition, the reasoning framework used to calculate the predictions of the quality attributes at the assembly level defines what we need to know from the components. There is a mutual dependency between the components and the assembly of the components where restrictions and limitations take place. It is not possible to predict whatever assembly property without setting certain requirements on the components and their information.

Predictable assembly deals with the method on how to gain equilibrium of the limits/restrictions and needed information. When using a commercial component model it is clear that many of the reasoning limits on the assembly level is already set. A commercial component technology, non open-source, has less modifiability and more constraints than a proprietary component technology, from a user perspective. Changes to the commercial technology cannot be carried out by the user of the technology; instead, those changes are imposed from the technology vendor.

In a similar manner, the information that commercial components can provide is limited and hence the level of flexibility is low. For example, if reasoning about the memory consumption is desired and there is no way to express or obtain the memory information from the components then we are bound to fail. On the other hand, if the component model is very flexible and extensible to provide lot of information it will be easier to build a reasoning framework for the desired attribute.

Components that express quality attributes in form of properties fulfill the necessary conditions for certification. Such components are certifiable. That means that somehow it is possible to gain objective trust in what the property says is true or at least to what extent and under which conditions it is true. If the property of a component states that memory consumption is maximum, let us say, 30 Kbytes, it is of value to the user to know how the data was provided and if there are confidence levels applied to it. The value of a property might be estimated, calculated or measured.

The measurement approach often results in probabilistic values based on statistical means with uncertainties in the measurements. As a result, assemblies are analyzed based on present or measured properties of the components. The theory used to analyze a property of an assembly is denoted a *property theory*. A property theory is the analytical theory used for a specific quality attribute and defined in an analytical model. If there are different attributes to be analyzed then theories and analytical models have to be developed respectively.

If the value of the property is measured, the statistical distribution and confidence level of the measured value is of interest for the user. There are also properties that can be measured exactly, like component dependencies, where a statistical distribution is not needed.

A problem is to gain trust in the infrastructure, underlying technology and certification of the components. This is addressed more in the Section 4.5 and in the work from Moreno et al [77].

Inspection of components and other software entities is a form of measurement, and certain error is always present depending on the granularity of the measurement instrument. This can cause a systematic error. Random errors that occur in the measurements are not taken into account in the systematic error. I.e. the systematic error is the measurement error minus the random error. Objective measurement requires a measurement object, in this case the components and assemblies, a measurement scale for the property of interest, and a measurement apparatus. The measurement object is called the *measurand*. An infrastructure and tools that help with analyzing the measurements are helpful but not required. Measurements are taken by the defined apparatus in a controlled test environment with control of depending variables of the measured quality attribute. When measuring the properties it is good to be aware of the probe effect. I.e. the measurement apparatus might introduce and affect the resulting value of the properties. It is of value to quantify the probe effect and take it into consideration in the property theory to get more accurate results.

Component properties can be calculated or measured. Measured properties are only accurate for one specific environment if not somehow parameterized. Component properties provided by measurements should be described with statistical description of the measurements taken, to know not only the value or the range of value a component property, but also the accuracy of the measurement.

If measured properties are used of the components then the prediction theory ought to consider the uncertainties in the measurements when making the property prediction of the assembly property. The need for calculating error propagation is depending from case to case and on how big deviations there is in the component property measurements.

4.2.1 Description of Accuracy of Property Prediction Theories

A property theory can be validated using empirical studies, and can have various descriptive statistics attached. The theory is a hypothesis of what the assembly properties are, given the component properties and the component interrelation. This hypothesis is tested with sample assemblies in the validation phase. The descriptive statistics can be seen as an attached label that provides information about what capabilities the theory has.

A *statistical label* is a statistical descriptor of the precision of the component properties and assembly properties and property theories. One such statistical label is the proportion. This term express how much of a certain proportion of all executions falls within a specified interval that meets a certain criteria. The proportion is important since a couple of the assemblies measured in the validation can behave unexpectedly for a random reason.

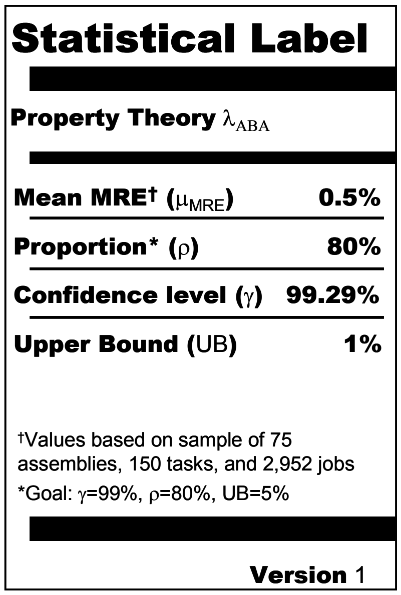
The statistical label is used to describe the accuracy of the property theory. Since the theory should hold for many assemblies, it is more difficult to describe and obtain these labels than for the component. Many different assemblies of components have to be analyzed in this process. The goal is to determine the accuracy of the predictions made with the theory of a certain quality attribute.

For measured assembly properties we want to know the relative error between the predicted value and the actual measured one. The relative error is expressed as the magnitude of relative error and is calculated as the difference between the measured property value *P* and the predicted value *P′* divided by the measured value *P*. The magnitude of relative error (MRE) is not applicable for quality attributes that cannot be representative with measurable properties.

$$MRE = \frac{P' - P}{P} \qquad \textbf{(20)}$$

For the cases where the values of the component properties are not expressed with a fixed value but rather a range, it is appropriate to take the range into consideration in the prediction theory. If the prediction theory uses ranges of component properties it is likely that the predicted value produced is a range and then we have to calculate maximum and minimum relative error. A good prediction theory shall include means to handle error propagation from the component property value to the assembly property.

For the property theory, average MRE can be expressed over all statistical samples taken during the empirical validation. The proportion and confidence level conveys the probability that the MRE for a particular prediction falls within a specified MRE interval. If a prediction falls within the lower bound of an MRE interval then the prediction of the property is better than the mean MRE of the statistical sample. The mean MRE corresponds to the statistical sample and we want to know the frequency of predictions that are better than the mean. We illustrate a statistical label with an example in Figure 16.



Statistical Label

Property Theory λ_{ABA}

Mean MRE† (μ_{MRE})	**0.5%**
Proportion* (ρ)	**80%**
Confidence level (γ)	**99.29%**
Upper Bound (UB)	**1%**

†Values based on sample of 75 assemblies, 150 tasks, and 2,952 jobs

*Goal: γ=99%, ρ=80%, UB=5%

Version 1

Figure 16: Example of a Statistical label for a component technology (With courtesy of Scott Hissam)

Figure 16 shows how a label can look like for an example property theory and the resemblance with a nutrition label on a pop can is noteworthy. Theories andn their accuracy can be expressed with statistical information to provide better understanding of the exactness provided. Figure 16 demonstrates the statistical label for a property theory used to predict latency properties at the assembly level given the components their connections and their latency information. For this particular theory, the validation was based on a sample of 75 assemblies, 150 tasks and 2952 jobs[1]. The tasks and jobs are specific terms for this example component technology and are explained more in a Chapter 5 for a concrete example.

The different variables in the statistical label are first the magnitude of relative error (MRE) that gives the error of the predicted value compared to the actual measured. The mean MRE is for all predictions and measured value in the given sample. That is, the validation of this component technology carries out measurements for the whole sample and then the mean is taken.

The shown example shows also a label version number. It is important to keep track of the versions used of the component technology and the property theory when generating the predictions. There might be many theories used for the same implementation of the component technology.

Another reason for keeping track on the version is that the theory and implementation of the component technology is done in an iterative manner where both technology and theory is refined depending on each other. The method for how to create this underlying component technology, also from now on called prediction enabled component technology (PECT) is later described.

The upper bound magnitude of relative error (UB) is where we find assemblies relative to the proportion. Having a proportion of 80%, UB of 1% and confidence level of 99.25%, means that 8 out of 10 predictions fall within a relative error of 1% with a confidence level slightly more than 99%,

[1] Chapter 5 presents how the assemblies are generated and what architecture is used.

under the condition that this theory and an assembly produced in this component technology is used. More information on this particular example can be found in [43].

By having labels on our components in a similar manner and for the component technologies and theories we can have objective trust. If we state a prediction, it is possible to know what confidence we have in the prediction and how this confidence was achieved. Later this chapter takes a deeper look into how validation of property theories and how statistical goals can be achieved. However, first we look into the concepts of developing a prediction enabled component technology.

4.3 Prediction Enabled Component Technology

Prediction Enabled Component Technology (PECT) is one approach to achieve predictable assembly from certifiable components. It is the approach taken by SEI and it is executed in the PACC research initiative and in collaboration with ABB and Mälardalen University [42,43,45-47,119]. In this approach the prediction-enabled capabilities of the analytic model is packaged together with the constructive mechanisms of the component technology as shown in Figure 17.

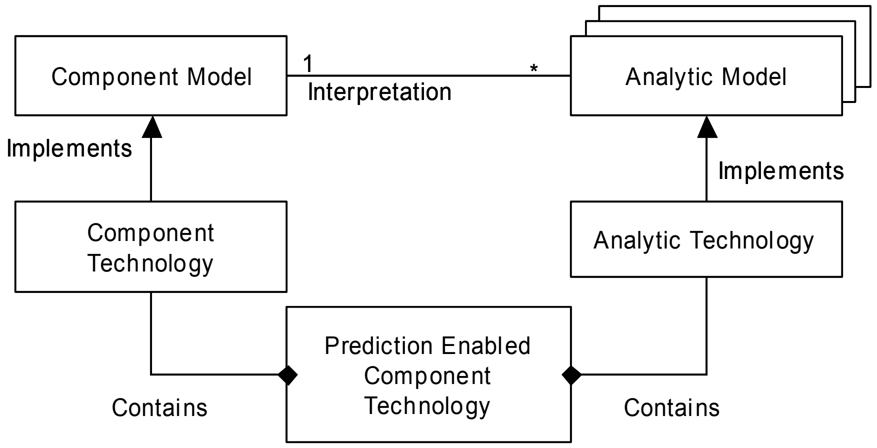


Figure 17: A PECT package component and analytic technologies together

The constructive parts of a component technology are used to build and satisfy both functional and non-functional requirements. Components have to comply with the restrictions and rules set by the component technology. As the analytic part also is included in a PECT, the rules and obligations on the components from this perspective are also enforced. Without sufficient component information, it is not possible to do any analytic reasoning and one way to make sure the components provide enough information is to enforce it in the component technology. However, there is certain information that can be added after the components are implemented, example of such data is measured and estimated attributes.

The theory and algorithm to calculate predictions are part of the PECT and later validated. To achieve confidence in predictions made with the PECT, it is necessary that the PECT is validated There are many if not indefinite number of assemblies of components that can be created using a PECT so it is difficult to validate that the theory holds for the whole space of possible assemblies. By selecting certain sample assemblies representing the problem domain, we can achieve certain trust in the predictions of the PECT.

The PECT is a framework where components are instantiated and run, i.e. it has a runtime environment, it also has one or several embedded reasoning frameworks. A construction environment can be used to describe how the components are instantiated and their interrelations via connectors. A simple example of how an assembly of component could look like is given in Figure 22. Another example on how to express and describe assemblies is to use a composition language [57].

Each component must comply with the rules that are set up by the PECT runtime and the analytic framework. For instance, there might be specific interfaces that have to be provided, and each component must be packaged in a deployable format. The deployment format is normally a compiled binary of the component code. When the components are deployed, i.e. installed, in the target environment they are ready to be started

The assembly describes what components are to be instantiated and how they should be connected. What is important to clarify again is that when a component is executed it is actually one instance of the deployed component. It is possible to have many instances of the same deployed component.

Consider a graphical component displaying some measurements; such a component can be used several times in one application but it is only deployed once. Each instance is unique and has its own state and representation in the target environment. Even as there are two kinds of components as shown above, the term component is used both to denote the deployable component and the component instance, which often leads to misunderstanding. From the context, it is often possible to determine the kind of component, and even in this thesis the term component is used for both instantiated and deployable components.

After the components have been deployed in the target environment it is important to measure and gather the quality attributes that depend on the surrounding environment. Performance and latency are examples of attributes that heavily depend on the execution environment. A PECT might include a framework to do these measurements in an automatic fashion, but this is not required although desired. Automatic measurement would mean to instantiate one component from each deployed component and measure with some pre set values. The measurements must be stored for later retrieval as a statistical label for the components properties.

When a specific assembly is defined in the constructive component model and transformed into the analytical model, we can ask for predictions of certain properties. Information about the component properties and connections is used as input to the prediction algorithm. The assembly is the actual application or part of the application that the user will run to perform the wanted functionality. In addition, depending on the component technology different approaches is taken how assemblies are managed.

Our approach is that when the application is executed, no measurements will be taken or any other predictive actions, the user should have confidence that the predicted attributes fulfill the set of quality requirements. At application startup, the assembly information is passed to the PECT runtime that instantiate all components and configures the connectors between the components. If the PECT is built using existing component technologies with different architectural rules, other approaches can be taken.

Ordinary component technologies like COM do not have a description of the assembly to be run. In COM it is the application, and indirect the application engineer, itself that decides what component to use and instantiate. In addition, the interactions between the components are set by the application and are not explicitly described. With such a component model without the notion of assembly, it is hard to create any a priori statement about quality attributes of the application. With the PECT concept, it is required that the components and their connections are explicitly described.

4.4 Building a PECT

A prediction enabled component technology has to be carefully designed, built and validated to provide confidence in proof of correctness. This section describes our method how to perform this task of designing and constructing a PECT. A PECT should be seen as a component technology with extensions and restrictions that make it prediction enabled.

We define the design of a PECT in four fundamental phases, they are definition, co-refinement, validation and package, as illustrated in Figure 18.

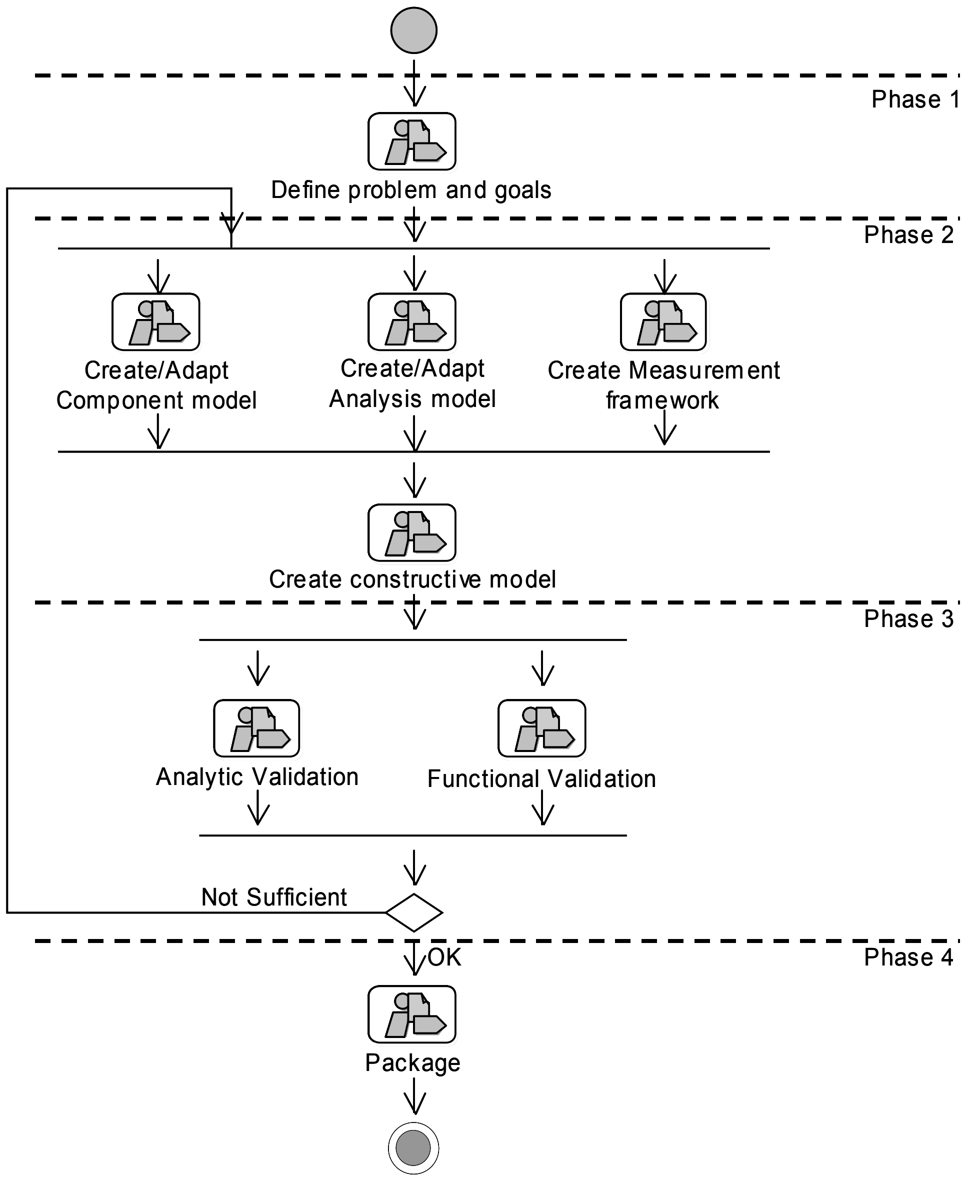


Figure 18: An overview of the PECT method using RUP notation [64]

During the definition phase, different requirements and goals are set up for the PECT. The goals define what the predictions will do and how to represent a prediction. Also in this phase, the quality attributes to be analyzed are identified. Important to remember when defining the goals and objectives of a PECT is that it should be used to build future functional applications for the user. Both functional and quality aspects have to be considered when building a PECT, since assemblies created in the PECT will be part of functional applications. When the goals are set, a component model, analysis model, and measurement framework is created.

In the co-refinement phase, the analytic model is designed together with the constructive component model. There might be several analytical models that have to be integrated depending on the number of different types of properties that we want to reason about in the PECT. The designed models might be based on existing models and then these are adapted to fulfill the prediction requirement. Both the component model and analytic models must consider each other's requirements, i.e. they are interdependent. The models are implemented in a technology, which produces one instance of a PECT, but not necessarily the final version. The technology can be based on existing component technologies to reuse as much of the infrastructure as possible. This phase also includes the creation of tools and methods to do measurement for validation support.

The produced PECT instance is then put into the validation phase where the accuracy of the predictions is validated. Functional validation may also take place during this phase. Validation is to prove that if the PECT fulfills the stated statistical goals for the predictions. If the goals are fulfilled it is time to package the PECT, but every so often, a couple of iterations are needed before the PECT satisfy each goal.

To package the PECT is an important part of the development. This phase includes how to deploy the PECT in the target environment. There might be part of the PECT that needs to be measured and configured in the target environment. One goal is to make the use of the PECT as easy as possible, and hence a lot of automation support should be packaged into the final deployable PECT.

The whole development process produces several deliverables of which some are optional. However, the overall goal is to produce a PECT that has the following capabilities:

- Zero programming assembly

  Components must be assembled without additional programming since the predictions are made on the base of having a homogenous component environment. If additional logic has to be provided then the logic has to be encapsulated in PECT compliant components.

- Automatic interpretation

  To perform predictions in a target environment where the target assemblies are unknown, the PECT must be able to automatically interpret a constructive assembly into an analytic assembly. The analytic assembly is then used to perform the desired prediction.

- Objective trust

The objective trust is achieved by validating the prediction theory using a wide variety of assemblies. There are statistical means to calculate how many assemblies have to be executed to meet the statistical goals set for the prediction theory.

Certain additional capabilities usually provided from a PECT developed following this method are:

- A set of components

  A starting set of components has to be provided for doing empirical validation and testing of the PECT. These components could be reused in assembling new applications using the functionality of these components.

- A framework for taking measurements

  Measurement of the components' properties in the target environment is crucial to perform predictions about attributes of an assembly, hence a framework for measuring each individual component is necessary. Components are measured whenever deployed in a target environment. Predictions will be based on those measurements and will be valid only for that particular environment.

- Set of packaged solutions

  The PECT is packaged in a way that when applications and components are deployed into the technology, there should be as little manual work as possible. Automated measurement of components and configuration in the target environment is included in a packaged solution.

To deliver the desired capabilities we need to perform several activities. In the Rational Unified Process (RUP) [64], an activity is generally assigned to a specific worker and has duration of a few hours to a few days; we relax this condition somewhat and allow multiple workers to contribute to an activity, where that activity has an indeterminate duration. As with the RUP, each activity has a clear purpose, usually centered on creating or modifying artifacts, these are the tangible deliverables of the PECT development method.

The activities that will be carried out are depending on each phase and the generated artifacts. When the activities for each phase are assigned to just one worker, it should be interpreted not as an exclusive assignment, but rather as an assignment of primary responsibility. In general, multiple workers are required for most artifacts. For example, defining the requirements for a PECT is primarily the responsibility of the customer, but setting the appropriate requirements requires the contributions of the PECT designer, attribute specialist, and component model specialist. The artifacts are the results of workers performing activities.

A worker identifies the behavior and responsibilities of an individual or group of individuals working together. One example of a worker is the designer who designs the system. The individual that works as the designer can participate in other defined activities, i.e. an individual can perform several different activities through acting in different roles.

We have identified several different workers participating in this process. As they work in many of the given phases, we portray them in this section rather than in the phase's sections.

The *measurement specialist* knows how to measure and empirically validate the different theories in the PECT. Typically, this worker has to have sufficient knowledge of statistical methods and measurement of quality attributes to perform the task.

A c*omponent developer* designs and develops components for the PECT that reside and execute within the PECT runtime. The components developed might be intended for pure test, empirical validation, general functionality, or for a specific application. Usually there is more than one component developer active in the process at the same time especially when it comes to application specific development.

The *attribute specialist* knows the theories behind the chosen attribute for which to perform predictions. This worker might also support the measurement specialist with measurement techniques depending on the attribute. For example: if the attribute is latency then the attribute specialist would have to know real-time systems and how to analyze such systems.

The *PECT designer* holds the important holistic view of the PECT and nurtures the other workers with necessary information to keep the different parts of the PECT development synchronized. Important for this worker is to lead the co-refinement phase and negotiate the constraints between the component model and the analytic model.

The *component model specialist* knows the component model of the PECT and is the counter part of the attribute specialist. It can even be that this is the architect of the component model if the component model is proprietary. If the component model is already defined then depth knowledge about the model is required.

A s*ystem specialist* has deep knowledge about the target environment. It is of vast importance to know the target platform inside out, since its behavior usually affects the performance of the components residing in the PECT. The PECT may need to be designed to restrict the use of the environment to meet the set goals. For example if the kernel-programming interface of the operating system is used freely, it will definitely be much harder to achieve any convincing predictions. Therefore, a restricted use of the environment might be a necessity, however the domain expert has a say in this.

The *domain expert* knows how the PECT will be used from an end customer perspective. The expert provides also input about existing standards, methods and user profiles. The PECT is restricted in its design by the intended usage of the system but also from the set of prediction requirements. It is the domain experts' role to make sure that the PECT is not overly restricted and cannot provide applications with intended functionality. For instance, the domain expert will negotiate with the system specialist about how much restriction to put on the use of the operating system.

The *customer* knows what applications are to be produced using the PECT. Apart from that, the customer states the statistical goals and quality and functional requirements of the PECT.

4.4.1 Definition Phase

The definition phase has two parallel paths, shown in Figure 19, one to define the model problem and another to gather the requirements for the property of interest. The *model problem* defines the functional requirements and their priorities. It is the problem definition that the PECT will solve. Ordinary requirements engineering techniques can be used to perform this task, examples are found in [98].

If the one track is capturing the functional part then the other will identify the prediction goals of the quality attributes. First, we define what quality attribute is of interest and how this is represented as an assembly property. Then we define the goals to achieve for these properties. There might be other quality attributes, not subject for prediction, the PECT must fulfill. These attributes and the requirements they impose have to be captured in the first phase where the model problem is defined.

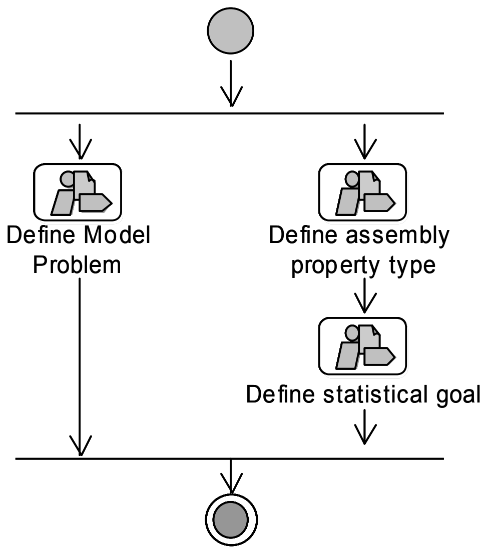


Figure 19: The definition phase with two parallel definition tracks

The type of the quality attribute to be handled will determine later what analysis technologies we could use to build our prediction theory. Quality attributes are defined as properties on the assemblies that will be predicted using the PECT. The attribute specialist together with the customer specifies the type and the desired goals.

In addition, the goal for a normative confidence interval should be defined. This is basically the statistical goal for the prediction, i.e. the prediction provided must be within a magnitude of relative error of some percent with a certain confidence. For an example, see Figure 16 on page 57. Table 2 summarizes the activities, artifacts and workers for the definition phase.

Table 2: Definition phase activities, artifacts and workers

Activity	Artifact	Worker
Define functional requirements	PECT requirements	Customer
Define assembly property type	Property definition	Customer
Define normative statistical goal	Statistical goal	Customer

4.4.2 Implementation and Co-refinement Phase

The main idea how to build a PECT is to make progress through co-refinement of a component model and an analytic model. The first step of co-refinement is to merger the restricted component model and the analytic model.

An interoperation of a component model for the analytical model must be defined. This interpretation will identify which properties from the components are used in the property theory in the analytical model.

The analytic model is then used to produce a property theory that states how attributes shall be predicted for the specified component model. That is, a theory for how to calculate the assembly property from the component properties is developed. An example of co-refinement where different property theories are created and where the goal is achieved in steps can be found in [43].

During the implementation the most important part is the creation of the component model, analysis model and measurement framework. As shown in Figure 18, these are created in parallel but not without affecting each other. All three depend heavily on each other and a design decision in one of the models may affect the others. This is because the three models are integrated into one constructive component model after each iteration step in the method.

A measurement apparatus must be developed for both components and assemblies. Each component has to be measured to obtain its quality attributes and a component test bench is to be developed to gather the properties in an easy fashion. The component test bench might also be packaged with the PECT for later automatic testing of the components during deployment of the components in the target environment.

In a similar manner, an assembly measurement apparatus must be developed. Sample assemblies created in the PECT have to be measured to validate the prediction stated about the assembly. To do that in an automatic manner we need to develop a measurement framework that also might be packaged for later real world application validation. For more information, about metrics and measurements of software see [34].

The outcome of the integration into a constructive model is called a PECT instance, since this constructive model has some predictable capabilities and is the asset that is validated. During the integration the property theory is embed in the constructive model. When the property theory is embedded then we have automatic interpretations of assembly instances from the component model to the analytic model. Desired predictions can be made analyzing such an instance.

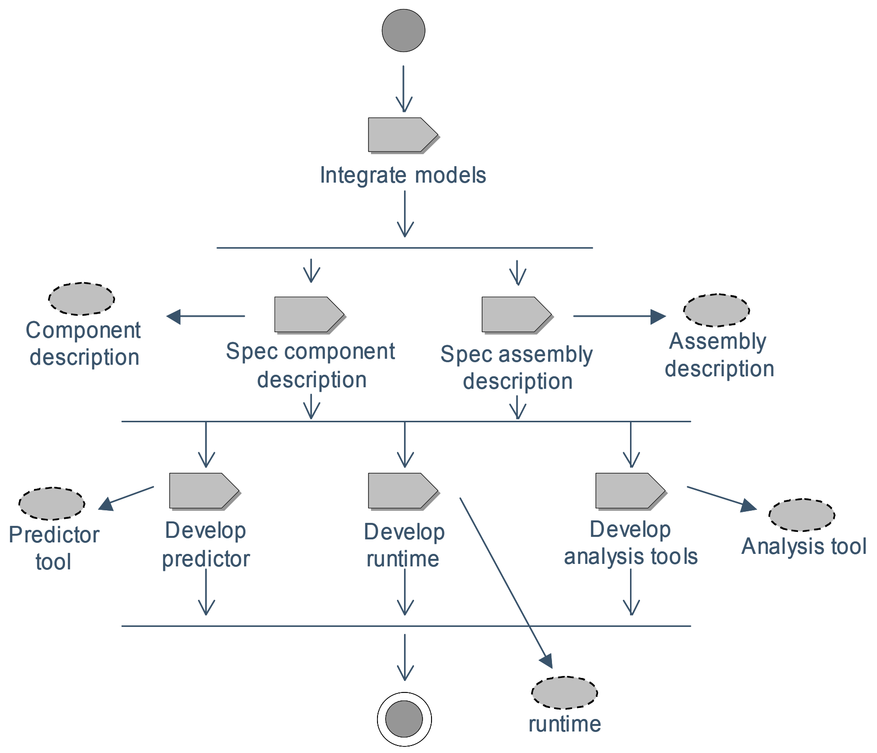


Figure 20: The integration of the different models

Figure 20 shows the integration part of the method in more detail. After the component model, analysis model, and measurement framework have been integrated, certain development activities will take place. The way components and assemblies are described is specified and that generates two artifacts, namely the component and assembly description.

When the two persistent description formats are decided upon it is possible to develop runtime, predictor and analysis tools. The runtime tool takes an assembly description as input and instantiate all the components and sets up the connections between the components. The runtime tool also starts the assembly after all properties have been set for all the components. To make predictions and analysis of the assembly, tools that take the two description formats are developed. These tools work with the assembly and components in an analytic way without actually running the components in the assembly. The analysis tools work on taking components and instantiating them to do the analysis, while the predictor tool works on mere theory and information about the components. I.e. the analysis tool work on concrete components while the predictor tool works on abstract representations of the components. The predictor tool is used for validation purposes and implements the same algorithms as the analytic tools. The analytic tools are packaged together with the PECT. As shown in Figure 18 the constructive model or PECT instance is validated both empirically and functionally after the integration step. The empirical validation is covered in Section 4.5 in more detail. How to perform the functional validation is not covered in this thesis. If the PECT instance has sufficient predictable powers then it is packed and the development is brought to closure.

Table 3: Co-refinement and implementation phase activities, artifacts and workers

Activity	**Artifact**	**Worker**
Create or define component model	Component model	Component model specialist
Define analytic model	Analytic model	Attribute specialist
Define property theory	Property theory	Attribute specialist
Create measurement apparatus for component and assemblies	Measurement framework	Measurement specialist
Embed property theory in constructive model	Automated reasoning capability	PECT designer
Integrate component and analytic model with the measurement framework	Constructive technology	PECT designer
Develop analysis tool	Analysis tool	Attribute specialist
Develop runtime	PECT runtime	PECT designer
Specify description schemas	Component and assembly description schemas	PECT designer
Develop a tool that can create predictions given assembly and component descriptions	Predictor tool	Attribute specialist

4.4.3 Validation Phase

When a PECT instance is created, we have a constructive component technology that can be used to produce real components and assemblies. We must validate the PECT to obtain the statistical label that we aimed for in the definition phase. As Section 4.5 describes the process how to conduct an empirical validation this section only describes briefly the validation phase.

The phase starts with refining the goal and the property of interest together with the current property theory. Note that both the goal and theory has been developed earlier in the process of developing the PECT instance, for the empirical validation, it is just important to restate this information. Now, in the validation phase, we merely restate the theory to know explicitly what theory we are using for the current experiment. Second, we describe the experiment, the test environment, and the sample selection process. In addition, it is important to characterize the different possible assemblies in the PECT to select a good representative sample set of the targeted problem domain.

After collection of sample data it is possible to analyze the results and these might indicate that the experimental setup is not sufficient to produce the wanted results. For example the measurement apparatus might be faulty or the selection of the sample might be non-representative of a realistic assembly population. If it is not sufficient, then we go back to the second phase to refine the technology and experimental setup until satisfaction is achieved.

As an outcome of the empirical validation, there are several artifacts. These artifacts are test components, assembly generators, sample set and results. The test components can be synthetic, simulating real components and they are used to be able to create all the different assemblies needed to do the analytic study.

Synthetic components are also required because we do not yet have the real components that will be used to build the end user applications in the PECT. This approach might be troublesome if certain quality attributes depends on the use of real components. If a set of real components and assemblies were available, an enumerative study could have been performed. An enumerative study works with an existing population compared to the analytic studies that address a fictive population.

The assembly generator is used to produce the sample set of assemblies given certain characteristics. Sample characteristics are decided upon when the PECT is bounded and during sample selection as explained later. A sample set consists of a number of assemblies described in the defined format and the set has to be stored for possible later verification.

Table 4: Validation phase activities, artifacts and workers

Activity	Artifact	Worker
Create synthetic components	Test components	Component developer
Create assembly generator	Assembly generator	Measurement specialist
Create validation sample	Sample set	Measurement specialist
Obtain component measurement	Labels for the components	Measurement specialist
Measure and calculate statistical label as result	Statistical labels of the property theory	Measurement specialist

4.4.4 Packaging Phase

In the packaging phase the PECT is prepared for deployment under the leadership of the PECT designer and the domain expert. It is of vast importance that the delivered PECT fulfills the customer requirements and is easy to use. After probably several development iterations the PECT will have a statistical label that describes the characteristics of the property theories provided in the PECT. Normal activities such as documentation and installation support are part of the packaging phase but they are not the fundamental goal of this phase. More important is to capture the ease of use requirements when it comes to stating predictions for the user.

The analysis tools developed should be packaged in a way so it is easy to analyze a created assembly. An ambitious PECT designer might even have provided a graphical construction environment that can be used to create component instances and connections between those. This development environment might provide dialogs for stating queries to the analytic underlying tools that will provide predictions for the user.

If components have been developed during the PECT development time, then these components can be packaged together with the PECT to serve as examples or reusable components for the application development. The components could be automatically deployed when the PECT is installed.

Measurement frameworks developed should also be packaged since in many cases the components must be re-measured due to dependencies on the target environment. For example, execution time of a component that depend on the actual hardware it executes on. Also the framework can help the application developer to test the application to a greater extent.

Table 5: Packaging phase activities, artifacts and workers

Activity	Artifact	Worker
Create deployable PECT	Packaged PECT	PECT designer
Make measurement framework available	Packaged measurement apparatus	Measurement specialist

4.5 Empirical validation

This section describes a method how to make an empirical validation of a PECT. The section starts with an overview followed by a stepwise description of each validation step.

One of the important steps in the development process of a PECT is the co–refinement, the process by which we refine the component and analytic models together to create a PECT that can meet the prediction goals that have been set. After a co-refinement step, the predictive capabilities of a PECT is compared against the goal by doing an empirical validation. This validation consists mainly in comparing the predictions produced by a PECT against the actual measured properties of the assemblies. As this is a straightforward approach, although the measurements can be complicated, the selection of the assemblies to sample are not so straightforward. A major question that we focus on in this section is what assemblies to pick to validate the PECT.

The empirical validation is based on a statistical enumerative study and is intended to accept or refuse a property theory. An enumerative study analyzes the capabilities of the PECT for a given sample of assemblies. If the prediction error falls within acceptable bounds, we accept the validity of the property theory. Otherwise, we conclude that the theory has been falsified and another iteration of the co-refinement and implementation process is needed.

To validate a prediction theory for properties, it is necessary to select a sample set of assemblies that are representative of all possible assemblies creatable in the PECT. To decide upon what samples to choose, there must be a means to limit the selection space. The following three steps give an overview of the empirical validation procedure.

- First we need to describe the characteristics of assemblies to be able to categorize and classify them. The characteristics of an assembly are described by assigning values to certain variation points. The variation points represent different dimensions in which the assembly may vary, for instance size is a variation point.
- Second, when we have different categories of assemblies, we can choose our desired sample set by taking assemblies from each category. We want to make sure that the assemblies in the selected set are different enough from each other to be suitable for the validation of the prediction theory.
- Third, after we have a sample set of assemblies, we will predict the desired property and measure the property for every particular assembly from the sample to verify them. To avoid measurement errors the measurements should be taken several times and statistically processed. The predictions for all the assemblies in the sample set have to be verified in this manner, to validate the prediction theory as a whole.

A central principle of our approach is that the predictive capabilities of a PECT must be rigorously validated, and the component properties that underlie these predictions must be accurate. Both are essential if design predictions, and the components that they depend upon, are to be trusted and possibly certified

To assign the statistical labels to a PECT, we need to have a systematic approach. The following approach to perform the empirical validation has been used to validate prediction theories in two different PECTs, which are used to build operator logic and controllers for a typical industrial automation system [43].

The empirical validation procedure is described in more detail in the following five sub sections and the steps shown in Figure 21.

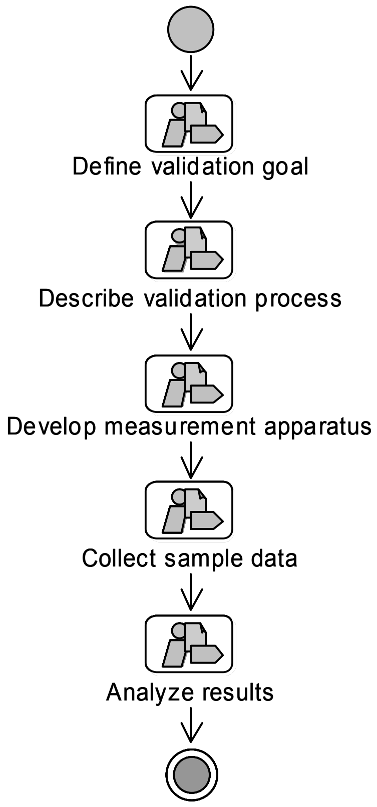


Figure 21: Process of empirical validation

4.5.1 Define Validation Goal

The property of interest is part of the PECT requirements and has been decided upon in the definition phase of the PECT development, but as it is possible that there are several different reasoning frameworks for properties built into the PECT we have to define which property this particular validation experiment aims for.

Apart from the property theory, there is also need to specify the property of interest for the components. The component properties are used to calculate the assembly properties and they do not have to be of the same type as the assembly property.

When the component properties depend on the environment, which they often do, they have to be measured in the target environment and ultimately certified for the component in question. If the component properties can obtained, quantified and certified, then we have the base for achieving predictable assembly from certifiable components. The benefit of having certifiable components is to have the possibility of objective trust in the specified component properties.

As a hypothesis, the property theory is sound over all the assemblies that can be produced in the PECT. That means that any assembly created using the PECT has a property that can be predicted with the existing theory. Since it is impossible to empirically validate the theory for an indefinite set we have to restrict the theory and the sample set.

The objective of empirical validation is to know how good the predictions made in a PECT are. There are several ways to validate if the predictions are accurate or not. The goal is to have a statistically labeled PECT, which is empirically validated. The statistical analysis is centered on the magnitude of relative error (MRE) of the predictions (See equation 20 on page 56).

The validation can have two goals, which are reflected in the end by two different statistical labels. In one case, there is a prediction requirement that has to be met, for instance, the MRE for the predictions has to be no greater than 0.5%. We call this a normative validation, and the resulting label will state the percentage of predictions for which the norm or requirement will be met with the PECT being validated.

A second kind of goal is to determine the predictive power of a PECT with no specific prediction requirement to be met. For example, we want to determine the bounds for the MRE of 80% of the predictions. This is an informative validation, and it renders a label indicating bounds for the MRE for a given proportion of the predictions.

The goal of the empirical validation has to be clearly stated before performing the study. For both informative and normative validations, we need to set the confidence level (i.e., the probability of the statistical results being right). Usually, either 95% or 99% confidence levels are used [88].

For an informative validation, we need to define the percentage of predictions that we want to include in the bounds for the MRE. Due to the nature of this validation, to inform the user of the PECT of its predictive power, there is no pass or fail test for the validation. Nevertheless, if the resulting bounds for the MRE are too big for any practical use, it can be concluded that more co-refinement of the PECT is needed in order to improve the performance of the PECT.

For a normative validation, we have beforehand a prediction requirement that has to be met, and that is the goal. However, this alone does not give as a pass/fail condition to test for, because the statistical method will render the bounds for *p*, the percentage of the predictions that will meet the requirement. If we set a minimum acceptable value for *p* upfront, then we can test whether the validation succeeds or not.

4.5.2 Define Validation Process

The experiment must be described and documented to be able to recreate it later if needed. The description shall include the precision of the measurement of each component, since a lower precision of component measure will give a lower precision of the PECT prediction. The statistical error from the measurements of the assemblies is a mean to know the precision, if the error is large then the precision is low and vice versa. When the measurement of assemblies is subject to error it is important to document the number of measurements taken (i.e., samples) and their standard deviation so that the precision of assembly measurements is also available. This description should cover four vital parts:

- How the sample population is bounded and limited;
- What components are being tested;
- How the samples are selected;
- The use of synthetic composition into assemblies with predefined components.

The tests can be carried out in an iterative manner to gather more confidence in the result. It is in general recommended to start with a small selection of artificial samples and then move on to more realistic samples later. The validation is carried out after each set of samples have been measured and analyzed. There might be many factors affecting the result of the validation. The PECT implementation, the measurement apparatus, the synthetic components and how the sample assemblies are created, could all affect the validation. The aim for the validation is to validate the PECT but before we can rely on the results, it is necessary to minimize the errors from the test environment.

Samples will be selected according to the classification of the characteristics of the assemblies. It is possible that an infinite number of assemblies can be created in a PECT and it is important that during the experiment the selected sample set is as representative of the complete possible set of assemblies as possible. Characteristics can be captured in what we call *variation points*. The variation points are used to bind the space of possible assemblies.

Selecting which assemblies to use in the empirical validation as representatives of the indefinite set of all possible assemblies looks like a nontrivial task to perform. In fact it is not a trivial task but it is neither impossible. An assembly has certain characteristics that can be used to classify it and compare it to other assemblies.

For example, the number of components in an assembly does give us a number to distinguish a large assembly from a small one. If we know, for instance, that no more than 50 components are realistic in one assembly, we also have an upper bound well defined and no sample assemblies with more than 50 components have to be selected. Other variation points are related to the prediction theory or the certain quality attributes.

If the size of an assembly is one characteristic, then size is said to be a variation point. Another example is that all assemblies that have only one thread of execution will behave differently from assemblies with multiple threads. Assemblies can vary from other assemblies in defined variation points. All variation points can be defined as one dimension of variation. If assemblies have many variation points then they together will constitute an N-dimensional hyperspace of assemblies.

First, the different variation points are to be identified. There are suggestions of how to identify these architectural variation points called sensitivity point in the ATAM for architectures [61]. The ATAM definition of a sensitivity point is "A sensitivity point is a component or decision made in the architecture that is critical for the achievement of a particular quality attribute" [61]. One approach is to identify the sensitivity points which in general are variation points. Variation points shall be independent from each other. If a variation point can be calculated from other variation points then it is not needed and can possible be removed. If included the variation point will only introduce unnecessary complexity. On the other hand there might be dependencies between variation points that are hard to quantify, and these ought to be left in the characterization model.

Examples of variation points are number of components and performance. All variation points $v0$, $v1$ to vn, are dimensions in the variation hyperspace which can be described with a tuple $V= \langle v0, v1, \ldots, vn \rangle$. Too many defined variation points results in very complex domain of possible assemblies and should therefore be avoided.

After the variation points and the boundaries are defined, we say we have a bounded hyperspace of possible assemblies where the variation points are the dimensions bounded by the defined limits. The hyperspace serves as a body of assembly characteristics from which we pick a point as one sample. Each point in the hyperspace can inhabit numerous different assemblies that adhere to the same characteristics

A random sampling procedure provides best statistical data but it might be difficult to obtain a real random set of samples. On the other hand, convenience and judgment sampling are more practical and easy to perform, but the resulting statistical data might have less confidence.

Since all the variation points are not independent, we choose judgment sampling to pick the samples. Judgment sampling takes place when domain or other knowledge is used to set ranges of variation points. In judgment sampling there is knowledge defining a subset of realistic assemblies. The realistic assemblies are those that have characteristics matching the types of applications that will be created using the PECT in the future.

Using judgment sampling, we can achieve higher confidence in the property theories since the validation aim for the realistic subset of possible assemblies. The lion part of the sample set of assemblies should be taken from the realistic subset. This gives more confidence in the results since the PECT has been validated with sample assemblies that match the assemblies that later will be crated for real applications. As shown in the figure there are also some samples taken from the bigger set of possible assemblies. The idea behind is to spot check the property theory that it holds not just for the limited set of samples. If the spot samples deviate much from the other samples, it might indicate an underlying error that has to be investigated.

It is of importance not to limit the samples to just one corner of the hyperspace, and with that in mind, we pick different characteristics first from the boundary of the hyperspace and second with additional points all over the hyperspace. This gives us a sample set that represents and covers the different possible combinations of assemblies in the PECT.

Picking the variation points is easier if the PECT when is going to be used to develop products of a software product line [18]. In a product line, variation points are well defined together with the software architecture, making it easier to generate assemblies' representative of future products.

If there was a set of available components developed for the controller PECT it would be appropriate to incorporate them in the test as well as the synthetic components, but that depend on the component developer and the PECT requirements. The more realistic the assemblies picked for the validation experiment the more confidence we can have in the results obtained. An analogy is the development of medical drugs where tests on human beings do not start before testing in the lab with simulated behavior. The tests are stepwise taking into a more realistic environment.

For each selected sample characteristic, we generate a specific assembly that matches the characteristics. The set of generated assemblies is the actual sample set that has to be predicted and measured. Before each sample assembly is predicted, we need the values for the component properties used by the property theory. Unless these have already been provided by the vendor or certified by a third party, we need to measure all the components in that particular assembly to gather the desired properties.

After each assembly in the sample set is measured, the results are compared with the prediction made and the magnitude of relative error (MRE) is calculated. This gives us a sample set of measuring points and errors, and then used to do all the statistical analysis required for producing the label for the PECT.

4.5.3 Develop Measurement Apparatus

After the validation procedure is defined, the measurement apparatus has to be developed. This includes that we specify how conclusions are to be drawn from the results and a specification of the test environment where the experiment is to be performed.

The general approach for drawing inferences from a sample set is to make sure the number if samples are large enough to match the set statistical goal. For instance samples are run a hundreds times each to get the statistical labels for each sample or until the desired relative error between the measurements is less than the set goal, that is, the average MRE for that sample matches the goal. Each sample is run at least 30 times to guarantee a minimum quality.

When the statistical labels have been collected and combined, it is possible to get the labels for the PECT. The average MRE of all the sample runs and their MRE is used to label the PECT. A recommendation is to collect MRE, standard deviation, and the relative error of the measurement for each sample assembly run.

The inference process should also describe how each component is measured and how to combine results of all measures. There might be need to describe error propagation for certain types of PECTS. Tools could be developed analyze and measure components in an isolated context. These tools discover the analytic properties of the components and store the result with the component or separate from the component depending on the component model.

It is of importance that the test environment is described since it may affect the resulting properties largely. For instance, a performance latency property of an assembly will always be affected greatly by the computer environment, other processes running on the same machine can produce environmental noise in the measurements, and so on.

When the test environment is defined, it is also sometimes necessary to design and create tools and infrastructure for handling vast number of measurements and statistical calculations. The measurement apparatus designed into the PECT will be used to deliver the measurements for each sample but usually it cannot handle the measurement of thousands of samples. Measurements from samples could be stored for later analysis or the program that executes all tests can do the calculations needed on the fly. Of these two approaches, it is preferable to have an infrastructure that stores all data in raw format because there might be particular analysis that the user wants to perform later. If a program carries out the refinement of the data during the execution of each sample then it is not possible to do other analyses than those inhabit in the execution program.

There are several tasks, such as, generating samples, measuring components, measuring assemblies and predicting assemblies that have to be performed before the empirical validation is brought to an end.

The assembly generation task produces the sample assemblies that are used in the empirical validation. The sample assemblies can be created by hand and programmed individually, but automating this task increases the possibilities for having a larger set of assemblies. The larger sample set the better confidence in the results from the validation. If the population size is known it is possible to use statistical methods to calculate the needed size of a sample set for a certain confidence.

Assembly generation takes input from the defined assembly characteristics. A typical assembly generation tool knows how assemblies in the PECT vary and can take the variation input to produce assemblies. When an assembly is created, it is important to validate that it does not break certain rules of the PECT as the rules might be enforced or not by the PECT. Assemblies selected as samples for empirical validation must be valid in the PECT; one example is that the components together create a schedulable assembly if there are real-time components with timing constraints. A well-designed assembly generator should create only valid assemblies.

After the generation task has been performed, the resulting assemblies should be stored for later retrieval. The measurements of the samples must be performed several times depending on the wanted output and it is desirable to have the same sample set readily available. Having the same sample set makes it also easier to compare results if the PECT is improved or changed. Extending the sample set is also possible by running the assembly generator more times by adding more variation points and assembly characteristics.

After the process of the empirical validation is in place together with the description of the test environment, it is time to take a step back an evaluate if the defined process will achieve desired inference. In this phase, it might be necessary to try out the infrastructure and some of the components to get preliminary results and indications.

4.5.4 Collect Sample Data

The reasoning framework with the property theory is applied to all assemblies in the sample set to provide sample data. As each assembly is represented in the constructive model it is important to transform these into the analytical model, this is performed by applying the prediction tools in the PECT.

All component properties should be provided – either analytically (by calculation) or by measurement. A statistical process on these measurements can be implemented that is used to calculate mean values, deviations, etc. The component properties can be stored, in XML files or another persistent format, describing each component. These properties are then feed into the prediction theory and analytic model to provide a prediction. After the prediction is done, then each sample assembly evaluated is run for a defined number of times to gather the measured property or until desired average MRE is achieved.

It is very important to record how the samples were chosen and what the actual samples are. This is because we want to be able to recreate the same experiment later if the results are questionable. It is necessary to be able set to up and perform the experiment independently at another occasion. The PECT's labels are valid for all assemblies represented by the set of selected samples, but there might be other assemblies in the PECT, which will statistically perform different if the set of samples did not cover those assemblies, hence it is important to pick the different samples carefully.

4.5.5 Analyze Results

When the all sample data is collected the results can be analyzed. Normal statistical methods, for instance inferential or descriptive methods, can be applied to do part of these analyses.

All gathered data should be collected and analyzed together to get the final statistical labels for the PECT. When the labels are attained, it is possible to compare with the original goals set for the PECT. If the goals are not met then it is time for another step in the co refinement and implementation of the PECT.

4.6 Summary and Conclusion

This chapter describes the approach to take for achieving predictable assembly of components. The approach is to define a prediction enabled component technology that has prediction capabilities built in. When designing a PECT it is possible to use existing component technologies but that sometimes allow too much freedom to developers. A high degree of freedom might prevent the possibility to build prediction capabilities into a component technology. Instead, depending on the analytical model used it is important to restrict developers of components enough have the possibility to make predictions.

To take a non prediction enabled component technology and turn it into a PECT might be costly and hence it is important to analyze the benefits of introducing a PECT compared to the costs of not having it.

A method is developed in this chapter for building PECTs. Four phases defines the method and the first phase is where definition of the problem is made. In the second phase the different models and technologies for components and analysis and measurement tools are designed. The models might build on existing component technologies to reuse infrastructure. Validation of the PECT to see that ther requirements are fulfilled takes place in the third phase. The forth phase is the packaging of the component technology in a usable format.

The validation method consists of five steps, first the validation goal has to be defined. Second step is to describe the validation process. It might also be necessary to design and build measurement apparatus and that is performed in the third step. The forth step is to collect measurement data from the sample defined and that data is analyzed in the fifth and last step.

5 A PECT Experiment

Methods for building and validating a PECT are presented in the previous chapter and in this chapter, we exemplify these methods by the implementation and the validation of two experiment PECTs.

The quality attribute we investigate in these component technologies is performance which is represented as a latency property of assemblies. After the PECTs are built, we validate them and conclusions are drawn. The validation process described in Section 4.5 is used to perform this task.

We show the PECTs which consist of, a constructive model, an analytical model and an interpretation of the constructive model for the analytical model. We also identify which properties on the components that are relevant for the latency. Further we identify which type of interpolation of the component model should be included into the PECT.

We have two different model problems that result in two different PECT implementations, the operator and the controller PECT. These model problems are suitable for a research setting and they are abstract representations of real world problems. The controller PECT is built to execute tasks in a real-time environment while the operator PECT is built to show operator displays in an automation system. A simple automation system contains a controller and a human machine interface (HMI). The controller is responsible for controlling input and output signals to industrial machinery or devices, such as motors, valves and I/O. An operator display is part of the human machine interface (HMI) that is used to supervise the industrial process.

The chapter starts with a presentation of the goal and scope in Section 5.1. Section 5.2 gives an overview of the component model used called PIN. The first PECT, used to implement the operator model problem is presented in 5.3. The controller PECT is described in detail in Section 5.4. A conclusion of the chapter is presented in Section 5.5.

5.1 Scope and Goal of Chapter

The goal of this chapter is to describe two experiments implemented using the methods presented in the previous chapter. For the two experiments we show the validation method applied step by step.

The scope of the chapter is to exemplify how a PECT can be built and validated and how prediction results can be analyzed. The scope is not to show how to calculate latency, for that we refer to the real-time community and publications such as [62]

Statistical analysis is not the main scope of the chapter, but it is used for illustrating the validation of the prediction capabilities of a PECT

We contribute with an implementation of a PECT according to the defined method for a particular quality attributes. An additional contribution is the demonstrated validation of the implemented PECT using the defined validation method. We implemented the operator PECT and contributed to the implementation of the controller PECT.

5.2 The Component Model PIN

The component model used in this chapter is called PIN and is using a port-based architecture similar to the one defined in Section 2.6.3. Each component has sink and source pins to which other components can connect. A component takes as input its stimuli data provided on the sink pin and produces a response output on the source pin. The pins are the interface to the components but there are also configuration properties that are used to define initial state of the components. When a component is loaded into the runtime, each configuration property is passed to the component as initialization data.

Connectors are used to connect a component source to a sink of another component. The use of the connectors is defined by the assembly description. The assembly defines what components are included and their configuration properties together with the connectors between each component.

The source pins are of two types, asynchronous and synchronous. A request submitted on an asynchronous pin does not make the caller wait for the result. The caller sends the request and then the caller resumes control and continues to execute. A scheduler decides when the called component executes, depending on the priority and the current situation of the running components, for instance the component cannot be executed before all higher priority components are finished. A component that has received a request on an asynchronous pin must have its own thread of execution, it cannot execute on the caller thread.

A synchronous invocation is performed when the caller waits for the result of the request. Synchronous pins can be implemented with the execution on the caller thread or with an own thread. Synchronous pins can also be blocking or non-blocking. A blocking pin takes a semaphore or mutex to protect a critical section before execution.

XML is chosen as the persistent format for assembly and component descriptions for convenience reasons. XML is a well-defined format to store configuration and definition data, mainly because it is text based with many different tools available to work on the data. An example part of one operator PECT assembly described in XML can be found in Figure 22.

```
<?xml version='1.0' ?>
<Assembly xmlns="PinTekXML.xsd">
   <Components>
      <Component name="Gauge" type="SEI.Components.SEIGauge">
         <Position x="150" y="0" w="150" h="150"></Position>
      </Component>
      <Component name="Led" type="SEI.Components.SEILamp">
         <Position x="200" y="200" w="50" h="50"></Position>
      </Component>
      <Component name="Close"type="SEI.Components.SEIButton">
         <Position x="0" y="0" w="150" h="50"></Position>
         <Property propId="1" value="Close"></Property>
      </Component>
      <Component name="Switch" type="SEI.Components.Switch">
         <Property propId="0" value="OPC.Fix.1"></Property>
         <Property propId="1" value="Led1.Status"></Property>
         <Property propId="2" value="Switch2.Pos"></Property>
      </Component>
   </Components>
   <Connectors>
      <Connector>
         <Source component="Close" pin="0"></Source>
         <Sink component="Switch" pin="0"></Sink>
      <Connector>
   </Connectors>
</Assembly>
```

Figure 22: Part of the XML description of the assembly in Figure 27

Figure 22 shows four components of different component types, SEIGauge, SEILamp, SEIButton and Switch. Each component has certain configuration properties that are set when the component is instantiated. Typical properties for a user interface component are the coordinates on the screen where the component is to be positioned. A XML schema definition file gives the syntax definition of the assembly. In Figure 22 it is denoted PinTekXML.xsd.

The assembly description also describes how the components are assembled together using connectors. The connector in Figure 22 shows that source pin number 0 of the Close component is connected to the sink pin number 0 of the Switch component. The other connectors have been omitted from the figure.

The components in PIN are described in a similar manner in XML with all the component properties specified; one can say that the component description is the specification sheet of the component.

Assemblies in PIN can have different topology patterns and each assembly can inhabit many different patterns at the same time. For example, an assembly might have both loops and branches as shown in Figure 23. Topology patterns are used to identify how complex the topology of an assembly is. By having the topology defined we have a variation point for the later validation of the PECT prediction theory. An example pattern is the regulator with feedback (Figure 23) which has a loop in it.

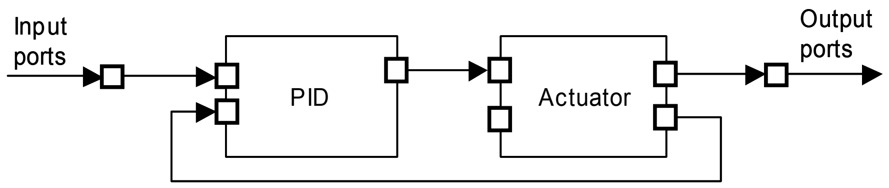


Figure 23: An example of a topology

In the example, there is a loop from the source of the actuator to the sink of the PID regulator. Such loops will vary the behavior of the assemblies; hence the topology rules shall be considered when generating valid assemblies.

Rules for both what is allowed and not allowed in an assembly ought to be defined for the analyzed PECT. An example of what is not allowed, due to execution rules, in PIN is to connect a source of a component to its own synchronous sink as shown in Figure 24.

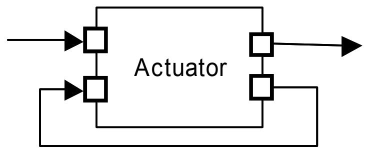


Figure 24: An invalid topology

Our way to represent each topology pattern is to number each pattern using bit representation for easy merge of different topologies. The combination of the topology patterns is then merely the union of the different bit representations, i.e., all patterns are represented by a bit position. For instance, we represent a loop pattern with the number 0x1 and a branch pattern with 0x2.

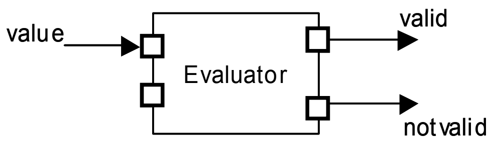


Figure 25: The logic of the component determine which of the sources that react on the sink

There are different means to specify reactions to what happens on sink pin and one approach is to use a notation taken from CSP (Communicating Sequential Processes) [48]. A CSP specification as the one in Figure 26 describes the reaction for the evaluator component in Figure 25:

```
EVALUATOR = value?x:N→if x>15
            then valid → EVALUATOR
            else notvalid → EVALUATOR
```

Figure 26: A short CSP expression describing the reaction to a stimuli on a sink pin

Which one of the sources named `valid` or `notvalid` that will trigger is depending on the value of the sink. If the sink value for the evaluator is greater than 15, the source named valid will trigger. In this example either the valid or invalid source pin will be triggered but not both.

We use the PIN component model [44] to implement the operator and the controller PECT. Two different implementations are provided one for each PECT. The constructive design of the operator component technology is built on .Net technology using Windows 2000. The controller PECT is designed in C and using real-time extensions to windows 2000 to acquire the real time performance.

5.3 The Operator PECT

This section outlines the operator PECT with the purpose to predict the latency for the human machine interface (HMI) applications and components. The operator is monitoring and controlling the automation system through the use of the HMI which is communicating with the controller in order to get data to monitor or setting execution orders for control.

The problem addressed in this experiment is the problem of predicting the latency from the components and connections that build a controller and HMI application. Another latency problem addressed in this experiment is the time it takes from a signal from the controller until the controller has responded with a corresponding output signal.

For the operator PECT the execution model is simple with only one thread of execution which gives no parallelism. Invocations between graphical user components are implemented in a synchronous manner which gives a single thread of execution once a component has been invoked.

The example of an assembly of display components is shown in Figure 27 where there are buttons that control the lab equipment and meters that monitor the running lab equipment. The operator can use the HMI to control and supervise the example process.

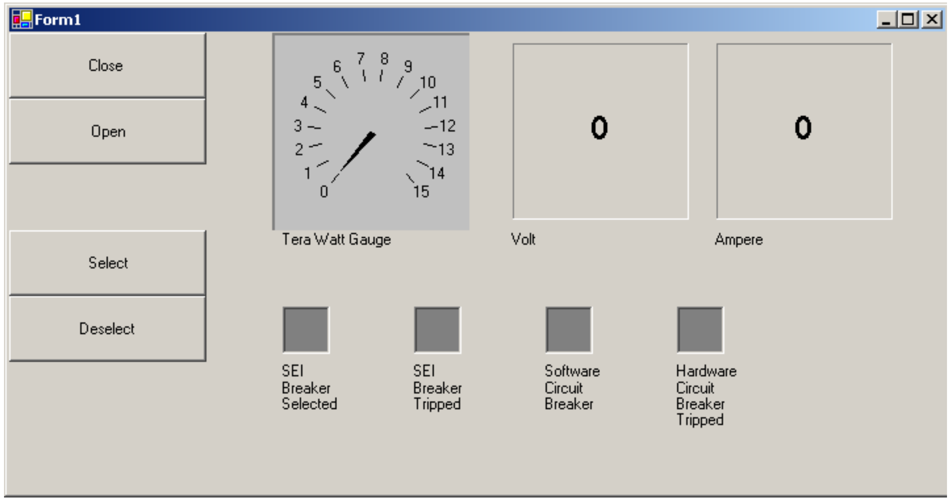


Figure 27: A HMI display with buttons, LEDs and Meters

After the constructive part was designed together with the infrastructure measuring, the first instance of a PECT was available. The PECT was validated by following the steps defined for empirical validation in Section 4.5.

5.3.1 Analytical Model

Latency is the time it takes to execute a request received on a component sink pin. The end of the request can be marked as when a result is provided on the same or another components' source pin. An example of latency is the time it takes to execute an assembly from time T_A to T_B, or the time to respond on a request on component C1. Measurements points are defined as pins on a component where measures can be taken as shown in Figure 28.

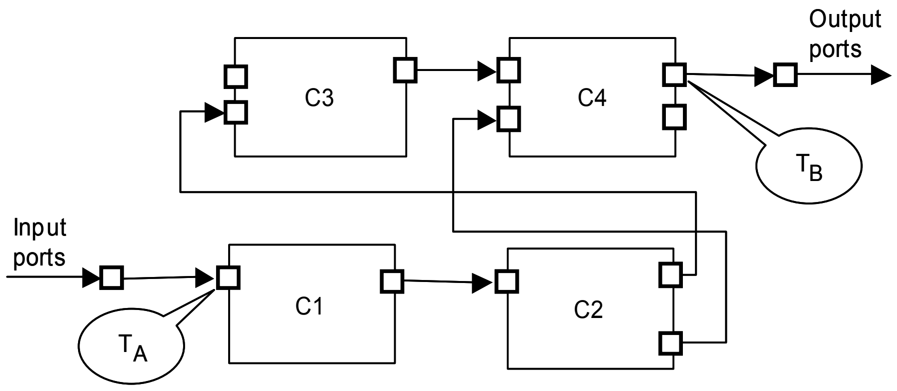


Figure 28: Measurement points T_A and T_B for latency are defined on the pins of components

For the analytic part we have chosen a simple theory for latency in the operator PECT on the assumption that execution times for each component could be added together, i.e. the execution time for each component in the path defined by the assembly is added. This assumption is valid under the condition that there is only one thread of execution and no parallel execution. The latency for an operator assembly with respect to a execution path, *L*(*A*, *P*) is

$$L(A,P) = \sum_{\forall C_i \in P} E(C_i) : P \subseteq A \times A \qquad (21)$$

, *P* is the path of components that will execute in the assembly

E(C_i) = Execution time of component C_i

Taking the summation of the components execution time might not be the most representative theory in the reality but it is good a starting point for the first iteration of co-refinement of the theory and models.

A prediction theory that would suit better would consider interference from surrounding environment and the overhead time introduced when components are connected. As illustrated in Section 4.4.2, the method of building a PECT includes co-refinement with an iterative approach. If the measurements taken during the validation indicate that the theory is not representative, then the theory, the constructive technology, or the measurement infrastructure need improvement. This example will also be discussed in Section 5.3.6.

5.3.2 Validation Goal

The design objective of the operator PECT is to enable prediction of the performance for issuing operator commands and handling controller notification with magnitude of relative error MRE ≤ 0.10 and confidence level $\gamma = 0.95$. This means that a prediction stated for an assembly will have a MRE less then 10% with a confidence of 95% in that prediction. Desired is also that the correlation of predicted and average measured latency is very high. The goal is formulated in Table 6.

Table 6: Statistical goal for operator PECT

Descriptive Statistics	Value
Mean MRE	< 10%

Confidence level γ	95 %
Spearman rank correlation of predicted latency and average measured latency	> .98 p-value < 0.05

The property of interest for the operator PECT is latency. We will predict the latency from one sink pin to a responding source pin.

5.3.3 Validation Process

Characteristics of the assemblies are defined for the operator PECT as a tuple with variation points. From the domain knowledge, it is possible to derive that there is no multithreading or blocking asynchronous calls taking place. Hence, for this example it is not required to vary along these dimensions in the hyperspace of possible assemblies.

Each assembly in this PECT will be characterized by a N-Tuple of variation points $V= \langle v_1, v_2, \ldots, v_n \rangle$, and the analytic study will vary the following assembly variation points:

1. v_1 = Set of canonical topology rules
   The topological rules define what layout of the assembly is present;

2. v_2 = Size ⟨#Components, #Connections⟩:
   The size is given in the number of components and the number of connections we have from each source pin;

3. v_3 = Performance ⟨ lower execution time, upper execution time⟩:
   The lower execution time indicates that a component cannot execute faster than this limit. A component cannot execute slower than the upper execution time.

Size and performance rules define variation points. In the operator experiment, the size varies from four components to 40 with the number of connections per component from one to 10. The performance spans over three groups having all components with and average execution time in the range from 5 to 15 ms, or range from 100 to 150 ms, or a mix of both ranges as shown in Figure 29. Average execution time is obtained by measuring the components in a test environment.

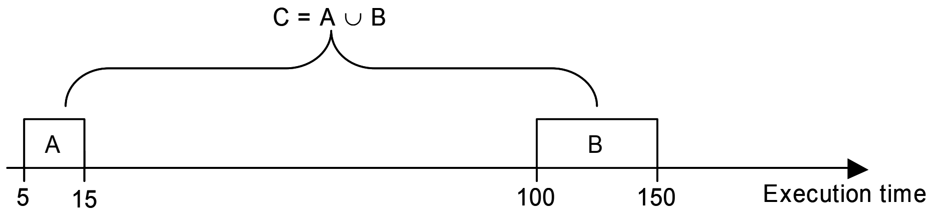


Figure 29: There are three sets A, B and C from which performance sampling is taken

The performance variation gives an indication of the execution distribution of components in an assembly and measurement of the components and assemblies will discover the real average execution time. A test bench is used to measure all components and each sink pin is measured at least 30 times and maximum 150 times, and then the data is recorded together with the standard deviation of the measurements. Figure 30 shows that the measured execution times for the five components C1 to C5 are roughly 10 ms and this average execution time will be used when predictions are later carried out.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Components>
   <ComponentProperties name="C1" pin="0" samples="30"
     avgElapsedTime="9.780041" stdDevElapsedTime="0.0262" />
   <ComponentProperties name="C2" pin="0" samples="30"
     avgElapsedTime="9.744655" stdDevElapsedTime="0.1778" />
   <ComponentProperties name="C3" pin="0" samples="30"
     avgElapsedTime="9.820084" stdDevElapsedTime="0.1776" />
   <ComponentProperties name="C4" pin="0" samples="61"
     avgElapsedTime="9.792901" stdDevElapsedTime="0.4214" />
   <ComponentProperties name="C5" pin="0" samples="30"
     avgElapsedTime="9.711122" stdDevElapsedTime="0.1260" />
   <ComponentProperties name="C6" pin="0" samples="150"
     avgElapsedTime="0.175674" stdDevElapsedTime="0.2820" />
</Components>
```

Figure 30: Example XML describing the latency property of several components

The measurements were performed assuming only one thread of execution, communication over synchronous calls and that components have only one sink.

To facilitate generation and execution of different assemblies we have built a special-purpose component of type (named SEI.Component.SEITest). This component is configurable, through parameters, number of pins and rough execution time. This component does not have any environmental dependencies; hence, no GUI is provided. The configurable parameters are controlled via properties of the component and the component task is to consume CPU time.

Five different assemblies are chosen with different characteristics to represent the first set of samples. The component used in these assemblies will be of the SEI.Component.SEITest type.

Judgment sampling is used to retrieve the different assembly characteristics and then assemblies are generated using the characteristics as input. Each sample assembly is saved for later retrieval; since we want to have the possibility to regenerate the very same experiment later.

5.3.4 Measurement Apparatus

A test environment and a measurement infrastructure is designed to perform the experiments. The target machine is a Pentium P4 class machine with 512 Mbytes of memory and the following software:

- ABB Aspect Integrator Platform (AIP) – This program is used to manage HMI and connections from the HMI to the OPC server at the controller;
- Microsoft products - Internet Explorer 6, DirectX 8, Visual Studio.NET, Win2k service pack 2;
- .NET Runtime with service pack 1;
- All latest critical updates available from Microsoft update site.

First, a sample is selected and an analytic representation in the analytic model is created for that sample. Then the components in the assembly are measured individually to gather the analytic properties. The properties measured are the execution time of each component. The component properties are stored in XML files describing each component. These properties are then fed into the prediction theory (equation 21) and the analytic model to provide a prediction.

After the prediction is done, each sample assembly validated is run 1000 times to gather the measured latency that is later compared with the predictions.

5.3.5 Collect Sample Data

A few samples were selected with the following characteristics to test the procedure.

- ⟨0x0, ⟨4, 1⟩, ⟨10, 10⟩⟩ is a straight execution of 4 components with very small execution time. A component is connected to exactly one other component and all components form one chain of execution.
- ⟨0x2, ⟨4, 2⟩, ⟨10, 10⟩⟩ is set of 4 components with execution time 10 ms containing at least one branch, i.e. at least one component has two source pins connected to other components.
- ⟨0x2, ⟨20, 2⟩, ⟨60, 60⟩⟩ is a set of 20 components with 2 connections accommodating at least one branch. The execution time is 60 ms.
- ⟨0x2, ⟨30, 4⟩, ⟨10, 100⟩⟩ is a set of 30 components with 4 connections accommodating at least one branch. The execution time is distributed randomly in the range from 10 to 100 ms.
- ⟨0x0, ⟨n,1⟩, ⟨10, 10⟩⟩ is multiple samples with $n \in \{1,5,10,20,50,100\}$ number of components in a straight path with execution time lower and upper bound set to 10.

After the first samples were tried out, more samples were added and the data was collected for analyze.

5.3.6 Analyze Results

The different samples where executed and analyzed and after the first set of executed samples it was possible to get feedback on whether or not the sampling procedure is working and if there are any apparent errors. The results from three different sample sets are shown in Table 10,

In the first two sets, (Table 7 and Table 8), the number of components varies while the other variation points are fixed. We take this approach to investigate how size affects the predictions. For a thorough analysis of interdependencies between variation points a designed experiment shall be used. How to conduct a designed experiment with multi-factors is described in [14].

In the case where the number of components is changed while the topology is fixed, the theory is a sum of component execution times. It is reasonable to conclude that execution time of the whole assembly should follow the predicted number with certain error. That is, it is reasonable that the error should not grow depending on the number of components. This hypothesis is tested in this experiment.

Table 7 and Figure 31 show a case in which the number of components is increased and there is an unexpected phenomenon with a growing MRE. The table shows that the predicted latency is close to the execution time of the component multiplied with the number of components. It also shows the calculated MRE of the predicted value relative to the mean measured value. The standard deviation with error and relative error from the measurements of the assembly is also shown. Each assembly, up to a size of 100 components has been measured at least 30 times.

Table 7: Sample where the size parameter is changed but the error grows unexpectedly[2]

#Components	Predicted (ms)	Average Measured (ms)	STD	MRE
1	10.042	9.988	0.439	0.54%
5	50.675	50.180	0.773	0.99%
10	100.410	103.819	1.671	-3.28%
20	198.845	215.441	1.811	-7.70%
50	489.980	611.949	3.148	-19.93%
100	978.577	1468.827	6.200	-33.38%
1000	10000.000	59572.878	No Calc	-83.21%

Figure 31 shows that the measured value deviates in a growing fashion from the predicted value. This observation is crucial since it indicates an error in the PECT. The error can be situated in the implementation of the component technology, the analytic model or in the validation infrastructure. After the planned set of samples was run, it could be suspected that the MRE would grow exponentially. To verify that suspicion, a 1000 component sample was run in addition to the planned samples with the resulting 83% MRE that points out a major flaw in the PECT.

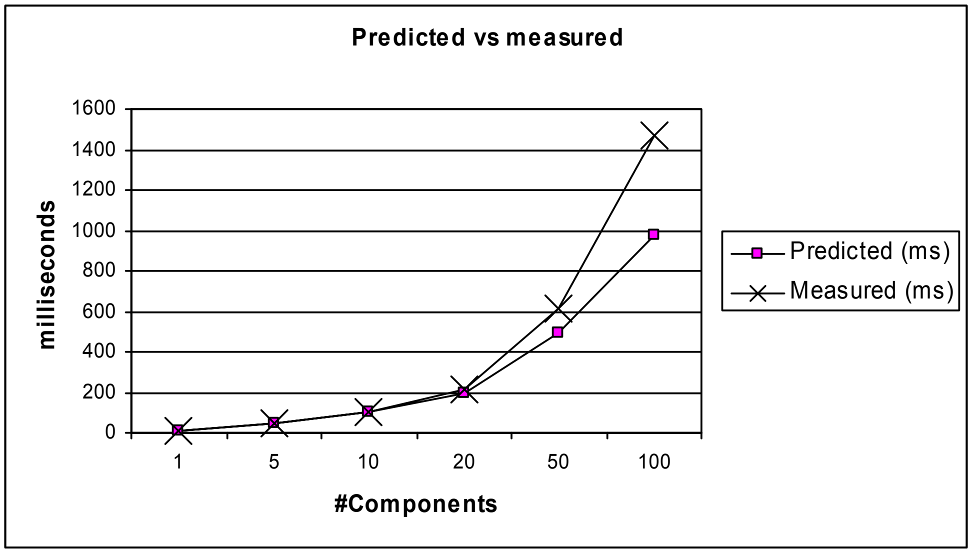


Figure 31: Predicted versus measured latency where large deviation is depending on design of load function

The measurement that indicated a fault in the PECT resulted in error tracking and new measurements after removal of the implementation fault. It was shown, after investigation, that the design of the synthetic components was faulty. Each synthetic component has a load function that generates CPU load by doing dummy work. The first version of the function was dependent on the

[2] The sample with 1000 components has a rough estimate of the predicted value and only one measured value, hence no STD

number of times it executed and that resulted in the enormous increase of execution time in large configurations. The fault was corrected and a new load function was designed that consumed time in a linear manner depending on input, which gave the desired behavior. Figure 32 and Table 8 shows what the measurements where with the new load generating function in place.

Table 8: Same experiment as shown in Figure 31 but with the error corrected in the PECT validation infrastructure[3]

#Components	Predicted (ms)	Average Measured (ms)	STD	MRE
1	9.469	9.780	0.017	-3.18%
5	46.743	46.994	1.025	-0.54%
10	94.191	92.858	2.275	1.44%
20	188.158	182.519	2.375	3.09%
50	469.725	453.506	3.609	3.58%
100	939.985	904.596	3.630	3.91%
1000	9300.000	9102.020	140.731	2.18%

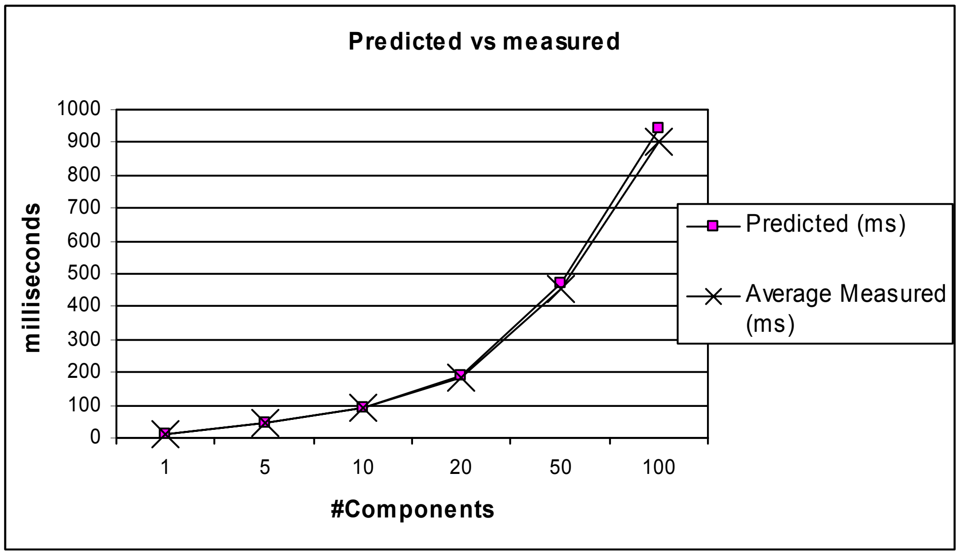


Figure 32: Deviation of predicted versus measured latency is less and in line with expected results after a fault has been removed from the PECT validation infrastructure

Another measurement approach was to change the variation point for the number of connections of each component. One experiment illustrated in Table 9 and Figure 33 shows the results when the number of connections has been changed and variation takes place in the number of components at the same time. Two different assemblies have been generated with the size of five components but with different interconnections between the components, hence the occurrence of two entries in Table 9 and Figure 33.

[3] The predicted latency for an assembly with 1000 components is rough and not calculated with any tool.

Table 9: Each component has five source pins connected to different components and the size is varied

#Components	Predicted (ms)	Average Measured (ms)	MRE	STD
1	10.46	9.96	4.9%	0.04
5	1210.11	1070.82	13.0%	6.03
5	3996.28	3553.21	12.5%	36.76
10	7471.76	7242.73	3.2%	62.50

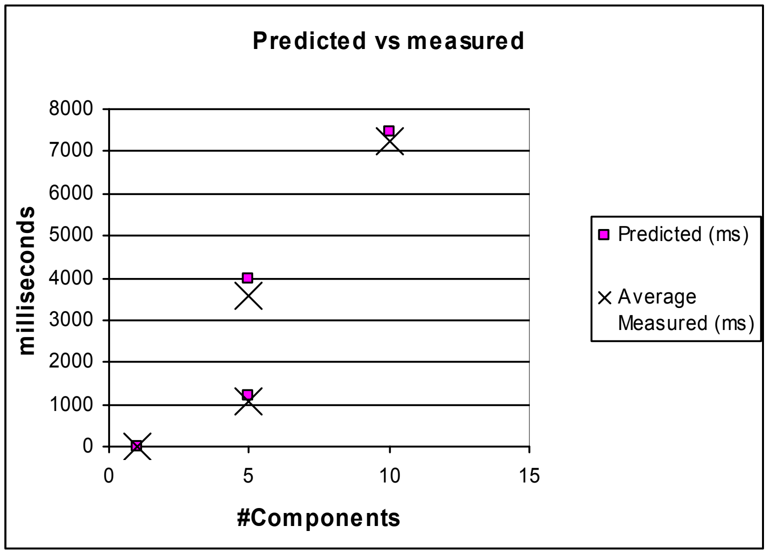


Figure 33: Measurements where each component has 5 source pins, i.e. the branch factor is 5

The measurement results of the connection sample indicate that the average MRE is roughly 10%. The overall results from the various samples give the following descriptive statistics as displayed in Table 10.

Table 10: Descriptive statistics of the operator PECT

	Value
Samples (N)	11
Mean MRE	4.8%
Median MRE	3.2%
Standard Deviation (SD)	4.11%
Spearman rank correlation of predicted latency and average measured latency	1.0 p-value < 0.0001

Shapiro-Wilk Normality test	Coefficient = 0.7425
	p-value = 0.0017

Spearman correlation is based on ranking the two variables, and so makes no assumption about the distribution of the values. A correlation data close to 1.0 is an indication that there is a strong correlation between the predicted and measured latency. The p-value is the probability that such correlation is a coincidence and as shown in Table 10 the probability is very small. The Shapiro-Wilk normality test shows whether the distribution is normal or not and a p-value greater than 0.05, which is the generally accepted level, indicates that the data follows a normal distribution. In the operator PECT example, we conclude that this is not the case and hence normal distribution statistical intervals cannot be used. In addition, there are too few samples in the first sample set to draw any real statistical conclusions.

As the operator PECT was too simple in terms of complexity of the execution model and the latency theory, no more work was done finalizing the work with for instance more samples. The execution model only had synchronous non-blocking calls in a single threaded system. Another reason for not continuing the development of the operator PECT was that it was shown by measurements in the validation phase that the time spent in graphical components in assemblies where neglectable compared to time spent in communication with the controller. Hence, the reason to develop an operator PECT diminished in the light of the more complex controller PECT. Typical graphical user interface components, such as buttons and meters, took roughly three magnitudes less time than the corresponding controller components.

As a concluding remark of the operator PECT it can be said that there are several possible explanations why expected behavior did not appear. It might be that not all source pins have been connected and that would result in strange execution behavior. Other reasons could be that compiled code is debug mode and executing with unexpected performance, the CPU could be overloaded leading to scheduling interference from other processes and program running on the machine or that a load function of a synthetic component does not provide the intended execution pattern. To improve testing and analysis, several special components with the purpose of testing the synthetic or real components could be developed. These would then be part of a test bench for acquiring component attributes in the target environment.

5.4 Controller PECT

The controller model problem is taken from a the substation automation domain and it was first described by Preiss in [86] and served as the basis for problem addressed in [43] and in this thesis. Figure 34 shows an example of substation logic and it depicts components of types specified in the IEC 61850 standard [50]. The physical circuit breaker Q0 is controlled by a software circuit breaker component. There are two measurement transformers T1 and T2 that are monitored by the software components TCTR and TVTR. The TCTR measures the current and the TVTR measures the voltage.

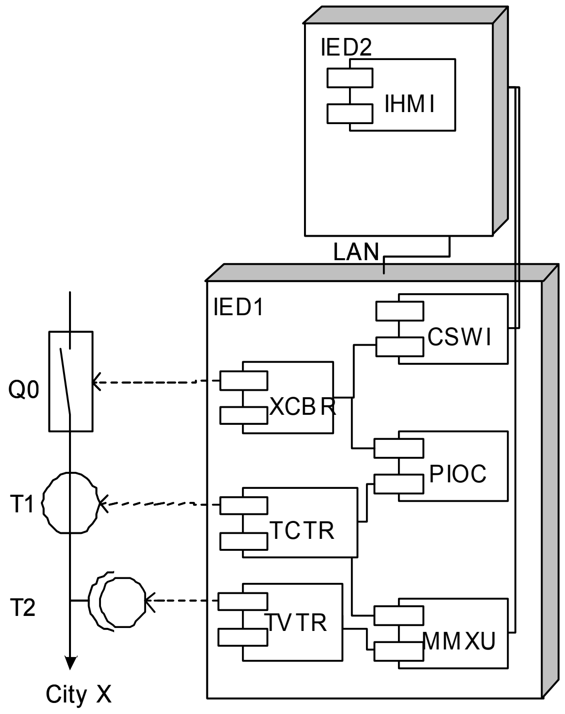


Figure 34: The substation model problem represented using the IEC 61850 standard components

The MMXU calculates the effect using the current and the voltage as input. Over-current protection is implemented in the PIOC component and it opens the circuit breaker in case of to strong current on the line. If the over current protection component does not react within a 100 ms, when the current exceeds the set limit, there is a risk that electrical equipment down the line will blow apart as the current might reach 50K Ampere in short circuit.

Overall switch logic is implemented in the CSWI component and the human machine interface resides on another physical node in the IHMI component. The IHMI component is the whole operator PECT that was described in the earlier section.

Because of cost and safety requirements, we decided to build a lab environment in hardware that serves as a test bed with somewhat the same behavior as a substation switch. Of course the current allowed was several magnitude less than in the real case and from that perspective this was an example test bench, but we wanted to see if it was possible to predict the time for the over current protection scenario and then measure on a "real" system. The switch, which was called SEI switch was developed to facilitate the prototype development of the controller PECT and also to provide a demonstration environment where it is possible to verify that the breaker is tripped by software respectively hardware. For schematics of the SEI switch hardware see the experience report [43].

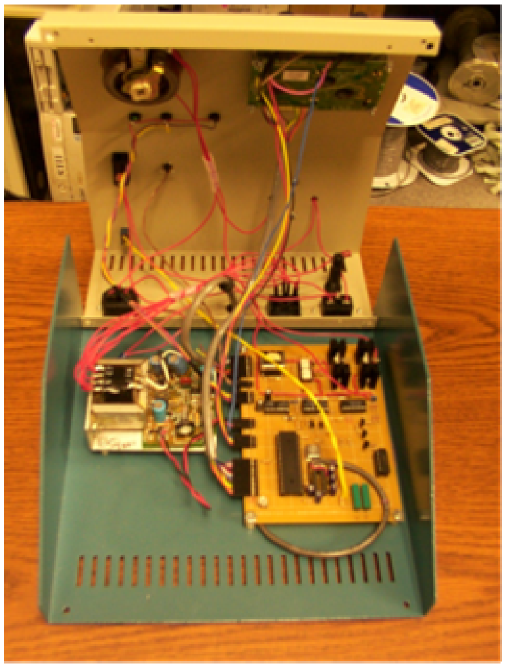
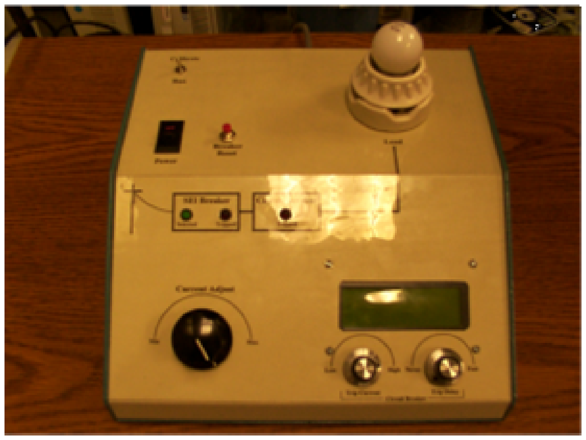



Figure 35: The SEI switch for lab purpose

The software logic is implemented in the controller PECT using the PIN component model. Each logical node component in the IEC controller logic is implemented by a PIN component in the PECT. The implementation of the assembly is shown in Figure 36.

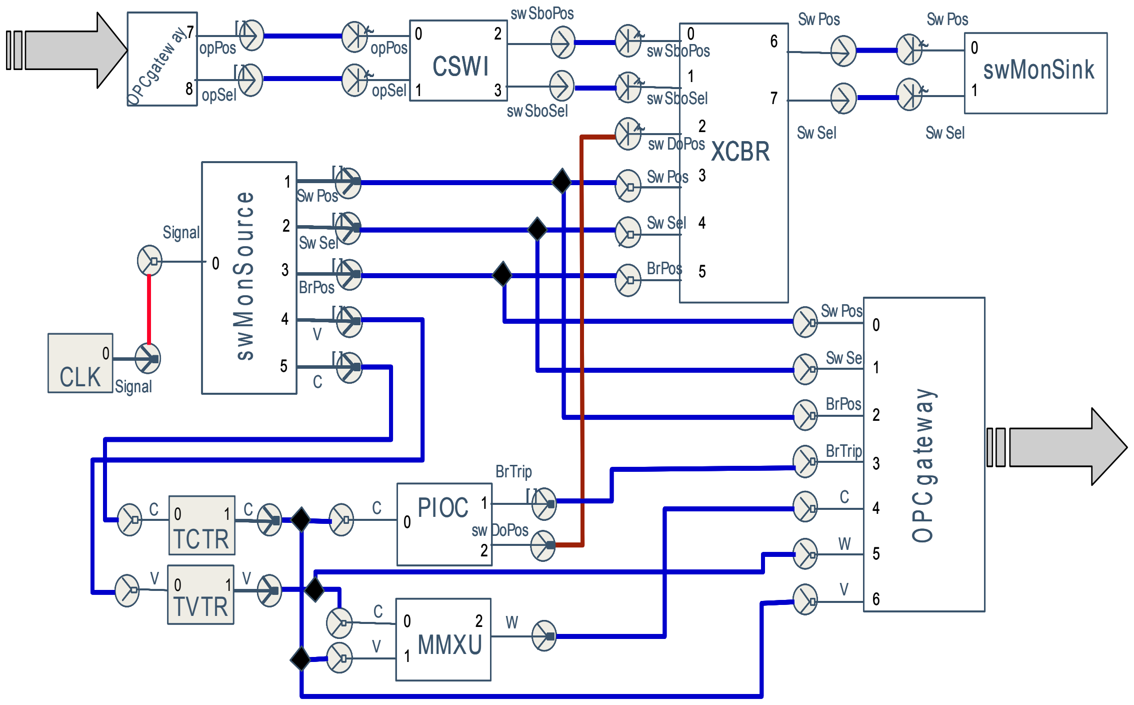


Figure 36: A component assembly that controls the SEI lab switch [44]

The component model is interpreted into an analytic model that can be used to analyze the desired latency. There are both real application assemblies like the one shown in Figure 36 and synthetic ones. The pure synthetic assemblies with no functionality are used for the validation of the controller PECT. Synthetic assemblies are generated depending on the validation scheme and the defined variation points. An example of an assembly in the constructive model is shown in Figure 37.

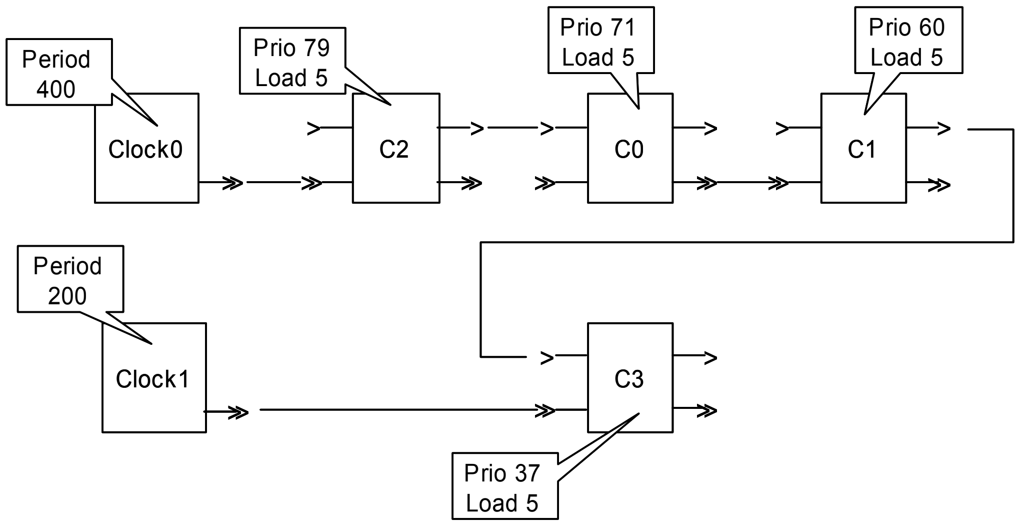


Figure 37: An example synthetic assembly generated from a variation point

The assembly shows contains four components and two clocks that have a period time of 400 ms and 200 ms, respectively. It is an assembly that fits the eighth assembly space tuple in Table 16, but there might be other valid assemblies that satisfy the variation points defined. As mentioned earlier several assemblies have been generated from the same variation point. Each component has a priority and load variable that have been annotated. A higher priority number implies that the component will execute before one with lower priority. The load factor is a constant that is feed into the component during execution and for the controller target environment a factor of five resembles roughly five milliseconds execution time.

5.4.1 Analytical Model

All components in the assembly including the clocks are interpreted into an analytic view where there is a notion of tasks instead of components. The transformation rules take the assembly from the constructive view to the analytic view. It is only in the analytic view where the property theories can be used.

As there are different ways to calculate latency, it is important to differ between worst versus average case latency. There are well-established theories for worst-case latency and the formula in (22) can be used to calculate that.

In this particular example fixed priority scheduling is used and worst case latency of component c_i, L(c_i), can be calculated as:

$$L^{n+1}(c_i) = c_i.wcet + B(c_i) + \sum_{\forall c_j \in hp(c_i)} \left\lceil \frac{L^n(c_i)}{c_j.T} \right\rceil c_j.wcet \tag{22}$$

,B is the blocking time, $hp(c_i)$, is the set of components having tasks with higher priority than component i, $c_j.T$ is the period and $c_j.wcet$ is the worst-case execution time of component c_i.

Apart from worst-case latency we are interested in average latency. The calculation method is to use simulation as a mean to calculate average latency. The simulation algorithm is developed by SEI and it is described in [44]. When the assembly has an analytic representation, it is possible to apply the property theory that will result in a prediction of the different execution times. The notion of components has been removed since a representation that fit the reasoning framework is needed.

Tasks are represented by insertion of clock components. Clock components fire events on their source pins with a given period. The components connected to a clock will execute according to the clock cycle and these components constitute the implementation of the task. A-periodic tasks are components connected to a source that provide data in a sporadic manner. For instance, a component might send data on its source pin only if a certain threshold is reached. The task waiting for that particular pin will then be invoked sporadically.

A *job* within a task is defined as a complete execution of a task. Depending on th e interference from other tasks in an assembly, a task can exhibit different latency. All tasks in an assembly execute in periods and a *hyper-period* is the least common multiplier (LCM) of the tasks periods. The job latency, also referred to as response time, is the amount of time it takes from the moment the task is ready to run to the moment it finishes executing. Average task latency is the average latency of each job in a task in a hyper-period.

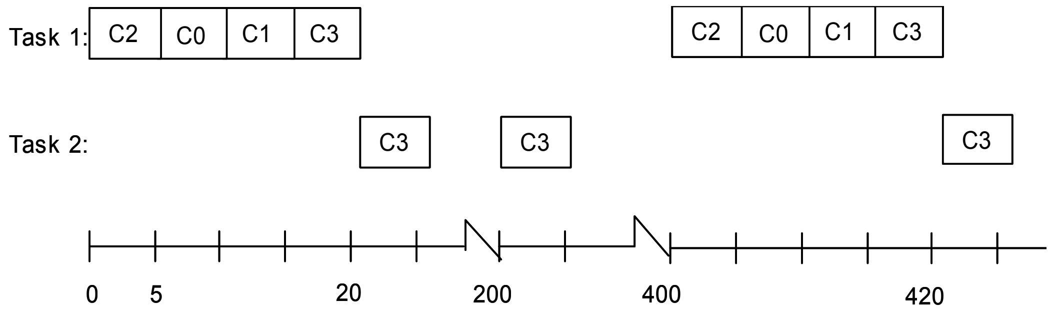


Figure 38: The analytic view of the generated assembly with two tasks and their execution

Figure 38 shows an example where task 1 releases four components, a job, for execution every 400 ms and task 2 that releases 1 component for execution every 200 ms. As the components in task 1 have higher priority than the one in task 2, which results in a blocking for task 2 in the first job.

Latency for task 2 and the first job is the latency for task 1 plus the execution time of component C3, i.e. roughly 25 milliseconds. For the second job in task 2 there is no blocking and the component can be executed without interruption with a latency of roughly 5 milliseconds. After 400 ms, the execution pattern restarts and this defines a hyper-period of 400 ms, to get the average task latency all jobs within a hyper-period have to be accounted for.

5.4.2 Validation Goal

In the SEI controller PECT example, the design problem was to develop a PECT that would predict latency between two pins in an assembly with an MRE ≤.5%. The empirical validation was required to be done with a confidence level γ =0.99. In order to have a pass/fail condition, we set a minimum acceptable proportion p = 80%. This specified goal is used later during evaluation of the actual sample results. Desired is also that the correlation of predicted and average measured latency is very high. The goal is show in Table 11.

Table 11: Validation goal for the controller PECT

	Value
Mean MRE	<5%
Confidence level	99 %
Spearman rank correlation of predicted latency and average measured latency	> .998 p-value < 0.05
Proportion	80%
Proportion criteria	≤5%

5.4.3 Validation Process

All components are measured and the different assemblies are run through an analytic tool that is able to predict the latency for each job within each task in the assembly. The analytic tool is based on simulation. One hyper-period is analyzed to make sure that all the different possible scheduling combinations are covered. The measurement infrastructure is described in more detail in 5.4.4.

The method for drawing conclusions is based on input from [90]. First, a sample set of several assemblies is created. Each constructive assembly contains components that are seen as tasks and jobs in the analytic view. That means that there are possibly several tasks and jobs within each assembly of components. For each task and job in the assembly, latency is predicted. All predictions are stored for later retrieval. Each job is measured 30 times making up the total latency for the task. As average task latency is of interest we have decided to define that as the average of all the job latencies. I.e. each job is measured 30 times to get the average job latency, and then an average of each job is taken to get the average task latency.

Each task measured generates one sample point. A *sample point* takes into consideration the magnitude of relative error between the predicted and average measured value. A regression analysis of all sample points is then performed to show any specific correlations that have to be considered. In addition, a histogram is created to illustrate the descriptive statistics for the all samples and their MRE.

The different characteristics in the controller PECT are captured by having a tuple of variation points $V = \langle v_0, v_1, \ldots, v_n \rangle$. Boundaries are set on each dimensional variation point. One boundary is that no assembly was considered for testing if it had more than 50 components.

Instead of designing a random generator that works justified over the hyperspace, the points are chosen and then an assembly is generated to match that point in the hyperspace. The variation points defined shown in Table 12 are defined. The actual samples chosen for each variation point is shown in Table 16.

Table 12: Defined variation points for the controller PECT

Variation Point	Name	Type	Range	Description
V1	Number of clocks	Integer	[1..4]	Total number of Clock components (tasks) that are used as a stimulus to other components within an assembly
V2	Number of comps.	Integer	[2..50]	Total number of components, in this case, synthetic components, that make up an assembly
V3	Number of conn-ections	Integer	[1..3]	Maximum number of connections allowed for a single component's source pin to be connected to other components' sink pins
V4	Min load factor	Integer	[5..20]	Minimum execution time for a component in the assembly. This number has to be less than or equal to the maximum load factor
V5	Max load factor	Integer	[5..30]	Maximum execution time for a component in the assembly
V6	Harm-onic Period	Bool	{Yes, No}	Describes whether the clocks in the system have harmonized periods. This variation point impose that the selected period will be harmonic with the rest of the tasks in the assembly
V7	Min clock period	Integer	[20..500]	Minimum clock period for the clocks in the assembly. This value must be less than or equal to the maximum clock period
V8	Max clock period	Integer	[100..2000]	Maximum clock period for the clocks in the assembly
V9	Com. type	Set	{A,S,M}	Determination of whether the connections between the components in the assembly are asynchronous ('A'), synchronous ('S'), or heterogeneous ('M,' a mix of synchronous and asynchronous) type communications
V10	Percent blocking	Integer	[0..100]	Percentage of the total number of synchronous pins used in the assembly that must be blocked or mutexed

In our example, there are certain variation points that are Boolean, such as, mixed connections or not, other variation points are expressed in the form of tuples. The tasks in the controller execute with different periods and this variation point specifies the range of periods that the tasks can have. For instance, in the controller PECT we have the range 20 ms to 100 ms for fast assemblies and 500ms to 1000 ms for slow assemblies. This means that all tasks within an assembly get their periods within the selected range. Domain knowledge is needed to be able to state the boundaries of the variation points.

If the periods are not harmonic the tasks will get a random value in the selected range, while if the periods are harmonic the assigned periodicities are a multiple of the lower bound in the range.

The size of an assembly is a three tuple of both the number of components, the number of connections and the number of clocks ⟨#Components, #Connections to source pin, #Clocks⟩. The number of clocks is equal to the number of tasks in the system in our example. Size is a variation point usually easily measured. It is good to have variation points that can be quantified, because we want an automatic way of generating the sample.

5.4.4 Measurement Apparatus

As the test and execution environment directly affect the execution time of components, it is important to record what software and hardware is running on the machine. Table 13 shows what software and hardware components are used in running the experiment.

Table 13: Software and hardware specification

Item	Value
System Manufacturer	Dell Computer Corporation
System Model	DIM4400
Processor	x86 Family 15 Model 1 Stepping 2 GenuineIntel ~1595 Mhz
Physical Memory	1,047,856 KB, 184-pin DIMM PC2100
Virtual Memory	3,570,352 KB
Hard Disk 1	WDC WD400BB-75CAA0, 40 GB, 7200 RPM 2 MB Buffer, 8.9 ms Average Read Seek Time
Operating System Name	Microsoft Windows 2000 Professional 5.0.2195 Service Pack 2 Build 2195
Compiler	Microsoft Visual C# .NET, 1.0 Build 3705
Compiler	Microsoft Visual Studio C++, 6.0 Service Pack 5
Real-Time Extensions	VenturCom RTX, 5.1.1 Build 3517

Figure 39 shows how an assembly generator can be used to generate the samples.

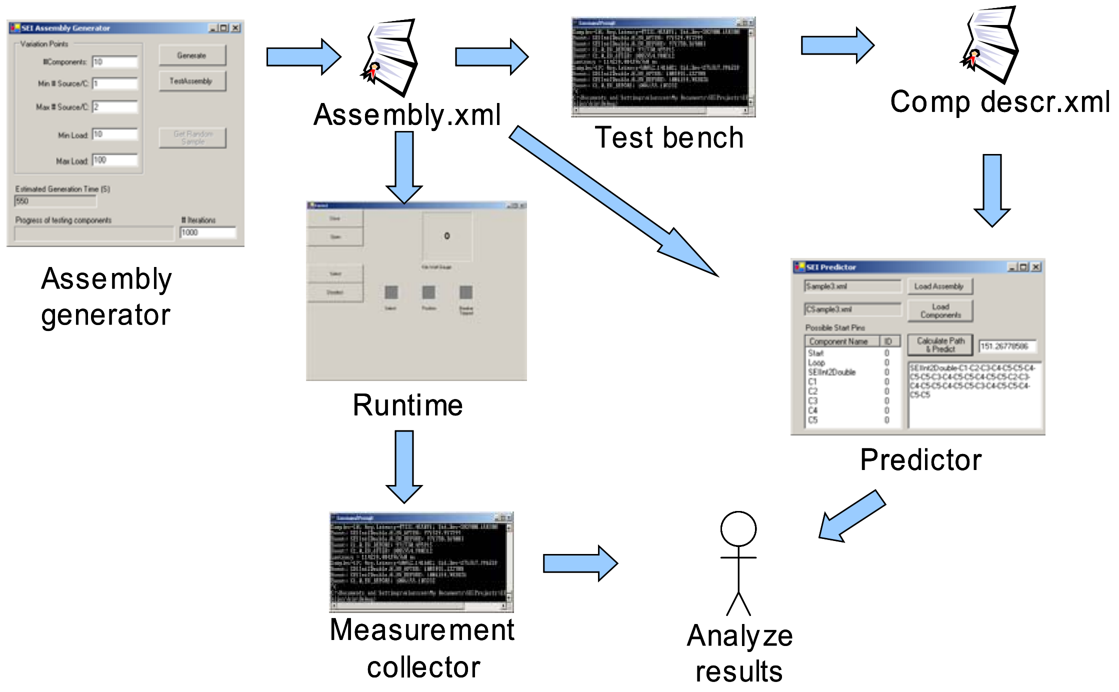


Figure 39: Overview of the analytic validation experiment

The generator takes as input the different defined characteristics and produces constructively valid assemblies using synthetic test components. In this experiment the synthetic components do not perform any realistic task other than consuming CPU time, or performing other dummy tasks. No side effects are allowed to avoid deadlocks etc.

The confidence in the future performance of a PECT is higher when the assemblies picked resemble future assemblies to be developed with it. However there is often no way to know what assemblies are going to be created with the PECT in the future and hence we have to perform the experiment with a guess of appropriate assemblies or at least with defined characteristics.

The assemblies generated are validated to be sure that they are schedulable and that predictions can be drawn from them before the assemblies are accepted to be part of the sample set. The assembly-generator randomly picks one assembly matching the specified characteristics and then it checks if it is valid. If the assembly is not valid, another assembly is picked and validated. This procedure continues until valid assemblies are found for all specified characteristics.

All components in all the assemblies in the sample set are measured individually in a test bench to gather the component properties. These properties are stored to be used later when the latency predictions are to be done. A separate predictor tool runs each assembly and produces predictions for each job in the sample set. In addition, the measurement points are calculated together with information about how to measure each job. The measurement collector uses this information to know what measurements to collect from the runtime system.

After the predictions are done and stored, we utilize a tool that starts all the assemblies with the runtime process and measures all the different jobs that were predicted. These measurements are then compared with the predicted ones to gather the result.

The tasks are scheduled, a hyper-period is calculated, and a simulation is run to obtain different predictions for different jobs and tasks. The code and explanation of the predictor can be found in [44]. The predictor takes as input the tasks it should analyze. The tasks have priorities, starting offset, subtasks and periods. Each subtask has priority and execution time. This execution and scheduling information is represented by properties of the components. These properties are accessible at runtime and when doing the analysis of the latency. After the predictor has run the simulation, a prediction for each job with in a task is presented as shown in Table 14.

Table 14: The predicted values for the example assembly

Assembly	Task	Job	Predicted Job Latency (ms)	Predicted AVG Task Latency (ms)
11	1	1	21.285	21.32
11	2	1	26.509	15.85
11	2	2	5.224	

All assemblies are deployed and measured in the target environment with the defined measurement infrastructure that runs the same assembly at least 30 times to get an average execution time per task and a standard deviation. An average of the measured task latency (Table 15), defined as the average of each job in a hyper-period, is used for comparison with the predicted one.

Table 15: Average measured latency is achieved for each task in the assembly

Assembly	Task	Job	Number of measurements	Average Measured Job Latency	Standard Deviation	Average Measured Task Latency
11	1	1	30	21.317	0.0122	21.317
11	2	1	30	26.442	0.0047	15.845
11	2	2	30	5.248	0.0045	

The same procedure is repeated for each assembly and the final results are then analyzed, a complete data set for both sample sets are to be found in Table 19 and Table 20.Each predicted average task latency is compared with the measured average task latency by calculating the magnitude of relative error (see equation 20).

5.4.5 Collection of Sample Data

Each characteristic description is defined through its variation points. Table 16 shows the characteristics of the samples in this experiment.

Table 16: Variation points for the sample assemblies

Assembly Design										
Space Tuple	**# of Clocks**	**# of Components**	**# of Connections**	**Minimum Load (ms)**	**Maximum Load (ms)**	**Harmonic Period**	**Minimum Period (ms)**	**Maximum Period (ms)**	**Connection Type**	**% Blocking**
1	1	2	1	5	5	Y	20	100	A	50
2	4	50	2	5	20	Y	500	1000	A	50
3	1	2	1	5	5	N	20	100	A	50
4	1	2	1	5	5	Y	20	100	M	50
5	4	50	2	5	20	Y	500	1000	M	50
6	1	2	1	5	5	N	20	100	M	50
7	2	4	1	5	10	N	100	500	M	50
8	2	4	1	5	10	Y	100	500	M	0
9	2	8	1	5	10	N	20	100	A	100
10	2	8	1	5	10	N	20	100	M	0
11	4	35	2	15	20	Y	500	2000	A	50
12	4	35	1	5	5	Y	500	2000	M	100
13	2	4	1	5	5	Y	200	400	M	25
14	3	10	1	5	5	N	200	400	M	25
15	4	15	1	5	5	Y	200	400	M	25
16	2	15	3	20	20	Y	400	1200	M	75

We have two different sample set of assemblies and different number of assemblies is generated for each set. In total 32, for the first sample set, and 75 for the second, valid assemblies were generated and each assembly contained different tasks and jobs. In total 65, measurement points were defined in the first sample set and 156 points for the second. The reason for the two sets is the need for a rerun of the experiment after correction of certain faults.

5.4.6 Analyze Results: part I

The data is analyzed in two parts, the first part applies certain statistical method to the first sample set and a fault is indicated. This part is included to illustrate the need and procedure for iteration and refinement of the constructive and analytical model when faults are discovered. Faults are indicated by the statistical results we obtain from data collection phase.

The first results of measuring the predicted 65 measurement points are displayed in Table 17. Measured values and MRE for each sample point is found in Table 19. The standard deviation correlates with the average measured latency for each sample. It is noteworthy that having synthetic components with well-defined behavior results in little surprises and deviation of the measured latency, but anyway we can find errors and faults in the PECT.

Table 17: Descriptive statistics of the first controller sample set

	Value
Samples (N)	65
Mean MRE	3.82%
Median MRE	0.43%
Standard Deviation (SD)	6.45%
Spearman rank correlation of predicted latency and average measured latency	0.998 p-value < 0.0001
Shapiro-Wilk Normality test for MRE	Coefficient = 0.66987 p-value < 0.05

The Shapiro-Wilk test shows a p value much less than the accepted 0.05 and that indicates that the distribution is not normal.

Figure 40 shows how well the predicted value is corresponding to the average measured value.

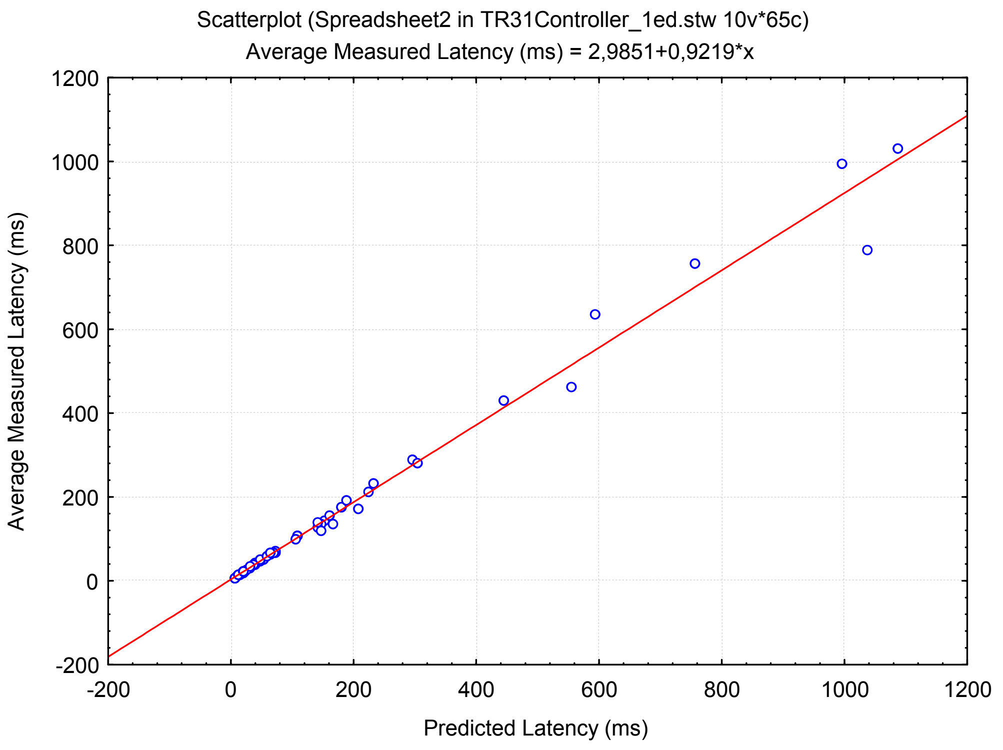


Figure 40: A plot of the predicted and average measured latency of the first sample set

For the sample, the correlation between the predicted value and the measured value is desired. A mean to analyze that relation is to use linear regression, which is a descriptive statistic technique that describes the strength of the correlation between two data sets and it is not directly useful for drawing inferences about future data sets. A consumer might be interested in linear correlation analysis as a descriptor of previous experimental validations of a property theory's accuracy. We characterize that accuracy using linear regression analysis, which allows us to assess the strength of the linear relation between two variables—in our case, predicted and measured average latency.

In our experiment correlation is high for smaller latency but when for larger latency the graph indicates less correlation as shown in the residual graph in Figure 41

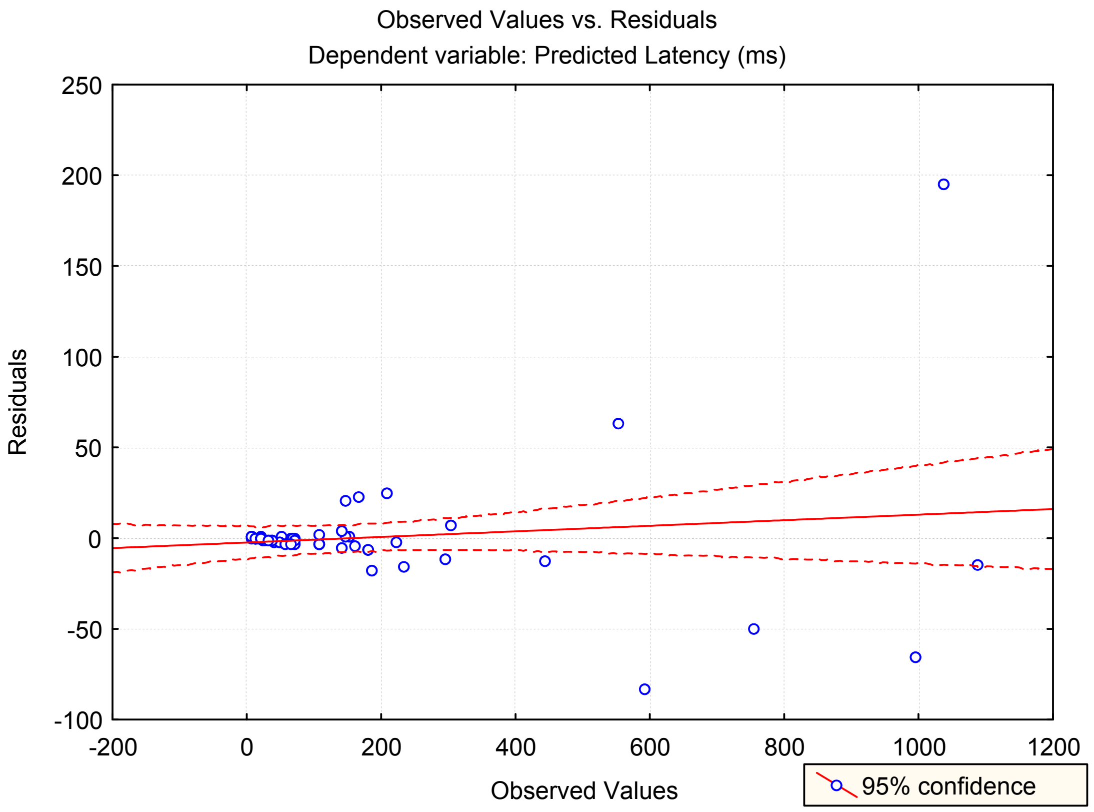


Figure 41: Residual analyze of the first sample[4]

The residual graph indicates that the size of the error, from the linear analysis, depends on the size of the predicted latency. Before this is investigated, we want to get an overview of the MRE.

To get an understanding of the distribution of the MRE we plot a normal probability plot, this plot is shown in Figure 42. In the plot we have the MRE as the observed value on the x-axis and the expected normal value on the y-axis. When the observed values (plotted on the x-axis) are normally distributed, then all values falls onto a straight line in the plot. When the values are not normally

[4] Regression analysis uses the terms predicted and observed values, and in our example the observed value is the dependent variable, predicted latency. Predicted value is the value calculated from the fitted linear equation. Residual is the difference between observed and predicted value.

distributed, they will deviate from the line. In our graph we deduce an interesting phenomenon around the zero point. When the MRE is below zero we have, except for two sample points, a normal distribution of the error. For positive MRE errors we get another shaped normal distribution; when prediction are made that are lower than the measured value we have a different distribution of the error compared with overestimated predictions. This implies we have a prediction theory, or model, that behaves differently depending on whether it over or underestimates the real value.

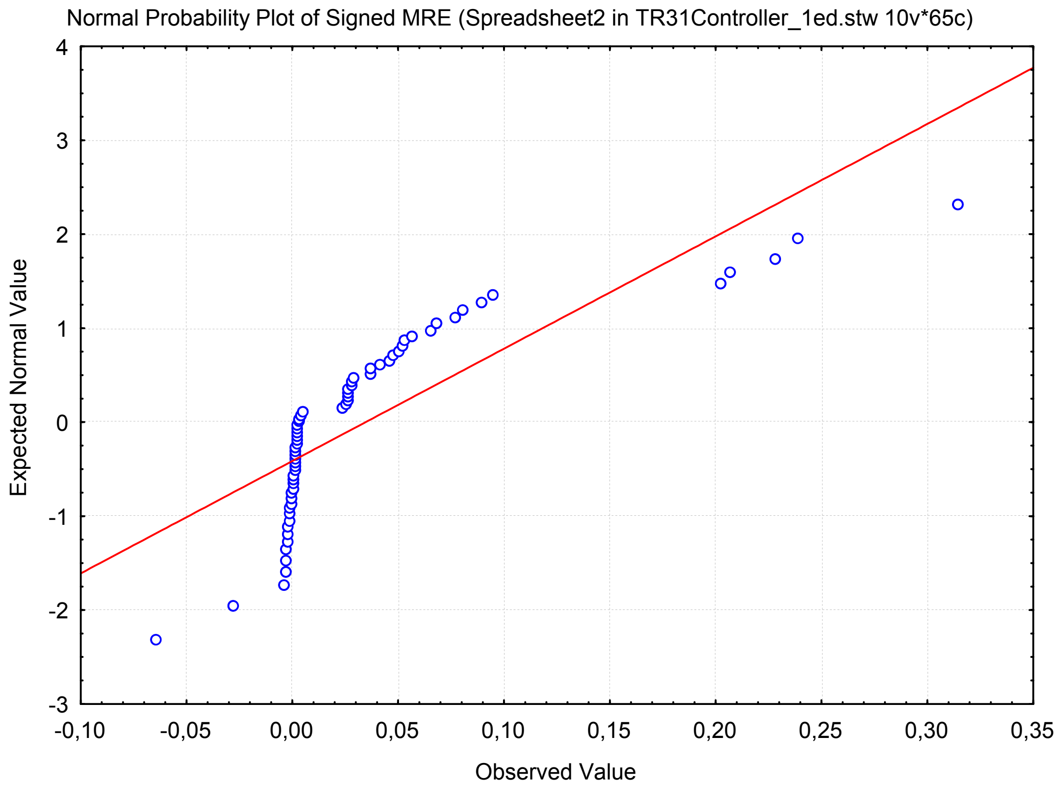


Figure 42: Normal probability plot that indicates that we have two MRE distributions in the first sample[5]

To get an overview of how big the errors are we plot a histogram (shown in Figure 43). This however indicates that the error is distributed both on the positive and the negative side, indicating over and under-predicting of the latency. As we are more interested in the absolute value of the MRE we create another diagram showing the average of the MRE (show in Figure 44).

[5] The observed value is the MRE of the predicted versus the measured average latency

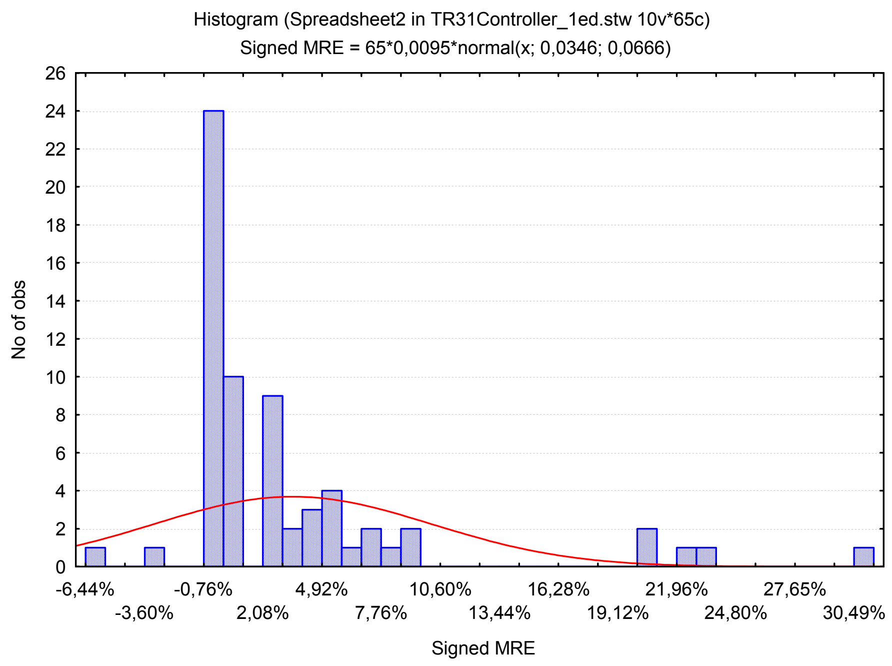
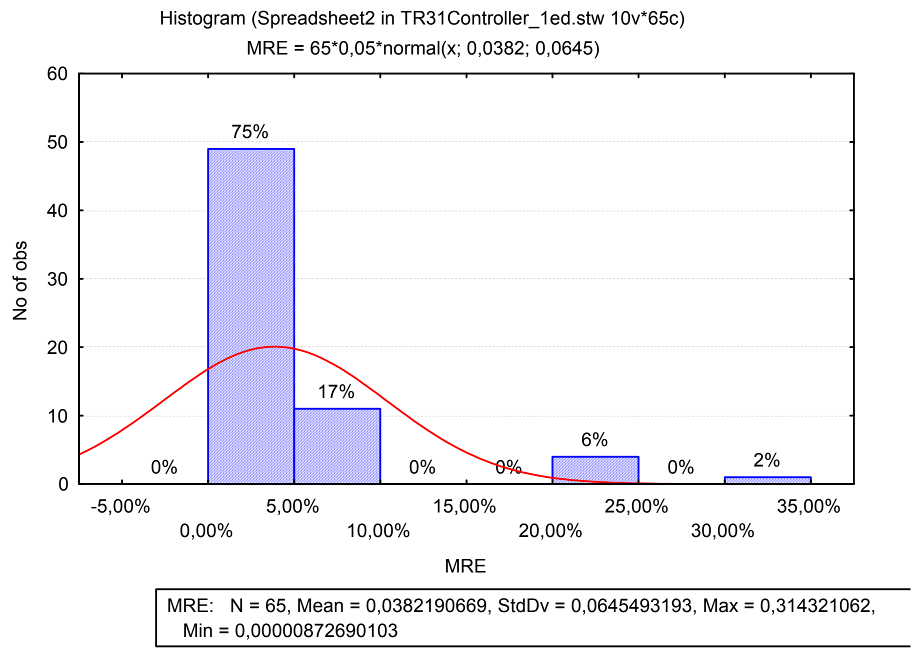


Figure 43: The histogram showing the magnitude of relative error



MRE: N = 65, Mean = 0,0382190669, StdDv = 0,0645493193, Max = 0,314321062, Min = 0,00000872690103

Figure 44: Histogram of the absolute MRE for the first sample set

A second look at the data shows that about 75% of all predicted values are well within the 5% MRE and that the rest is much above. That is however not good enough and the data indicates on possibly errors in the measurement framework, the constructive model, or the analytic model. A further analysis of the data is shown in [43]. After an investigation it was concluded that a fault in the constructive component model allowed inconsistencies between the analytic and the constructive model. The fault was corrected and a new round of empirical validation was executed.

5.4.7 Analyze Results: part II

An additional analyze was performed after error correction and with an extended set of assemblies. The results are discussed in this second part of the analyses. This time the results were much more encouraging. The results are shown in Table 18 and in [44].

Table 18: Descriptive statistics of the second controller sample set

	Value
Samples (N)	156
Mean MRE	0.51%
Standard Deviation (SD)	< 1%
Spearman rank correlation of predicted latency and average measured latency	0.9998 p-value < 0.001
Shapiro-Wilk Normality test of MRE	Coefficient = 0.56239 p-value < 0.05

As for the first sample set we want to look at the correlation of the predicted value and the measured value. Figure 45 shows that there is a higher correlation of the two variables in this sample but still it is not possible to know if the dependency on the size of the latency is still present.

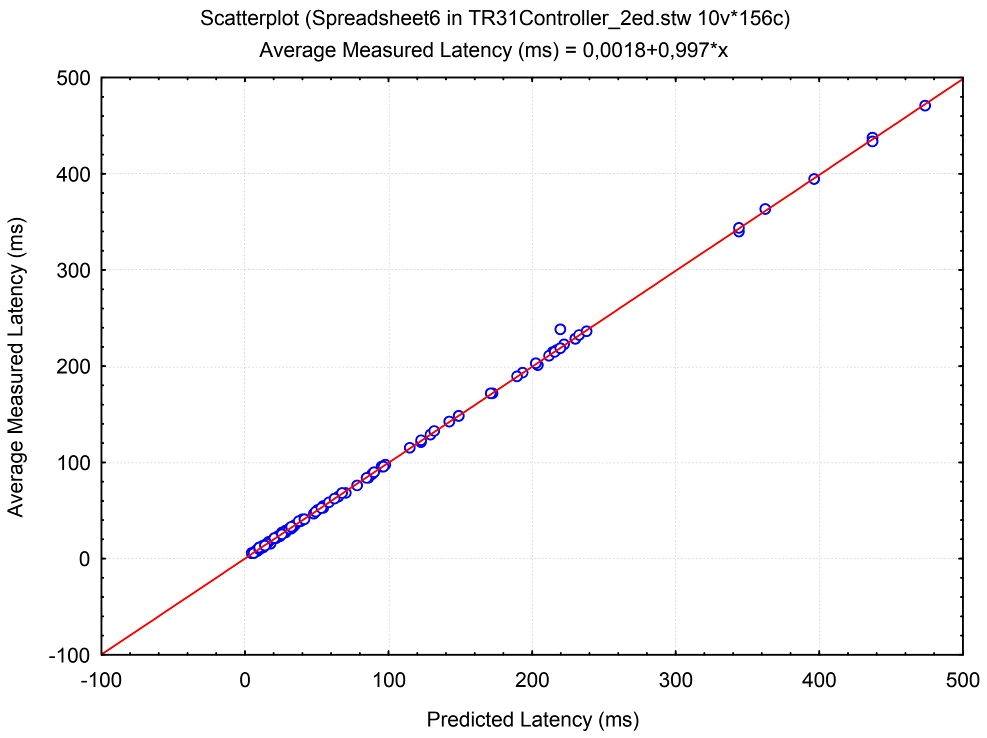


Figure 45: Predicted and average measured latency of the second sample

By looking at the residuals, it is possible to deduce this information. Figure 46 shows that there still seems to be a dependency on the size but it is not as big as in the first sample set where errors were present. It can be useful to describe the existence of a small dependency on the size, when we describe the property theory. For this example it is possible to say that predictions are accurate to a certain degree for latencies below 500 ms. This kind of information can guide the user of the PECT when creating new assemblies and performing predictions.

In this sample set we have one big outlier and probable reason for this one is a fault in the measurement apparatus. This outlier and its reason were not furthered investigated.

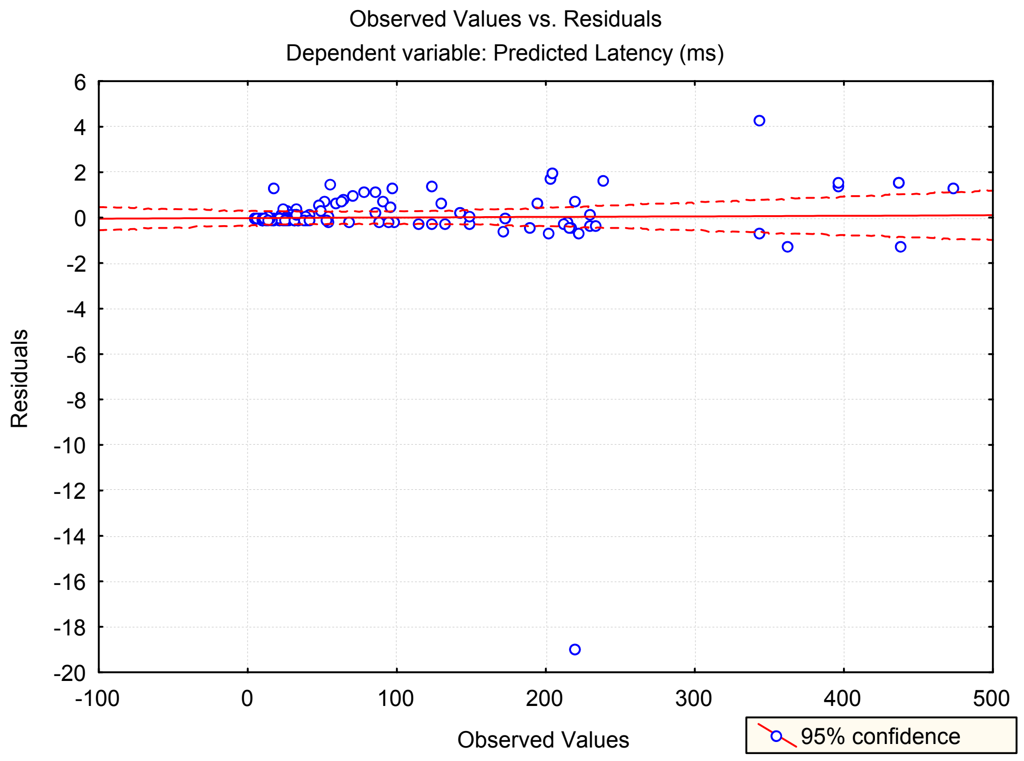


Figure 46: Residual analysis indicate one big outlier in the second sample set[6]

To view the relationships between the overestimation and the underestimation of latency we use the normal probability plot. This is shown in Figure 47 and the same phenomenon as in the first sample set is present for this sample set as well. There is a distinct break around zero indicating that the model is better when underestimating the latency, i.e. the variance is smaller for the underestimated values.

[6] Regression analysis uses the terms predicted and observed values, and in our example the observed value is the dependent variable, predicted latency. Predicted value is the value calculated from the fitted linear equation. Residual is the difference between observed and predicted value.

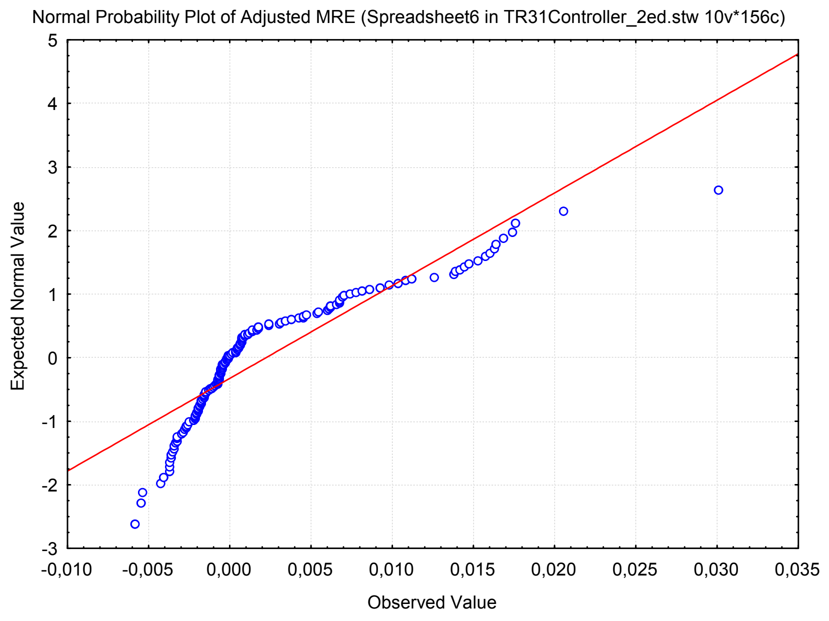


Figure 47: The normal probability plot shows the same characteristics as in the first sample set

When analyzing the MRE of the second sample set it is indicated that there are two MRE points 8% respectively –7% that will clutter a histogram, if these are assumed to be measurement error and removed, it is possible to see the distribution of the rest of the errors. Such a graph is shown in Figure 48.

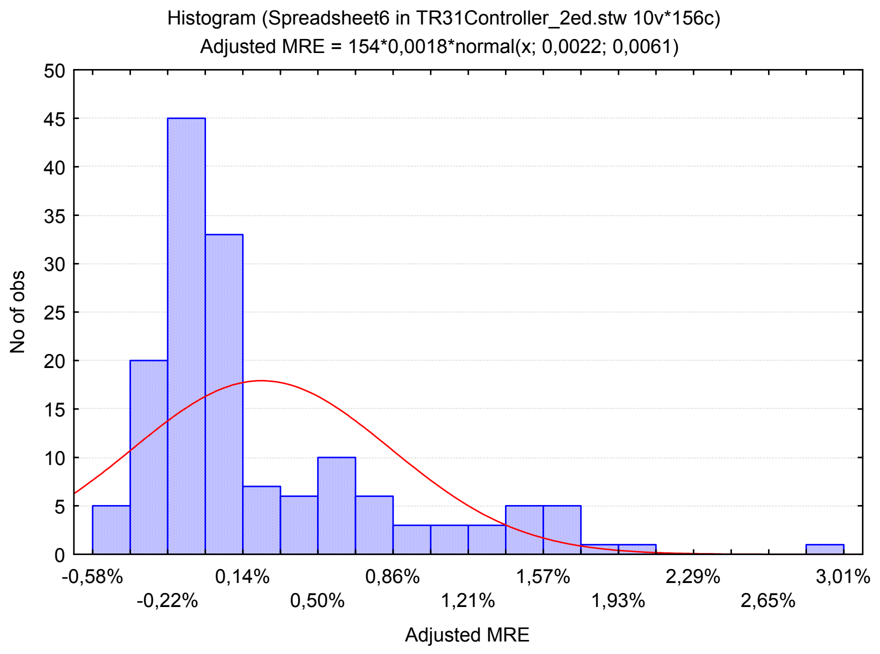


Figure 48: Histogram of MRE without the two outliers for the second sample set

A closer look at the points that are possible outliers, reveals that sample number 155 and task 3 consist of six different jobs with huge variation in latency (Table 20). Depending on where in the hyper period a job for task 3 is executed the latency is from 18 ms. to 445 ms due to interference of other tasks. It can be concluded that the large MRE of about -7% occurs for those jobs that have 445 ms latency in this particular sample. Sample 147 has similar characteristics as sample 155 but here the predictions are perfect when it comes to jobs with huge latency and hence it can be suspected that the outlier is from measurement error. For the 8% outlier a closer look shows that there are 2 jobs of which one has an MRE of roughly 15%, this single deviation results in the average MRE of 8%. We attribute the error to a fault in the measurement framework.

To demonstrate the absolute magnitude of relative error we plot the histogram as shown in Figure 49. It can be deduced that the average MRE is well below the targeted 5%, and the mean MRE is 0.51%, which is very good for the set goals.

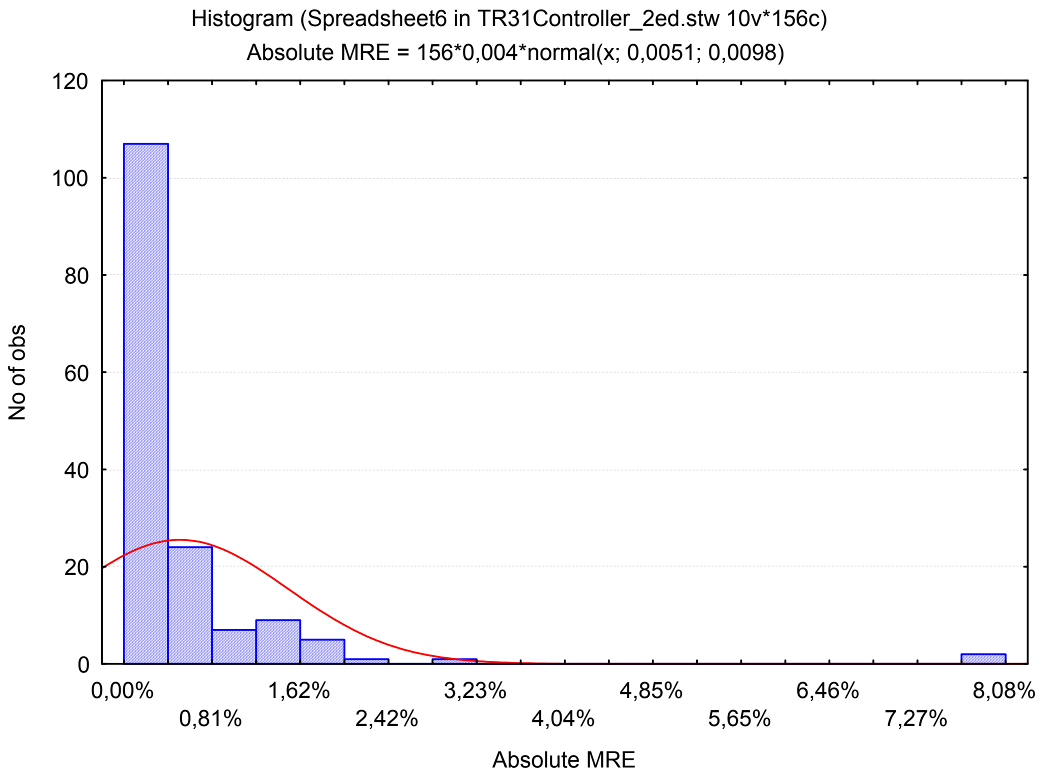


Figure 49: Histogram of the absolute MRE for the 2nd sample set

5.5 Summary and Conclusion

"If your experiment needs statistics, you ought to have done a better experiment" – Ernest Rutherford, Nobel Prize winner 1908

This quote pinpoints that statistics might not be necessary to conclude an experiment. However, it has been shown that the technique using statistics gives a good instrument for analyzing how good the prediction theory is. In addition, the method gives a way to find faults. When outliers or other non-expected occurrences of sampling results are found it can indicate faults in the implementation of the constructive component model and measurement apparatus. Hence, the statistical method can be used to indicate faults in the PECT as well as validating the accuracy of the predictions made.

Statistics can be used in a wider extent to reason about different correlations of attributes of a PECT. For instance can the design of the statistical experiment use the approach defined by Box et al [14] for multi factor analysis and picking the variation points.

In this chapter, we have shown implementations of two different PECTs and validation of those. The validations were conducted using the method defined in Chapter 4.

It is shown in this chapter that the operator PECT problem was carried out until a first PECT instance was created and validated, but not further. The controller PECT inhabited more challenging domain problem and value for the customer when it comes to latency predictions. For this and several reasons, the controller PECT was taken further.

The data shows that there is not one normal distribution of the MRE but rather that there are two normal distributions, one for underestimated predictions and the other for overestimated predictions. These two distributions indicate that underestimated and overestimated predictions should be analyzed separately or that the analytic model used for the predictions is behaving different depending on the two cases. It would be interesting to analyze the origin of this behavior but that is not part of the scope of this thesis.

A weak spot in the experiment validation is that there is no evidence that the chosen sample set is a good representative of the whole population. By using judgment sampling, we have to estimate the variation points that will be used for taking the sample and we have to trust that those variation points used to generate the assemblies are a real and good subset of the possible population.

The values produced for the average MRE and the correlation do however give indications on how good the property theory is; this approach gives value in reducing testing and providing methods for the developers to reason about their models, implementations and theories.

Errors present in constructive component model or the analytic model can be found in the validation phase using this validation technique. Finding the errors early results in that they do not have to be found in the test phase or even worse at the customer site. The earlier, in the development cycle of an application, faults are identified and removed the lesser the cost for the development.

6 A Dependency Example

The previous chapter illustrated a latency experiment by implementing two PECTs targeting different problems. Latency is a measurable property at the assembly level and this chapter looks at a non-measurable property, namely, dependency. By knowing dependencies between components, we aim to reason about the consistency quality attribute at the assembly level.

A system of components is usually configured once only during the build-time when known and tested versions of components are used. Later, when the system evolves with new versions of components, the system itself has no mechanism to detect if new components have been installed [67]. There might be a control that the version of the replacement component is at least the same as or newer than the original version, but check can usually be worked around or ignored. This approach might prevent the system from using old components, but it does not guarantee its functionality when new components are installed in a possible inconsistent manner.

This chapter addresses the issue of predicting consistency in component-based system.

Other work in the area of managing complexity in software systems is also addressed in [83], and these experiences from product data management and software configuration management can be used to discuss the dependency problem. This problem is discussed in this chapter.

The organization of the remainder of this chapter is that Section 6.1 gives an overview of the scope and goal of the chapter. Section 6.2 discusses the problem of consistency. Component models with and without expressed dependency relations are discussed in Sections 6.3 and 6.4. The chapter is concluded and summarized in Section 6.5.

6.1 Scope and Goal of Chapter

The scope of this chapter is to exemplify a quality attribute that is not measurable but rather calculated. We look first at the problem to achieve consistency between components in a software system or assembly. Two component models are used as an example, the first with expressed dependency relations and the second without.

For both example component models, we present possible approaches to discuss consistency or dependencies.

The goal of the chapter is to demonstrate a reasoning methodology for another, quite different, quality attribute.

6.2 The Problem of Consistency

Dynamic configurations of components are needed to permit architectures to evolve during the life-time of a system. This occurs in practice when components create instances of other components during the system execution. Architectural Description Languages (ADL) can be used to express such architectures [97]. A system with a configuration of components in which the components may be replaced with new components is designated a subject of change, i.e. its contents change during its lifetime. A configuration is dynamic if it allows that components can be updated after the deployment of a configuration. A static configuration has the same set of components over its lifetime and nothing changes after the system is launched.

One example of a dynamic configuration is a PC running with executables and shared libraries which might be upgraded during runtime, another is a control system in which the control algorithm component is detached from the memory and a new component is inserted. Configuration management typically addresses this type of problem with identification, versions and dependencies but has not been applied to the particular issue of dynamic configurations. These issues have been explored in [67,68].

Applications are also written today which upgrade themselves when needed, there are programs that checks with the home server for newer versions each time they are invoked. The management of systems in which components or applications upgrade themselves without notification is a challenge. Imagine components using another component which suddenly upgrades itself. How is deterministic behaviour possible in such a system?

Runtime reconfiguration is possible by altering interface bindings [82]. It is possible to change to a new component since the interfaces of a component arbitrate all communication to and from the component through proxies. If the components are state-less, the component can be removed and replaced with another which provides an equal or greater interface. In the case of a component with state, the state must be stored by the old component and reloaded to the new component. This requirement means that new versions of a component must be able to load and interpret the storage format from all older versions. The persistent data must be tagged with protocol version to permit the use of multi-versions of data.

To illustrate the problems we highlight two relevant cases and propose solutions to the consistency problem.

a. New features will eventually be added to any software product and those new features might be implemented by a set of new components as well as new versions of already existing components. Doing this, there is a potential risk that components could end up being incompatible with components already used in the product, both with respect to versions and variants. Figure 3 on page 20 illustrates such an inconsistency. This case is also related to maintenance of a product that may alter the characteristics or quality attributes of a particular component.

b. When an assembly of components is composed it is of importance that the right versions of components are deployed. One assembly might very well use certain versions of components that are in direct conflict with the components of another assembly. In many component models, multiple versions of the same component may not coexist. In those cases, there is a risk that components are assembled in an inconsistent way, by means of having the assembly include two or more different versions of the very same component. It is desired to prevent such invalid assemblies by being able to predict whether an assembly is consistent or not.

The analysis of relevant properties of assemblies in a software product perspective can be referred to as *impact analysis* [117]. Thus, it is desirable to analyze the impact of a change, e.g. installing new features in a product, maintaining existing components, or construct a completely new product based on reusable assets within the product line.

In many component models there is generally a lack of information available for identifying components in systems. For those cases, no information about version, change history or creation is available. There is no standard interface which can be used to gather sufficient information about the component to permit the creation of a dependency graph. Although there are component models that differ provided from required interface and those implicitly identify dependencies. A provided interface expresses what the component provides in term of functionality. The required interface specifies what the component need from its environment and other components. More information on component specification can be found in [71]. Dependency information between components is necessary to predict the effects of updating the system with new components.

The idea of having version dependencies is very similar to how .NET assemblies use meta-data to describe dependencies to other assemblies [106]. Dependencies can be expressed and assured using object constraint language (OCL [122]) constraints for the components. A new constraint has been added to all components that state the dependencies between components. This constraint can be used to evaluate and regarded to analyze the assembly.

In an approach where variants of components are of importance, as for instance in product lines [18], the handling of consistency is a 2-dimensional problem as shown in Figure 50. A component in a product line may be compatible with- or dependent of several different variants of other components. For instance, a GUI component for an embedded system could differ between products in a product line, e.g. high-end products with a color display and low-end products with monochrome displays. The color display and the monochrome displays are variants of the same feature, i.e. the feature of presenting information graphically to a user of the system. In turn, several versions can exist of every variant of a component. Typically, new versions emerge from error corrections and from new functionality being added.

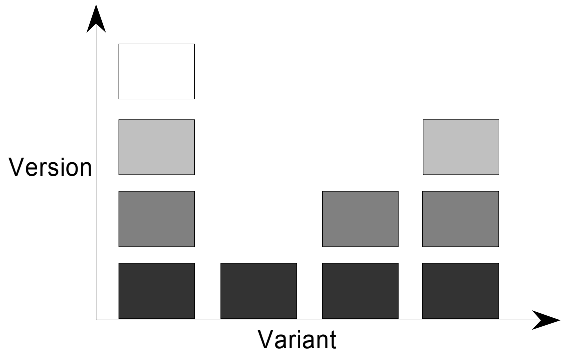


Figure 50: The 2-dimensional version-variant concept

We want to built a PECT that by prediction of the consistency solves the dependency problems. To do this we have to develop the property theory, identify the properties on the components and interpretation of constructive model for the analytic model.

A version of a component can be defined as a non-functional property of the component. In that case, the dependencies between components are expressed through such a property. From the discussion above, we can conclude that a component can depend on several different variants of a component but only one distinct version of each variant.

To be able to analyze the consistency of an assembly we introduce a property on the assembly. The consistency property, *A.consistent*, is related to a capability to predict consistency of an assembly. An assembly is considered consistent if the versions of each component are correct according to the dependency specification of a product. The specified features of a product determine what components versions should be included in a product. To be able to guarantee consistency we need to specify what versions of components a product depends on.

The quality attribute consistency is defined at the assembly level as a property. For the purpose of predicting variant- and version consistency on an assembly, a property that expresses the dependencies must be introduced on the component level. The analytical property of a component *c.depends* is used as a set of dependencies. In addition, a component must include specification of a version and a variant. The dependency set is containing information about all depending components to *c* and their variant and version. A tuple <C, variant, version> identifies a variant version of a component C. The component properties are then used to calculate if the assembly is consistent or not.

We calculate the consistency of all variants and versions in an assembly can be calculated with the following definition (equation 23). The property consistent is of type Boolean with possible values true and false and has the following definition:

An assembly A is variant- and version consistent, *A.consistent* iff:

$$A.consistent = \forall \langle\langle c_i, \text{variant}, x\rangle, \langle c_i, \text{variant}, y\rangle\rangle \in V \times V: x=y \qquad (23)$$

where $V = \bigcup_{c_i \in C(A)} c_i.depends$,

C(A)= the set of all components in an assembly *A*,

variant is a component variant and *x,y* are versions.

That is, the assembly is consistent if a component does not appear twice with different version in the union set of all dependencies. Even if the newer versions of components are compatible, we do not allow different versions of the same component to be present in one assembly. In the next section, we will look at an example where this theory is applied.

6.3 A Component Model with Expressed Dependencies

The component model used in this example is based on the port-based object approach, presented in Section 2.6.3, in which components are connected to each other by data ports that constitute a components *data interface* [102].

We illustrate the problem with an example of adding a new component C4 to a software system with. C2 is introduced dependent on the execution of C3 and the output from C3 and C2. Before a new component is added, we want to predict the impact it has to the system. For instance we want do an impact analysis by calculating *A.consistent* over {C0, C1, C2, C3, C4} (Figure 6 and Figure 51). The component C4 also expresses its version relation to other components. Component C4 depends on a particular version of C3. The dependencies are expressed using a precondition that asserts that the correct version of C3 is in C4's list depends.

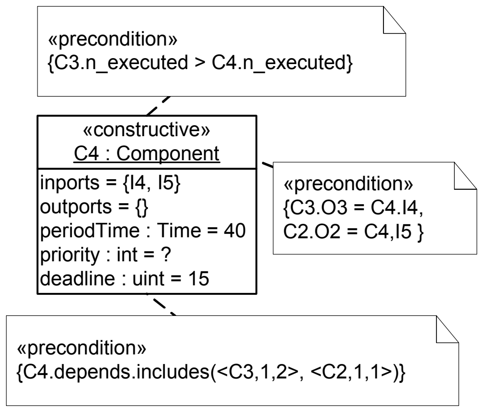


Figure 51: The new component c4 exemplifies a new feature

In a similar way it is possible to define, and apply, any other important property theory in order to analyze the impact of adding a new component to a system, for example a latency theory predicting execution time.

When component C4 is added to the assembly, it is possible to check the version consistence by using equation 23 above.

$$V = \bigcup_{c_i \in C(A)} c_i.depends \text{, } c_i \in C(A) = \{<C3,1,2>,<C2,1,1>,<C1,1,3>,<C1,1,3>,<C0,1,2>\}$$

As no component occurs in the set V more than once with different variant and version the assembly is consistent. If, however, C2 had depended on a C1 with version 4 there would have been an inconsistency since C3 demands version 3 of C1. This would have been seen in the set as two different tuples for C1 and that is not allowed.

By having a component model that allows or even forces components to express dependencies gives possibilities to reason about consistency problems and affect analysis.

6.4 Component Models without Expressed Dependencies

If a component model does not express the version data and dependencies, it is much harder to verify version consistency between the components in an assembly. For those cases, it might be possible to obtain dependency information by other means.

To demonstrate this, we have developed a tool, designated the dependency browser for the evaluation of the components in Windows 2000. Components in Windows 2000 are based on COM technology that does not provide sufficient version information [66]. The main requirement for the prototype was to be able to parse a Windows 2000 system for its components and their dependencies. An iterative development model was used to be able to show results more quickly. The dependency browser incorporates from the proposals we have developed in [67,68]. It is able to browse the dependencies in a system and to store them under version control. It is also used to gather information about changes made between two system configurations. Certain measurements, such as complexity analysis, are also provided.

There are different levels of dependency between components in a system; in a Windows system there are dependencies between shared libraries, as well as between static and dynamic COM components. Applications such as Word, Excel or Explorer, are treated as executables with their dependencies obtainable from the executable file itself. Since all Windows executable files comply with the portable executable format it is easy to track the shared libraries but not so easy in the case of COM components. Scanning all shared libraries and executables in a system creates a basic dependency graph. Various features of the tool then extend this graph. The windows registry has been used to gather information about each component, which is then added to the dependency graph. Gradually, a configuration graph is built up for use in configuration management. Processes can be supervised and when new components are dynamically loaded into the memory, the graph is extended with dynamic dependencies. However, the creation of a complete dependency graph at the Windows platform has been a tedious task, as there are too many dynamic dependencies difficult to detect because they have not been activated during periods of time when the system is supervised.

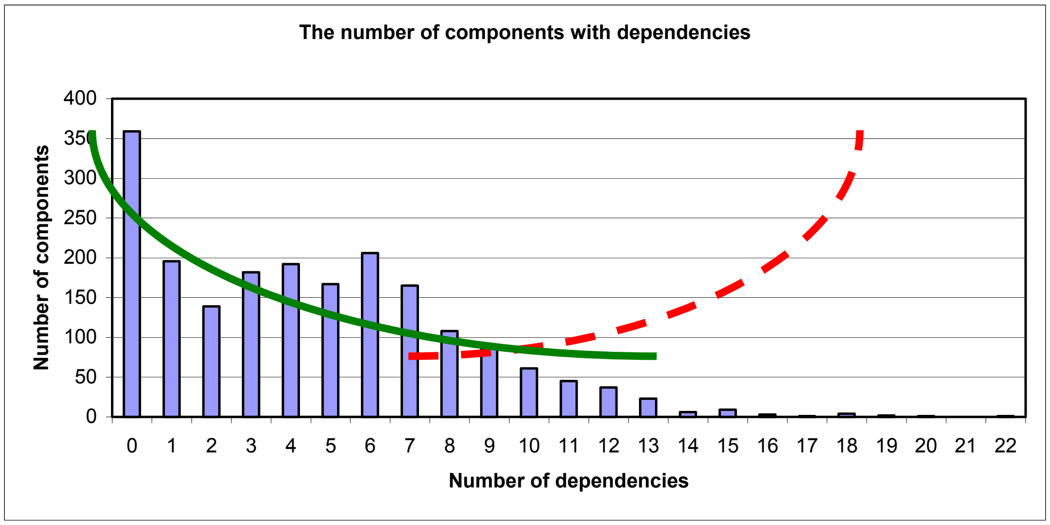


Figure 52: Analysis of number of dependencies from each component on Windows 2000. A system with a decreasing trend, shown with a solid line, is less complex than a system having an increasing trend, shown with a dashed line.

Experiments have shown 1993 components and 8936 edges in a Windows 2000 workstation configured for software development. The number of components and edges differ slightly between systems because of small differences in the installations as the test was carried out on difference computers.

Figure 52 shows the number of library dependencies for each component in the Windows 2000 system. In the graph, it is shown that most of the components have less than five dependencies, this figure shows that such a system is less complex than a graph where the majority of the components have more than five components. The general complexity can be derived from the graph since a low number of dependencies results in a less complex system. The number of components with zero dependencies can be treated as basic components with low complexity and the component that depends on 22 other components has a high complexity. In this example, we have measured only direct dependencies before the transitive closure has been calculated. If the trend is decreasing as show with the solid line in Figure 52, it is a measurement of a less complex and better structured system compared with a system having an increasing trend of dependencies between components.

A graph similar to that shown in Figure 52 can be created to express the reverse dependencies. Such a graph shows the components on which most other components depend. If there are few components on which many other components depend, these few components should be changed with discretion. On the other hand, all components without dependents can be exchanged without risking the function of the system. All the dependencies must be viewed from the dependants' perspective before performing this kind of analysis, i.e. a component must know who is depending on it. A measurement such as this describes how many dependencies there are to a particular component.

The results show that it is difficult to identify all the components and their dependencies on the Windows 2000 platform. The consistency theory can be applied when the dependencies are discovered but not when there are dynamic dependencies.

6.4.1 Introducing Versioning and Dependency Information

A possibility of managing dependencies is to adding an additional interface to each component that describes the dependencies. One component model on Windows 2000 is COM and this section discusses COM as an example. COM treats interfaces in a manner unlike other object models such as CORBA. COM components expose themselves and communicate through COM interfaces only. Moreover, COM is designed to work with loose references between components. There is no requirement that the clients shall know the class declaration since every class declaration contains implementation details. Components should be able to add or remove interfaces without affecting existing clients.

As components are loosely coupled there is no information connecting different versions of components with each other. A COM component finds its fellow components through the Windows registry in which all installed components store their activation data, such as Interface id, class id, library locations and where to find their stubs and proxies. Connections between components are set up first at runtime. A client uses a unique key to find the server component in the registry and then the COM runtime will load the corresponding component or stub into the client memory.

Unfortunately, there is no capability in the target system for finding which interfaces are used by a component. This prevents us from getting proper information about all dependencies in the system.

Hoek [110] presents ideas about how product-line architecture components can be identified. The properties he defines are name, revision, interface, connection, behavior, constraints, representation and origins. These are properties to be considered and placed under a version interface as proposed in [67].

If we do not know which components a program uses at runtime, we must request that knowledge. This can be obtained if the provider of the components implements a specific interface for version management, which we designate `IVersion` (Figure 53). The `IVersion` interface can return facts about version, name, creation date, compatibility change, interfaces provided and components used. If the components had such an interface, it would be possible to write a tool that could browse and record the dependencies between the components.

```
interface IVersion : IUnkown
{
  HRESULT Name([out , retval] BSTR *name);
  HRESULT Version([out , retval] VERSION *version);
  HRESULT CreationDate([out , retval] DATE *date);
  HRESULT TypeOfChange([out , retval] BSTR *name);
  HRESULT History([in] LONG size,
        [out, size_is(size)] HISTORY history[*]);
  HRESULT HasInterfaces([in] LONG numOfElements,
        [out, size_is(numOfElements)]IID interfaces[*]);
  HRESULT UsesInterfaces([in] LONG numOfElements,
        [out, size_is(numOfElements)] IID interfaces[*]);
}
```

Figure 53: IDL specification of IVersion.

The proposed version interface has several methods, which are described below:

- `Name`, `Version` and `CreationDate` identifies the component.
- `TypeOfChange` indicates the compatibility level affected by the change.
- `History` informs about previous versions of the component and which type of change applied between them.
- `HasInterfaces` shows all interfaces provided by the component.
- `UsesInterfaces` lists all interfaces used. This list makes possible the building of the dependency tree of the components.

In the absence of a standard version interface, another method is to parse in some way the dependency data from source code files to provide a list of dependencies with the release of a new product. This has some major disadvantages. Firstly, it cannot be applied to third party components. Secondly, it might work for the first level of dependencies where there is source code, but if other third party components are included, no information can be obtained because of the lack of source code.

A possible partial solution to the problem of finding dependencies between components is to track the interfaces from the registry repository as shown in the previous section. All interfaces are registered in the Windows registry with information about where to find the dynamic link library which implements the stubs and proxies for that particular interface. This mechanism provides us with the information we need to see if an interface has been changed during an update.

6.5 Summary and Conclusions

In this chapter, we have discussed system configuration consistency and illustrated how it is possible to build reasoning frameworks that support the consistency quality attribute.

In many component technologies, the interfaces have a lack of providing dependency information, the focus in existing component models are mostly on provided interfaces and not on required interfaces. If information about required interfaces is specified and can be obtained at runtime, it gives much more possibilities for impact analysis and consistency checks.

The consistent property is a typical example of emergent or derived property because it appears only when the application is assembled together, the components cannot be consistent by themselves.

For a dependency analysis, there is a need to identify a component its version and the components required. One approach to identify a component is to have name, creation time, size and a magic number (a unique number set by the compiler). If a version identifier is provided for a particular component, this could also be used. The identification data is used to calculate a unique key to be used to compare components. The key is divided into two parts, one for identification, and the other for version.

If no version information can be obtained from the component itself, it can be added using the *embedding* pattern described in [37] to wrap components with a version information interface.

7 Possible Implications of Design Decisions Based on Predictions

This thesis has presented how it is possible to achieve prediction capabilities in component-based systems. But even as it is shown that predictability is achievable it is not certain that the predictions can be used as is and for every type of application. This chapter outlines what to consider when using predicted data for design decisions.

There might be severe consequences deploying faulty or misunderstood software into a computer system controlling safety critical hardware. In many cases, software is deployed in such environment even if it is known that there are existing faults in the software. This chapter analyzes the ethical grounds used when taking decisions that might have severe consequences. In particular, we take predictions of quality attributes into consideration. Different quality attributes, such as performance, are hard to know even with thorough testing and that leads to new techniques being developed for predicting such attributes.

These techniques might not be fully proven and decisions based on using the predicted values must be based on a moral ethics ground. If the critical decisions are based on a moral egoism, that only gains the software company or developer, there is a risk that catastrophic consequences might follow. For instance, if a predicted execution time is out of the limits and a critical dead line are missed.

As everything is about making business it is very common that company decisions is based on a moral egoism and that might sometimes be justified sine the duty of the company is to make money. From the business perspective it might be the right moral to use, but there are other perspectives and consequences that have to be considered when taking a decision.

This chapter is started with a scope and goal description in Section 7.1. The chapter then follows with an introduction to the moral problems in Section 7.2. Section 7.3 outlines several software risks and 7.4 the problem of predicted data is described in Section 7.5. The chapter is concluded in Section 7.6.

7.1 Scope and Goal of Chapter

The scope of this chapter is to describe and highlight the risks of using predicted data when making design decisions. Included in the scope is a short introduction of moral problems and certain ethical aspects that ought to be considered when to use predicted quality attributes.

The goal of the chapter is to convey the importance of understanding the risks of software and that design decisions must be taken seriously. Additional goal is to point out possible consequences of such decisions.

7.2 Ethics in Computing

As software is becoming more complex, it is harder to test functions to assess the quality needed. It is not possible to test exhaustively large software systems since the number of potential states and execution paths grows tremendously. As shown in this thesis a current trend in research is to predict certain quality attributes of a software program reducing the actual attribute testing and then use these predictions when designing a system. Quality attributes such as scalability, performance, memory consumption and reliability, are often not considered part of the functional requirements of a software system and therefore often neglected in the design process. Customers of software mostly consider a particular functionality when they buy software and very often, certain quality attributes are not requested. These attributes are most likely taken for granted since of course the quality of a product matters to the end customer. This reasoning implies, however, that functionality requirements get higher priority over the non-functional quality requirements in the development process. In addition, having quality attributes on a lower priority means that most likely the attributes are not thought of until the system is implemented and tested.

The actual quality of a software product is discovered in the testing phase and then appropriate actions are taken to “fix” the right quality. Often only the quality attributes for which it is apparently shown that they do not meet the requirements are considered. Other attributes, not directly visible in the development process, but very important for the products’ lifecycle, are not considered at all. Such attributes may be related to reliability and safety issues.

There are several types of problems in management of quality attributes:

- The customer does not explicitly specify the quality attributes. They are assumed implicitly or not assumed at all.
- They are not properly specified for the delivered products.
- They are specified but not properly verified.

There are several reasons why the quality attributes are not properly treated. *Ignorance* can be one factor. Another factor can be *high pressure* to keep costs down and to meet time-to-market requirements. As the lack of quality might *not be visible* directly, it is easier to “forget” them and leave the problems for the future. There is a general problem of educating software developers and to make software engineering curriculums that adapts industrial problems and settings [25,123].

Quality attributes should not be allowed to be a discovered at later phases of software development; they should be explicitly addressed from the very start of the development. Nevertheless, this approach seldom takes place and software development must rely on testing to a huge extent.

The approach of predicting quality attributes already in the design phase has a goal to explicitly reason about the quality and to determine these attributes in advance. An implication is that predictions can very seldom be 100% accurate; usually there is a different level of confidence in the predictions. The question that arises is then: What if a software product is stated to have certain quality attributes and a customer makes decisions on that information to build a system where the environment, or even worse people can come to harm if the quality requirements are not fulfilled. The quality attributes define the non-functional behavior of a system to a large degree. Another case might be that the customer knows the software quality has been estimated but the customer still makes decisions to build a safety critical system.

If conscious design decisions are made even if harm to people might occur then it is questionable what is morally right to do. What are the moral standards when it comes to such decisions? Customers from different countries and cultures than the software developing company most likely have different opinion on what is right or wrong. By studying ethics, which is the theory about morality, a better understanding is attained of how certain behavior of software products can be accepted or rejected. The study of ethics is often called moral philosophy. To clear the difference between morality and ethics the following definitions are used [41]:

- *Morality*: first-order set of beliefs and practices about how to live a good life, i.e. the practice.
- *Ethics*: a second-order, conscious reflection on the adequacy of our moral beliefs and means to reason about morality, i.e. the theory.

Figure 54 shows ethical areas that apply to computing. The areas commerce, computer abuse, privacy, speech issues, social-justice issues, intellectual properties, basics and risks are not exclusive but could be used as categories for further discussion.

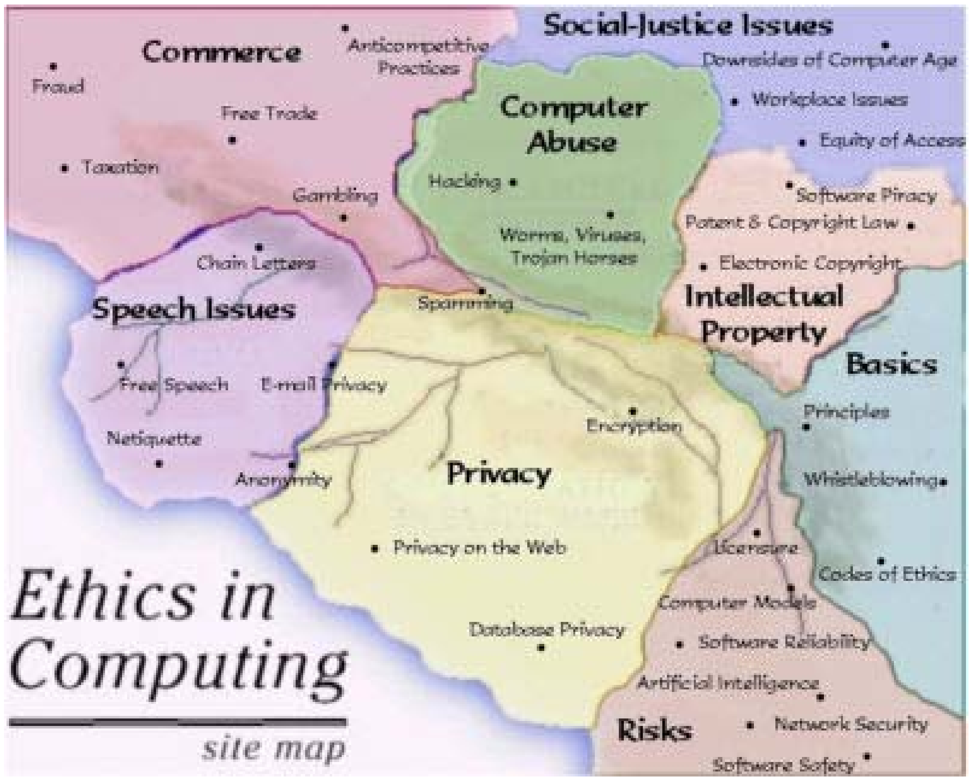


Figure 54: A brief overview of ethics in computing [80]

The problem of quality attributes is mainly found in the risks of computing but there are also commercial aspects that have to be taken into account. For instance, the software vendor might certify that the quality attributes of delivered software are met. This certification can be absolute or with certain confidence and it is of interest to the end user of the software to know the confidence in the software components that will be used. Further, a customer must estimate which level of confidence is satisfactory for a particular use of the system.

7.3 Software Risks

A driving force of software developers is that the software produced is going to be used to solve people's problems and fulfill needs [15]. Software can be used to solve almost an indefinite number of different problems and this implies that software products get deployed everywhere, even in an environment where a failure could cause terrible effects, e.g. fly-by-wire system. Having the airplanes equipped with fly-by-wire system allows reduction in mechanical parts and a more advanced flight. The term by-wire generally denotes removing existing mechanical controller functions and replacing those functions with software-controlled sensors that react on inputs from the surrounding environment including the driver. Most commercial airlines today use fly-by-wire and the trend is to introduce drive-by-wire systems in automobiles. The steering rod and wheel, although a construction well tried out and working, are to be replaced with software controlling the wheels over a computer network. Having such a system would allow new functions for the driver. The computer software could improve the ability of the driver by; for instance, react faster than a human on icy roads reacts.

An underlying problem with moving the control from the human to the computer is the reliability of the software controlling the wheels. The quality attributes of software become critical in a sense that software behavior must be known before the situation occurs. Testing cannot cover all possible events that a car driver can experience. Hence, the software or system designer has to make decision based on estimated quality attributes. On the other hand, in most of the cases, software methodologies cannot prove and guarantee a correct behavior of software in all possible cases. What is the morality of replacing a well-established technology such as the steering rod and wheel with a new technology knowing that it will cost lives or injury to human beings? On the other hand, there is a moral dilemma if, with the same introduction of technology, it is possible to prevent harm or save human beings.

A decision on when to use new technology, or predicted values as a base for design questions, which might have severe consequences, is a kind of moral decision. There are several bases for making a moral decision. Moral decisions can be based on different ethical theories [41,72]. Some of them are:

- Divine command theories, i.e. do what a sacred text tells, or the will of some undisputed power, e.g. will of God.
- Utilitarianism or Consequentialism, i.e. the best actions to take is the one that procures the greatest happiness or good for the greatest number.
- Virtue ethics, i.e. take action that maximizes virtue and minimizes vices.
- The ethics of duty or deontological[7] ethics. Base the decision on the duty of the decision maker. Do your duty.
- Ethical egoism, the only person to look out for is yourself. This is direct contrary to Utilitarianism. Ethical egoism maximizes on the person instead of the greatest number.
- The ethics of natural and human rights. Decisions take into account that all people are created with certain unalienable rights.

[7] 'Deon' = Latin for duty

When utilitarianism considers a state, for instance happiness, richness or quality of life, only the greatest number on the positive side is counted. Virtue ethics on the other hand consider the whole group which gives some relation between utilitarianism and virtue ethics. Let us take happiness as one example. Utilitarianism maximizes on the number of happy people but allows for some very unhappy ones as well. A situation might occur when the decision makes many people extremely happy and some very sad.

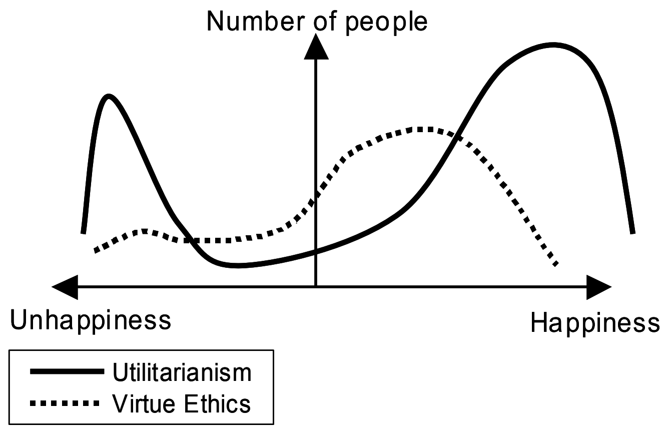


Figure 55: One example of a relation between utilitarianism and virtue ethics

Virtue ethics considers both the bad with the good and by making the decision so that fewer people are extremely happy there might be room to have fewer people unhappy. One can illustrate this with an example (shown in Figure 55); on the behalf of some very unhappy people we can get more very happy people vs. trying to have fewer very happy people allowing having less miserable ones. This example points out the differences between utilitarianism and virtue ethics. For other virtues, such as empathy or self-control this relation might not apply.

Very often decision makers do not think on what theory bases a decision is made. The person that takes a decision is living in a culture with more or less set moral standards. Hence, the decisions easily fall into that ethical norm in which cultures moral standards are based. Knowing the basis of decision-making and understanding the moral standards of, for instance, the customer, could decrease the number of bad decisions.

The risk of software must also be understood and risk assessments should be a natural part of software development. Today, often risk assessment in software development is more about checking the risks of not delivering the expected results on time. Software risks should not be mixed with software process risks. For example, a *software risk* is when software replaces hardware safety equipment such as an emergency brake. A *software process risk* is when technology used in a project becomes obsolete during development.

Apart from these underlying theories of what decisions are based on, there are ethics norms or codes of ethics defined in the software field. There are software engineering code of ethics and professional practice from IEEE-CS and ACM [52] and the ten commandments of computer ethics [6]. These codes of ethics are more policy like and state general principles about how the software developer should behave. For instance take the first commandment from Computer Professionals for Social Responsibility (CPSR) [6]: Thou shall not use a computer to harm other people, or the 3rd principle of the IEEE/ACM code of ethics, software engineers shall ensure that their products and related modification meet the highest professional standards possible.

Neither of the two codes of ethics addresses how to act and take decisions when it comes to using software that might risk damage to people or property. We should look more into the prediction of quality attributes and what that means from an ethical standpoint in the next section. It is important to know that it is common in the situation of taking decisions that a commercial side of the decision always forces decision makers to make compromises. But to make compromises is also to make decisions.

7.4 Predictability Based Decisions

Quality attributes of software such as reliability, scalability, performance and security get, as stated earlier, their values after the design is done. Even at the test phase, it is not certain that all attributes are tested fully and documented. By having a well-defined software architecture considering the quality attributes from the beginning, a better quality can be achieved. The software architecture gives certain control over how quality is to be achieved. By having the architecture set rules and limitations on the actual software produced in the architecture, it is possible to design the software so that certain quality attributes can be predicted in the design phase. For instance in an automation controller system, where there are several controllers connected to a super-visionary level of computers, it would be good for the software designer to know the time it takes to perform a specific task without implementing and testing the task. One control system example is the control of a steel producing rolling mill. A controller might have several tasks but one might be to fill melted steel into a cast of some sort. This task might be time critical and if not fulfilled melted steel might splash into the open harming human or machine. In this case, it is of great importance that the controller designer knows that the task can be fulfilled within the specified period. Time or latency to execute a task becomes a quality attribute that the software has to fulfill.

A control program is often designed in an environment using the IEC 6 1131-3 standard for control program languages [49]. This standard has the notion of function blocks and these are software components. These software components are part of a software architecture and as outlined in [44,91] it is possible to make predictions about assemblies of components if the components fulfill the design of the software architecture.

The main problem is not to make predictions about the actual latency of a task but to prove that the predictions are correct. Moreover, what is the confidence in the predictions and how can a designer make decisions to use that software if the probability of not meeting the deadline is known. That decision becomes a moral decision, which is based on ethical principles. Consider the control example, where a task must be accomplished within, let us say, 100ms. The prediction is that the task takes 95±5 ms with a confidence of 99% in the prediction. This means that in 1 out of 100 cases, the prediction is wrong, but does that mean that the deadline of 100ms will be missed? Most probably not since the latency might actually be shorter. In addition, does a missed deadline of 100ms always mean that the hot steel will splash into the open? Most likely not since there might be other tasks or precautions taken. There are other means of knowing the execution time of a task and one example is to calculate the worst-case execution time (WCET) [70]. Calculating WCET, however, might be a more expensive approach and not feasible in all cases. The empirical prediction might fill the holes where WCET calculations are not feasible but the problem of knowing the confidence in the result is introduced.

Having a theory of how to predict quality attributes, such as latency, in a component based software system is certainly advantageous, but the dilemma is to know how good the theory is. The verification of any theory can be formal or empirical where the formal verification actually proves the correctness and the empirical gives a degree of confidence in the theory. When a designer of a system only has certain confidence in the predictions about quality attributes then moral decisions must be made when and how to use the software. However, there are other means to achieve safety even if the controller is not 100% proven. Decision to introduce safety equipment should be taken if vital parts cannot be proven 100%, or even if a proof indicates 100% fail-safe since the confidence in the proof itself and non-occurrence of unexpected events might be lower than 100%.

Having a utilitarianistic base for a decision whether to use a certain control software or not can certainly lead to problems. The theory says that such decision is made to procure the greatest happiness for the largest number of customers. The controller might deliver excellent functionality and contribute in producing the very best quality of steel, but there is a known risk that the worst might happen, i.e. that hot steel will kill someone. The decision can simply economically calculate the impact of being sued for someone's death compared to all the other happy customers buying controllers.

Basing the decision on virtue ethics would mean that maximum happiness or virtue is gained but also that every action is taken to minimize the vices or in this case the risk of a fatal event. Unfortunately, decision makers sometimes base their decisions on egoistic ethics and that maximizes on the happiness of the actual decision maker and no one else.

7.5 Consequences

What are the consequences of a design decision? In many cases, the software engineer might neglect the consequences and instead put too much focus on the technical challenges. Brooks stated already in the seventies that programming is fun and that the quality part of the work is not considered fun [15]. Every programmer wants to feel the sheer joy of making things. This is inhabited in every human being that we want to create things, even better if the things we create come to use. Technical challenges are part of the fun of creating programs and consequences of what happens if the program malfunctions is not. If too much focus is on the functionality of the software, there is a huge risk that quality and consequences are neglected. Sometimes this is neglected consciously but maybe more often it is neglected not knowingly. How can moral decisions be made on the design of software if awareness of such decision is non-existent? Maybe the conscience will help the software developer making design decisions.

To have a sense of right and wrong or conscience is a base for taking decisions. But as pointed out by Twain and Freud [36,41], conscience can be wrong. In addition, conscience is almost exclusively about negative answers. An example could be a software developer asking herself if she should leave out the error detection code and focus on the functionality. The conscience should tell the software developer that, the error detection code has to go in otherwise it is likely that it will never be part of the design. Nevertheless, since conscience can be wrong it is probably better to have a conscious decision about the design principles. Support for conscious decision-making can be achieved by, for instance, code reviews, guidelines, understanding of the customer requirements and target application.

The decision making process in software engineering is often complicated by the fact that multiple responsibilities are assigned. A software engineer is often responsible for requirements analysis, research, design, implementation, testing, bug fixing, report and documentation writing and even project management. All these roles require decisions to be taken, decisions that might have severe impact on the health of another human being. This is not always clear and very often the designer have no time to start philosophical discussions about what are the actual consequences of the decisions made.

The responsibility and ethical choices are very much related to an individual. If an individual is involved only in a part of a process, he/she is not necessarily aware of the entire process and the possible consequences of a particular decision. It is also easier to decide not to be aware of that. In a component-based approach, components are developed separately from systems and often the component developers are not aware about possible use of their components and possible consequences of malfunction of the components. While a non-proper behavior or a lower quality behavior of a component in a test environment is not dangerous, the consequences of the same malfunctions can be disastrous in a safety-critical system using these components.

7.6 Summary and Conclusion

In this chapter we have discussed the ethical aspects of taking design decisions based on predicted data. It is pointed out that wrong decisions can have severe impact on environment or human life.

In the predictable assembly approach the aim is to predict the component and to some extent the quality attributes of a system with certain accuracy. The positive aspect of this approach is to provide some explicit specifications and to describe them in terms of statistics indicating by this that the specifications are not true. The risk of this approach is that the specifications are taken for granted and that the "high confidence numbers" mean a guarantee that there is a high probability that the system will work correctly.

Individuals cannot be trusted to take responsibility for these issues, instead they have to be handled by a professional organization, making conscious and professional decisions based on policies known internally, as well as externally. Professionalism in these, and other, aspects provides a competitive advantage.

8 Conclusion and Future Work

The predictability of quality attributes represented as the properties of components and assemblies would provide the possibility of reasoning about software quality attributes well before the testing phase in the development of a software application. Predictability would also provide the means of addressing the evolution of the architecture of a system or the results of changes in execution environment. Before a change is made, it would be possible to predict certain quality attributes to see if they are compatible with the results of the change. If components are reused or exchanged a precise specification of component properties would support management of the components.

8.1 Conclusion

This thesis addresses the problems of achieving predictability of quality attributes in component-based software systems. The general question on how to address quality attributes in software systems is approached by considering the underlying hypothesis that a component-based approach will support and facilitate the reasoning about quality attributes. Taking this approach, we have posed the following main research question:

Can the quality attributes of an application or a system be predicted using a component-based approach, given the attributes of the components?

We have discussed this question in Chapters 3, 4, 5 and 6, beginning by asking which assumptions must be considered when developing a technology that will support predictability. The assumptions are related to the quality attributes themselves (discussed in Chapter 3), the component technology that includes both a constructive component model and analytical models describing the composition of the properties, and finally, usage of the component-based applications. We must determine if there are quality attributes that are composable and predictable. To answer the question, we have built a general method for developing a PECT and for validating this. The method has been demonstrated using several examples of quality attributes.

It is shown in Chapter 5, with the latency PECT experiments performed, that it is possible to predict a quality attribute, latency, in a component-based software system. It has been shown in two experiments that it is possible to predict the latency quality attribute. Other types of attributes have been studied in the work of the SEI. Their work has shown that model checking can be applied, according to the same principles of prediction-enabled component technology, to predict safety attributes [57,119,120].

We have considered the research questions in relation to two different quality attributes using different component-based systems and component models. The thesis does not emphasize the component models or the quality attributes themselves (this was done as part of the research project but is not at the focus in this thesis), but emphasizes the process of validation and adjustment between component models and property theories.

The reasoning, the analysis of the experiments and relations to industrial settings have shown that certain quality attributes of systems can be predicted from the properties of their components. It has also been shown that the introduction of this method is not simple, but that it enables different levels of accuracy to be achieved according to the specification and restrictions of component models.

In order to answer the main research question we have considered several sub-questions:

Is it possible to develop a component technology that will support reasoning about system quality attributes from component properties?

In chapter 5 we have demonstrated the possibility of developing such component technology by developing a component technology supporting reasoning about latency attributes at an assembly level. To reach that goal certain additional rules must be applied to the component model. Since many component models today are not designed to include prediction capabilities, it is possible that prediction will not only require the introduction of additional specifications and rules but may also require restrictions on the component models or restrictions in the use of the component models.

This subtle difference in restriction has a relatively big impact. If we restrict the component model, we thereby actively prevent users from building components that violate the rules. On the other hand, if we restrict the use of the component technology, there is still the possibility of misuse of the underlying component technology. This is equivalent to informing a developer that he or she may only use certain functions of the underlying operating system.

In such a case, we must rely on the self-discipline of the developer. If on the other hand, the underlying technology prevents misuse, the self-discipline of the programmer is not a primary concern. Software developers must, of course, have a certain amount of self-discipline when they work in such an open and free world as software development.

A more serious problem could be to define appropriate restrictions to achieve predictability of different quality attributes – the problem is similar to the problem of trade-off analysis of software architecture to achieve particular quality attributes [61].

How restrictions on the analytic model are imposed is of importance since these restrictions have a considerable effect on how we can achieve prediction capabilities in our systems. The prediction capabilities of analytic models will also restrict the component model and hence the component model must be designed with prediction in mind.

There are different component technologies which conform differently to different analytic models. Further research is necessary to determine how the analytical capabilities that support predictability can be improved, A) by improving component models, B) by introducing different restrictions on the existing component models C) by means of a combination of these.

A lesson learned from addressing this research question is that even if it is possible to develop component technologies that support reasoning about quality attributes, the restrictions imposed or needed might reduce the functional capability of such a technology. There is a trade-off between the functional and analytic capabilities of such a component technology. It is important that what is produced has sufficient functional capability to be applicable to real world practical problems. If we fail to do this, the technologies developed will remain academic.

How can we verify the predictions to gain objective trust?

A theory explaining how the quality attributes of a system are determined by the assembled components is valuable or may even be essential if prediction of the attributes is required. If such a theory is available, it is possible to reason about the whole by knowing the parts. Such a theory can be implemented for automatic reasoning or for manual calculation. The predictions required should be available as soon as the components have been assembled. Access to information about the assembly and the component properties should make this possible. Prediction in practical applications requires the automated implementation of the theories. This also suggests that the reasoning theory and analytic model must be designed into the component technology from the beginning.

To be of any value, a prediction theory must be validated. It is shown in Chapter 5 that a set of samples for the validation must be selected carefully using argumentation from the theory and from the experimental results. We have developed a method consisting of several phases which are demonstrated with examples of latency and dependency properties. If the predictions give a statistical value, the validation method must also be based on statistical methods. It is important that the sample set is truly representative of the population of all possible assemblies.

If we cannot show that the sample set is representative of the population, it is difficult to obtain objective trust in the validation. However, it is possible to show how the validation is performed and by having domain knowledge, it is possible to achieve a certain degree of subjective trust. This trust might be sufficient, depending on the criticality of the target applications. Chapter 7 discusses the implications of basing design decisions on predicted quality attributes in more detail.

The lesson learned is that it is important to be able to show how the objective trust has been obtained. It is therefore important to file experiment data and also to be able to demonstrate the method of validation. It is also important to be able to verify that the selected sample for validation is reliably representative of the possible assembly population.

Which types of quality attributes are suitable for prediction?

Chapter 3 names many different quality attributes and their relation to prediction. The attributes that are directly composable are easier to reason about. For automatic reasoning of an attribute, it is important that it is possible to quantify the attribute and that there is information available of how the components concerned are assembled. There should be a reasoning theory about the attribute in question to permit prediction of the attributes of an assembly knowing only the component attributes.

If the quality attributes of a component are difficult to determine, it is not suitable for prediction. The first step to achieve predictability at the assembly level is to learn the theory of the attribute on the component level. An exception to this is derived attributes that are a function of several other attributes at the component level such as concurrency or deadlock-free. (a component cannot cause deadlock by itself, it must be interacting with other components to cause a deadlock).

We can conclude that attributes depending on the system environment or the usage profile are difficult to predict because of the difficulty of obtaining and specifying the system or usage information for the reasoning framework.

The classification of quality attributes could be extended and the development of analysis patterns and strategies for achieving predictability of the different types of attributes would be very valuable.

In general, experience has shown that the prediction or design of quality attributes of software system is difficult. The use of a component-based technology should simplify reasoning about quality attributes. Component-models specify the composition rules via interfaces and rules and by restricting and reusing rules it is possible to reuse reasoning techniques and in many cases, to automate the procedures

We have observed that in many cases it is very difficult to classify quality attributes because they overlap. They have different abstraction, different complexity and very often a vague specification. In different domains, they may have different meanings and are achieved by different means. Different reasoning frameworks or patterns for each attribute or class of attributes should be developed.

8.2 Future Work

The field of quality attribute prediction is extensive and more research can and should be performed in this field. Future work in the development of component-based technologies could include the implementation of the PECT approach for a particular component-model in industrial settings and the development of a number of property theories with their relations to the component model. Doing this would increase confidence in the use of research results to solve problems in practical industrial settings.

How to design and build component models that permit prediction is still to be investigated more thoroughly. Research is needed to determine how to develop the component model to make increasing numbers of quality attributes predictable i.e. to obtain an extendable component model that can be used in the future to reason about many different quality attributes.

Additional research is needed into the building of component technologies which support prediction, to find out what kinds of component model restrictions can be introduced and which cannot (or are very difficult to introduce). This would give us guidelines simplifying the process of building prediction-enabled component technologies.

This thesis has concentrated on measurable properties and the theories and their prediction. As future work, we propose to investigate further how to achieve objective trust in the attributes of a component and in the predictions made for them. More quality attributes should be studied to automate the reasoning about them.

There are many approaches to take for the different types of quality attributes. For certain attributes it might be possible to reuse reasoning frameworks and theories. Future work required in this field is more investigation into the predictability of quality attributes and property theories. Which quality attributes are important in which domains? Not all quality attributes are of importance for all problem domains and the classification of attributes according to their importance for certain domains would be of interest. This has been done in the dependable systems domain and there is now a platform and common understanding of the quality attributes for this domain [4].

Further research is needed to determine how the introduction of predictability affects component models. For example, how does the upgrades and evolution of a component model, to acquire more predictability, affect the maintainability of a software system in which it is incorporated? Will the introduction of predictability with its restrictions actually reduce maintainability? Consideration of these questions is proposed for future work.

Areas other than prediction of quality attributes can be considered for future work of achieving predictability by other means. Prediction of component-based development processes and their execution to know how long time it takes to implement a certain software function is a challenge.

Predictability of the development processes is as important as the predictability of the final quality attributes of the software. There is a constant demand to predict the cost, quality and the development time of a software product. In reality, green field development of new products is very rare. There is often legacy software to be taken into consideration during the development of a system. There are also other aspects of predictability which can be considered for future work.

9 References:

[1] World Wide Web Consortium, *XML* , http://www.w3c.org/XML/, 2001.

[2] ABB, ABB Inc., *ABB Robotics*, www.abb.com/robotics, 2004.

[3] Aoyama M., "New Age of Software Development: How component-based Software Engineering Changes the way of Software Development", In *Proceedings of 1st workshop on Component Based Software Engineering*, Software Engineering Institute, 1998.

[4] Avižienis A., Laprie J.-C., and Randell B., "Fundamental Concepts of Computer System Dependability", In *Proceedings of IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable, Robots in Human Environments*, 2001.

[5] Barbacci M., Klein M., Longstaff T., and Weinstock C. B., *Quality Attributes*, report CMU/SEI-95-TR-021, CMU/SEI, 1995.

[6] Barquin, R., Computer Ethics Institute, *The Ten Commandments of Computer Ethics*, http://www.cpsr.org/program/ethics/cei.html, 2003.

[7] Bass L., Clements P., and Kazman R., *Software Architecture in Practice* (2nd edition), ISBN 0-321-15495-9, Addison-Wesley, 2003.

[8] Bertoa M. F. and Vallecillo A., " Quality Attributes for COTS Components", In *Proceedings of 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pp. 54-66, 2002.

[9] Besnard, J. F., Keene, S. J., and Voas, J., *Assuring COTS Products for Reliability and Safety Critical Systems* , IEEE Computer Society, 1999.

[10] Bosch J., "Component Evolution in Product-Line Architectures", In *Proceedings of International Workshop on Component Based Software Engineering* , 1999.

[11] Bosch J., "Product-Line Architectures in Industry: A Case Study", In *Proceedings of 21st International Conference on Software Engineering*, ACM Press, 1999.

[12] Bosch J., *Design & Use of Software Architectures*, ISBN 0-201-67494-7, Addison-Wesley, 2000.

[13] Bosch J. and Stafford J., "Architecting Component-based Systems", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[14] Box G., Hunter W., and Hunter S., *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, ISBN 0471093157, Wiley-Interscience, 1978.

[15] Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition*, ISBN 0201835959, Addison-Wesley Longman, 1995.

[16] Brown A. W., *Large-Scale Component-Based Development*, Prentice Hall, 2000.

[17] Cheesman J. and Daniels J., *UML Components - A Simple Process for Specifying Component-Based Software*, ISBN 0-201-70851-5, Addison-Wesley, 2000.

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

[19] Cook J. E. and Dage J. A., "Highly Reliable Upgrading of Components", In *Proceedings of 21st International Conference on Software Engineering*, ACM Press, 1999.

[20] Crnkovic I., Schmidt H., Stafford J., and Wallnau K. C., *Anatomy of a Research Project in Predictable Assembly*, 2002.

[21] Crnkovic I., Asklund U., and Persson-Dahlqvist A., *Implementing and Integrating Product Data Management and Software Configuration Management*, ISBN 1-58053-498-8, Artech House, 2003.

[22] Crnkovic I., Küster J.K., Larsson M., and Lau K.-K., "Object-Oriented Design Frameworks: Formal Specifications and Some Implementation Issues", in Barzdins J. and Caplinskas A. (editors): *Databases and Information Systems*, ISBN 0-7923-6823-1, Kluwer Academic Publishers, 2001.

[23] Crnkovic I. and Larsson M., "A Case Study: Demands on Component-based Development", In *Proceedings of 22nd International Conference on Software Engineering*, ACM Press, 2000.

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

[25] Crnkovic I., Larsson M., and Lüders F., "Implementation of a Software Engineering Course for Computer Science Students", In *Proceedings of 7th Asia-Pacific Software Engineering Conference (APSEC)*, 2000.

[26] Crnkovic I., Schmidt H., Stafford J., and Wallnau K. C., "4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction", In *Software Engineering Notes*, volume 26, issue 6, pp. 33-40, 2001.

[27] Crnkovic I., Schmidt H., Stafford J., and Wallnau K. C., "5th Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly", In *Software Engineering Notes*, volume 27, issue 5, 2002.

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

[29] Dolbec J. and Shepard T., "A component based reliability model", In *Proceedings of Conference of the centre for advanced studies on collaborative research (CASCON)*, ACM, 1995.

[30] Dromey G.R., "A Model for Software Product Quality", In *IEEE Transaction on Software Engineering*, volume 21, issue 2, pp. 146-163, 1995.

[31] Eskenazi E. M., fioukov A. V., Hammer D. K., and Chaudron M. R., "Estimation of Static Memory Consumption for Systems Built from Source Code Components", In *Proceedings of 9th IEEE conference on engineering of computer based systems*, IEEE Computer Society Press, 2002.

[32] Estublier J. and Favre J.-M., "Component Models and Technology", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[33] Estublier J., Favre J.-M., and Sanlaville R., "Industrial Experience with Dassault Systeme Component Model", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, artech house, 2002.

[34] Fenton N. E. and Pfleeger S. L., *Software Metrics - A Rigorous & Practical Approach* (2nd edition), ISBN 0-534-95425-1, PWS Publishing Company, 1997.

[35] fioukov A. V., Eskenazi E. M., Hammer D. K., and Chaudron M. R., "Evaluation of static properties for component-based architectures", In *Proceedings of 28th IEEE Euromicro Conference*, IEEE, 2002.

[36] Freud S., "New introductory lectures on psychoanalysis. Lecture 33: Femininity", In *lectures on psychoanalysis, Standard Edition*, volume 22, issue 1, pp. 136-157, 1933.

[37] Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns - Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2, Addison-Wesley, 1995.

[38] Gorton I., Liu A., and Brebner P., "Rigorous Evaluation of COTS Middelware Technology", In *IEEE Computer*, volume 36, issue 3, pp. 50-55, 2003.

[39] Gyllenswärd E. and Kap M., "A Framework for Integrating Business Applications", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[40] Heineman G. T. and Councill W. T., *Component-based Software Engineering, Putting the Pieces Together*, ISBN 0-201-70485-4, Addison-Wesley, 2001.

[41] Hinman, L. M., University of San Diego, *Lectures on Ethical Theory*, http://ethics.acusd.edu/video/Hinman/Theory/, 2001.

[42] Hissam S. A., Moreno G. A., Stafford J., and Wallnau K. C., *Packaging Predictable Assembly with Prediction-Enabled Component Technology*, report Technical report CMU/SEI-2001-TR-024 ESC-TR-2001-024, 2001.

[43] Hissam S. A., Hudak J., Ivers J., Klein M., Larsson M., Moreno G. A., Northrop L., Plakosh D., Stafford J., Wallnau K. C., and Wood W., *Predictable Assembly of Substation Automation Systems: An Experience Report*, report CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 2002.

[44] Hissam S. A., Hudak J., Ivers J., Klein M., Larsson M., Moreno G. A., Northrop L., Plakosh D., Stafford J., Wallnau K. C., and Wood W., *Predictable Assembly of Substation Automation Systems: An Experience Report, Second Edition*, report CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 2003.

[45] Hissam S. A. and Ivers J., *PECT Infrastructure: A Rough Shetch*, report CMU/SEI-2002-TN-033, Software Engineering Institute, Carnegie Mellon University, 2002.

[46] Hissam S.A., Moreno G. A., Stafford J., and Wallnau K. C., "Enabling Predictable Assembly", In *Journal of Systems & Software*, volume 65, issue 3, pp. 185-198, 2003.

[47] Hissam S. A., Stafford J., and Wallnau K. C., *Volume III: Anatomy of a Reasoning-Enabled Component Technology*, report CMU/SEI-2001-TR-007, Software Engineering Institute, Carnegie Mellon University, 2001.

[48] Hoare C.A., "Communicating sequential processes", In *Communicatinos of the ACM*, volume 21, issue 8, pp. 666-677, 1978.

[49] IEC, *Application and Implementation of IEC 1131-3*, IEC Geneva, 1995.

[50] IEC, *IEC 618510: Communication Networks amd Systems in Substations*, report IEC-TC57-WG10/11/12-Draft, International Electrotechnical Commission, 1999.

[51] IEEE, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard*, Institute of Electrical and Electronics Engineers, 1990.

[52] IEEE, IEEE-CS/ACM, *Software Engineering Code of Ethics and Professional Practice*, http://www.computer.org/tab/seprof/code.htm, 2003.

[53] International Electrotechnical Comission, *Programmable Controllers - Part 3: Programming Languages*, report IEC 61131-3, 1992.

[54] Inverardi P. and Tivoli M., "Deadlock-free software architectures for COM/DCOM applications", In *Journal of Systems & Software*, volume 65, issue 3, pp. 173-183, 2003.

[55] ISO/IEC, *Information technology - Software product quality - Part 1: Quality model*, report ISO/IEC FDIS 9126-1:2000 (E), ISO, 2000.

[56] ISO/IEC, *Software Qualities*, report ISO/IEC FDIS 9126-1:2000(E), International Standards Organization, 2000.

[57] Ivers J., Sinha N., and Wallnau K. C., *A Basis for Composition Language CL*, report CMU/SEI-2002-TN-026, Software Engineering Institute, Carnegie Mellon University, 2002.

[58] Jogalekar P. and Woodside M., "Evaluating the Scalability of Distributed Systems", In *IEEE Transactions on Parallel & Distributed Systems*, volume 11, issue 6, pp. 589-604, 2000.

[59] Josefsson M., *Programvarukomponenter i praktiken -att köpa tid och prestera mer*, report V040078, Sveriges Verkstadsindustrier, 1999.

[60] Kazman R., Bass L., Abowd G., and Webb M., "SAAM: A Method for Analyzing the Properties of Software Architectures", In *Proceedings of The 16th International Conference on Software Engineering*, 1994.

[61] Kazman R., Klein M., and Clements P., *ATAM: Method for Architecture Evaluation*, report CMU/SEI-2000-TR-004, Software engineering institute, 2000.

[62] Klein M., Ralya T., Pollak B., and Obenza R., *A Practitioner's Handbook for Real-Time Analysis*, Kluwer Academic Publishers, 1993.

[63] Kotonya G. and Rashid A., "A strategy for Managing Risks in Component-based Software Development", In *Proceedings of 27th Euromicro Conference in the CBSE workshop*, pp. 12-21, IEEE Computer Society, 2001.

[64] Kruchten P., *The Rational Unified Process: An Introduction* (2nd edition), ISBN 0-201-70710-1, Addison-Wesley, 2000.

[65] Land R., "Improving Quality Attributes of a Complex System Through Architectural Analysis - A Case Study", In *Proceedings of 9th IEEE Conference on Engineering of Computer-Based Systems (ECBS)*, IEEE, 2002.

[66] Larsson M., *Applying Configuration Management Techniques to Component-Based Systems*, Licentiate Thesis, Dissertation 2000-007, Department of Information Technology Uppsala University., 2000.

[67] Larsson M. and Crnkovic I., "New Challenges for Configuration Management", In *Proceedings of 9th Symposium on System Configuration Management*, Lecture Notes in Computer Science, nr 1675, Springer Verlag, 1999.

[68] Larsson M. and Crnkovic I., "Component Configuration Management", In *Proceedings of 5th Workshop on Component Oriented Programming*, 2000.

[69] Larsson M. and Crnkovic I., "Development Experiences from a Component Based System", In *Proceedings of 7th Engineering Conference on Computer Based Systems*, IEEE Computer Society, 2000.

[70] Lindgren M., Hansson H., and Thane H., "Using Measurements to Derive the Worst-Case Execution Time", In *Proceedings of 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, 2000.

[71] Lüders F., Lau K.-K., and Ho S.-M., "Specification of Software Components", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[72] Martin M. W. and Schinzinger R., *Ethics in Engineering*, ISBN 0-07-040849-1, McGraw-Hill, 1996.

[73] McCabe T.J., A Complexity Measure, *IEEE Transaction on Software Engineering*, volume 2, 1976.

[74] McCall J., Richards P. K., and Gene F., *Factors in Software Quality, volume III: Preliminary Handbook on Software Quality for an Acqusition Manager*, report RADC-TR-77-369, vol III, Hanscom AFB, MA 01731, 1977.

[75] Microsoft Corporation, *The Component Object Model Specification, v0.99*, 1996.

[76] Mikhajlov L. and Sekerinski E., "A Study of the Fragile Base Class Problem", In *Proceedings of ECOOP'98 - 12th European Conference on Object-Oriented Programming*, nr Lecture Notes in Computer Science 1445, pp. 355-382, Springer-Verlag, 1998.

[77] Moreno G. A., Hissam S. A., and Wallnau K. C., "Statistical Models for Empirical Component Properties and Assembly-Level Property Predictions: Toward Standard Labeling", In *Proceedings of 5th Workshop on component based software engineering*, 2002.

[78] Morisio M., Seaman C. B., Parra A. T., Basil V. R., Kraft S. E., and Condon S. E., "Investigating and Improving a COTS-Based Software Development Process", In *Proceedings of 22nd International Conference on Software Engineering*, ACM Press, 2000.

[79] Morris J., Lee G., Parker K., Bundell G., and Peng Lam C., "Software Component Certification", In *IEEE Computer*, volume 34, issue 9, pp. 30-36, 2001.

[80] NCSU, North Carolina State University, *Ethics in Computing*, http://legacy.eos.ncsu.edu/eos/info/computer_ethics/, 2003.

[81] OPC, *OLE for Process Control*, report v1.0, OPC Standards Collection, OPC Foundation, 1998.

[82] Oreizy P., Medvidovic N., and Taylor R. N., "Architecture-Based Runtime Evolution", In *Proceedings of 20th international conference on software engineering*, ACM Press, 1998.

[83] Persson-Dahlqvist A., Crnkovic I., and Larsson M., "Managing Complex Systems - Challenges for PDM and SCM", In *Proceedings of International Symposium on Software Configuration Management, SCM 10*, 2001.

[84] Plasil F. and Visnovsky S., "Behavior protocols for software components", In *IEEE Transaction on Software Engineering*, volume 28, issue 11, pp. 1056-1076, 2002.

[85] Preiss O. and Naedele M., "Architectural Support for Reuse: A Case Study in Industrial Automation", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[86] Preiss O. and Wegmann A., "Towards a Composition Model Problem Based on IEC61850", In *Proceedings of 4th workshop on component based software engineering*, 2001.

[87] Preiss O., Wegmann A., and Wong J., "On Quality Attribute Based Software Engineering", In *Proceedings of EUROMICRO 2001 CBSE workshop*, 2001.

[88] Rabinovich S. G., *Measurement Errors and Uncertainties* (2nd edition), ISBN 0-387-98835-1, Springer Verlag, 2000.

[89] Reussner R.H., Schmidt H., and Poernomo I. H., "Reliability Prediction for Component-based Software Architectures", In *Journal of Systems & Software*, volume 66, issue 3, pp. 241-252, 2003.

[90] Robson C., *Real World Research* (2nd edition), ISBN 0-631-21305-8, Blackwell Publishers, 2002.

[91] Schmidt H., "Trustworthy components: compositionality and prediction", In *Journal of Systems & Software*, volume 65, issue 3, pp. 215-225, 2003.

[92] Schmidt H. and Reussner R. H., " Parametrized Comtracts and Adapter Synthesis", In *Proceedings of 5th ICSE workshop on CBSE*, 2001.

[93] Schultz W. and Polze A., "Aspect-Oriented Programming with C# and .NET", In *Proceedings of 5th international symposium on object oriented real time distributed computing*, pp. 241-248, IEEE Computer Society, 2002.

[94] Shaw M., "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.

[95] Shaw M., "The Coming-of-Age of Software Architecture Research", In *Proceedings of 23rd International Conference on Software Engineering (ICSE)*, pp. 657-664, ACM, 2001.

[96] Shaw M. and others, Abstractions for Software Architecture and Tools to Support Them, *IEEE Transaction on Software Engineering*, volume 21, issue 24, 1995.

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

[98] Siddiq J. and Shekran C., "Requirements engineering: The emerging wisdom", In *IEEE Software*, volume 13, issue 2, pp. 15-15, 1996.

[99] Sommerville I., *Software Engineering*, ISBN 0-201-39815-X, Addison-Wesley, 2001.

[100] Stafford J. and mcGregor J., "Issues in Predicting the Reliability of Composed Components", In *Proceedings of 5th workshop on component based software engineering*, 2002.

[101] Stafford J. and Wallnau K.C., "Component Composition and Integration", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[102] Stewart D.B., Volpe R.A., and Khosla P.K., Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, *IEEE Transaction on Software Engineering*, volume 23, issue 12, 1997.

[103] Svahnberg M., *Supporting Software Architecture Evolution*, Ph.D. Thesis, Blekinge Institute of Technology, Sweden, 2003.

[104] Szyperski C., *Component Software - Beyond Object-Oriented Programming*, ISBN 0-201-17888-5, Addison-Wesley, 1998.

[105] Takeshita T., "Metrics and risks of CBSE", In *Proceedings of 5th international symposium on software tools and technologies*, IEEE Computer Society, 1997.

[106] Thai T. and Lam H., *.NET Framework*, O´Reilly, 2001.

[107] Thane H., *Monitoring, Testing and Debugging of Distributed Real-Time Systems*, Doctoral Thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000.

[108] Tichy W., "Should computer scientists experiment more?", In *IEEE Computer*, volume 31, issue 5, pp. 32-40, 1998.

[109] Tichy W., Lukowicz P., Prechelt L., and Heinz E. A., "Experimental evaluation in computer science: A quantitative study", In *Journal of Systems & Software*, volume 28, issue 1, pp. 9-19, 1995.

[110] van der Hoek A., "Capturing Product Line Architectures", In *Proceedings of 4th International Software Architecture Workshop*, ACM Press, 2000.

[111] van Ommering R., "The Koala Component Model", in Crnkovic I. and Larsson M. (editors): *Building Reliable Component-Based Software Systems*, ISBN 1-58053-327-2, Artech House, 2002.

[112] Voas J., "Cerifying Off-the-Shelf Software Components", In *IEEE Computer*, volume 31, issue 6, pp. 53-59, 1998.

[113] Voas J., "A Method for Discovering Unforeseen Software Output, Modes and Missing System Hazards", In *The Annals of Software Engineering*, 2001.

[114] Voas J. and McGraw G., *Software fault injection: inoculating programs against errors*, John Wiley and Sons, 1998.

[115] Voas J. and Payne J., "Dependability Certification of Software Components", In *Journal of Software Systems*, volume 52, pp. 165-172, 2000.

[116] Wall A., *Architectural Modeling and Analysis of Complex Real-Time Systems*, Doctoral Thesis, Dissertation No. 5, Department of Computer Science and Engineering, Mälardalen University, 2003.

[117] Wall A., Larsson M., and Norström C., "Towards an Impact Analysis for Component Based Real-Time Product Line Architectures", In *Proceedings of Euromicro Conference on Component Based Software Engineering*, 2002.

[118] Wall A. and Norström C., "A Component Model for Embedded Real-Time Software product-Lines", In *Proceedings of 4th IFAC conference on Fieldbus Systems and their Applications*, 2001.

[119] Wallnau K. C., *Volume III: A Technology for Predictable Assembly from Certifiable Components*, report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.

[120] Wallnau K. C. and Ivers J., *Snapshot of CCL: A Language for Predictable Assembly*, report CMU/SEI-2003-TN-025, Software Engineering Institute, 2003.

[121] Wallnau K. C. and Stafford J., "Ensembles: Abstractions for A New Class of Design Problem", In *Proceedings of the 27th Euromicro Conference*, pp. 48-55, 2001.

[122] Warmer J. and Kleppe A., *The Object Constraint Language: Precise Modeling with UML*, ISBN 0201379406, Addison-Wesley, 1999.

[123] Wohlin C. and Regnell B., "Achieving industrial relevance in software engineering education", In *Proceedings of 12th Conference on Software Engineering Education and Training*, pp. 16-25, IEEE, 1999.

[124] Wohlin C. and Runeson P., "Certification of Software Components", In *IEEE Transaction on Software Engineering*, volume 20, issue 6, pp. 494-500, 1994.

[125] Yan L., Gorton I., Liu A., and Chen S., "Evaluating the scalability of enterprise javabeans technology", In *Proceedings of 9th Asia-Pacific Software Engineering Conference*, pp. 74-83, IEEE, 2002.

Appendix A

This appendix contains measurement data from the sample sets used in the controller experiment and a description of selected quality attributes.

Table 19: Recorded results of predicted and average measure latency for the first controller sample set

Sample	Assembly	Task	Job	Predicted Latency (ms)	Measured Average Latency (ms)	MRE	Standard Deviation (ms)
1	0	1	1	12.157	12.181	-0.20%	0.0032
2	11	1	1	152.203	142.800	6.59%	0.0025
3	11	2	1	223.730	212.652	5.21%	0.0011
4	11	3	1	146.122	136.769	6.84%	0.0019
5	11	3	2	108.058	105.056	2.86%	0.0019
6	11	3	3	108.058	105.078	2.84%	0.0020
7	11	3	4	108.058	105.063	2.85%	0.0020
8	11	4	1	295.343	288.478	2.38%	0.0004
9	11	4	2	179.671	175.116	2.60%	0.0007
10	11	4	3	179.671	175.123	2.60%	0.0006
11	11	4	4	179.671	175.122	2.60%	0.0006
12	12	1	1	36.480	36.373	0.30%	0.0045
13	12	2	1	12.160	12.150	0.25%	0.0024
14	14	1	1	106.665	98.740	8.03%	0.0031
15	14	2	1	140.991	128.770	9.49%	0.0037
16	14	3	1	38.779	37.785	2.63%	0.0007
17	14	3	2	19.798	18.844	5.07%	0.0033
18	14	4	1	72.320	68.429	5.69%	0.0017
19	14	4	2	53.339	49.534	7.68%	0.0031
20	15	1	1	48.077	47.991	0.18%	0.0020
21	16	1	1	6.078	6.090	-0.27%	0.0048
22	18	1	1	6.078	6.088	-0.25%	0.0044
23	19	1	1	6.078	6.099	-0.37%	0.0036
24	1	1	1	593.455	634.291	-6.31%	0.0347
25	1	2	1	995.948	995.237	0.07%	0.0004

Sample	Assembly	Task	Job	Predicted Latency (ms)	Measured Average Latency (ms)	MRE	Standard Deviation (ms)
26	1	3	1	754.928	754.640	0.04%	0.0003
27	1	4	1	553.665	460.407	20.26%	0.0002
28	21	1	1	6.078	6.086	-0.31%	0.0055
29	22	1	1	24.051	23.980	0.45%	0.0070
30	22	2	1	14.563	14.548	0.19%	0.0012
31	22	2	2	8.485	8.485	-0.38%	0.0041
32	23	1	1	72.517	72.180	0.47%	0.0027
33	23	2	1	41.100	41.004	0.27%	0.0011
34	24	1	1	65.423	62.560	0.09%	0.0016
35	24	2	1	6.079	6.101	-0.48%	0.0042
36	25	1	1	69.194	66.032	4.79%	0.0039
37	25	2	1	16.965	16.945	0.13%	0.0012
38	27	1	1	233.001	233.775	-6.30%	0.0582
39	27	2	1	304.572	279.491	10.18%	0.1280
40	27	3	1	445.338	429.370	3.72%	0.0002
41	27	3	2	140.767	137.264	2.55%	0.0001
42	27	4	1	161.388	155.648	3.69%	0.0004
43	28	1	1	12.156	12.105	0.78%	0.0136
44	28	2	1	31.941	31.872	0.24%	0.0009
45	28	2	2	25.863	25.798	0.29%	0.0015
46	2	1	1	12.157	12.191	-0.32%	0.0031
47	30	1	1	207.821	172.140	21.24%	0.0771
48	30	1	2	70.788	67.999	4.10%	0.0018
49	30	2	1	187.239	192.602	-5.95%	0.0263
50	30	3	1	166.675	135.748	22.78%	0.0018
51	30	4	1	146.124	117.977	23.86%	0.0005
52	31	1	1	1087.457	1032.781	5.29%	0.0009
53	31	2	1	1036.500	788.620	31.43%	0.0009
54	3	1	1	6.078	6.092	-0.33%	0.0039
55	5	1	1	12.908	12.921	-0.17%	0.0024
56	7	1	1	20.555	20.531	0.24%	0.0024
57	7	1	2	20.555	20.507	0.33%	0.0024
58	7	1	3	20.555	20.519	0.28%	0.0024
59	7	1	4	20.555	20.510	0.33%	0.0025
60	7	1	5	20.555	20.515	0.29%	0.0025
61	7	2	1	48.552	48.502	0.14%	0.0008
62	8	1	1	58.063	58.013	0.10%	0.0006
63	8	2	1	37.516	37.452	0.20%	0.0009
64	9	1	1	64.903	64.928	-0.21%	0.0019
65	9	2	1	32.412	32.435	-0.14%	0.0017

Table 20: Recorded results of predicted and average measure latency for the second controller sample set

Sample	Assembly	Task	Predicted AVG Task Latency (ms)	Measured AVG Task Latency (ms)	MRE
1	1	1	10.45	10.46	-0.16%
2	2	1	10.45	10.46	-0.16%
3	3	1	10.45	10.47	-0.20%
4	4	1	10.45	10.48	-0.33%
5	5	1	10.64	10.65	-0.05%
6	6	1	5.22	5.24	-0.36%
7	7	1	10.64	10.65	-0.07%
8	8	1	10.64	10.65	-0.06%
9	9	1	10.47	10.43	0.30%
10	9	2	26.21	26.01	0.78%
11	10	1	16.91	16.94	-0.18%
12	10	2	39.23	39.20	0.09%
13	11	1	21.29	21.32	-0.15%
14	11	2	15.87	15.85	0.14%
15	12	1	26.90	26.48	1.60%
16	12	2	64.44	63.44	1.57%
17	12	3	86.12	84.73	1.64%
18	12	4	96.76	95.21	1.63%
19	13	1	31.34	31.35	-0.01%
20	13	2	10.45	10.49	-0.37%
21	14	1	32.32	31.88	1.39%
22	14	2	70.06	68.86	1.74%
23	14	3	51.09	50.24	1.69%
24	14	4	77.99	76.64	1.76%
25	15	1	5.22	5.24	-0.32%
26	16	1	10.45	10.47	-0.19%
27	17	1	10.45	10.48	-0.34%
28	18	1	5.22	5.24	-0.33%
29	19	1	10.64	10.65	-0.03%
30	20	1	5.22	5.24	-0.37%
31	21	1	10.64	10.64	-0.02%
32	22	1	31.04	30.83	0.69%
33	22	2	28.74	28.49	0.86%
34	23	1	5.22	5.25	-0.42%
35	24	1	12.49	12.51	-0.17%
36	24	2	13.31	13.31	0.01%
37	25	1	54.72	54.75	-0.07%
38	25	2	5.22	5.24	-0.37%
39	26	1	20.90	20.91	-0.05%
40	26	2	10.45	10.49	-0.36%
41	27	1	54.69	54.45	0.43%
42	27	2	14.14	14.14	0.00%

Sample	Assembly	Task	Predicted AVG Task Latency (ms)	Measured AVG Task Latency (ms)	MRE
43	28	1	17.30	17.26	0.24%
44	28	2	22.87	22.62	1.08%
45	28	3	11.00	11.02	-0.18%
46	29	1	203.39	201.13	1.12%
47	29	2	55.10	53.49	3.01%
48	29	3	123.08	121.36	1.41%
49	29	4	17.56	16.25	8.08%
50	30	1	15.87	15.85	0.09%
51	30	2	23.90	23.91	-0.05%
52	31	1	204.20	201.66	1.26%
53	31	2	344.02	338.83	1.53%
54	31	3	58.96	58.16	1.38%
55	31	4	48.19	47.49	1.47%
56	32	1	5.22	5.24	-0.29%
57	33	1	214.89	214.51	0.18%
58	33	2	34.43	34.45	-0.05%
59	33	3	88.24	88.22	0.02%
60	33	4	148.64	148.53	0.07%
61	34	1	10.45	10.46	-0.16%
62	35	1	7.07	7.09	-0.27%
63	35	2	13.22	13.21	0.06%
64	36	1	437.79	437.83	-0.01%
65	36	2	17.21	17.25	-0.23%
66	36	3	33.51	33.52	-0.05%
67	36	4	68.25	68.22	0.05%
68	37	1	10.45	10.47	-0.18%
69	38	1	202.29	202.42	-0.06%
70	38	2	229.65	229.39	0.11%
71	38	3	12.29	12.34	-0.35%
72	38	4	39.96	39.94	0.07%
73	39	1	10.45	10.46	-0.15%
74	40	1	7.99	8.01	-0.25%
75	40	2	8.81	8.82	-0.06%
76	41	1	222.58	222.66	-0.04%
77	41	2	98.37	98.30	0.07%
78	41	3	114.67	114.60	0.06%
79	41	4	94.68	94.63	0.06%
80	42	1	10.45	10.47	-0.21%
81	43	1	10.64	10.65	-0.05%
82	44	1	229.52	228.74	0.34%
83	44	2	23.36	22.89	2.06%
84	44	3	31.35	31.34	0.04%
85	44	4	33.51	33.51	-0.02%
86	45	1	5.22	5.24	-0.32%

Sample	Assembly	Task	Predicted AVG Task Latency (ms)	Measured AVG Task Latency (ms)	MRE
87	46	1	41.16	40.87	0.70%
88	46	2	26.82	26.70	0.47%
89	47	1	9.23	9.23	-0.01%
90	47	2	6.15	6.16	-0.21%
91	48	1	6.45	6.47	-0.22%
92	48	2	9.84	9.86	-0.18%
93	49	1	473.42	470.87	0.54%
94	49	2	5.22	5.25	-0.58%
95	49	3	6.97	6.98	-0.21%
96	49	4	9.58	9.58	-0.08%
97	50	1	85.14	84.68	0.54%
98	50	2	49.10	48.65	0.93%
99	50	3	90.37	89.44	1.04%
100	51	1	95.98	95.21	0.81%
101	51	2	5.22	5.25	-0.55%
102	51	3	26.31	26.27	0.18%
103	51	4	21.09	21.06	0.14%
104	52	1	362.96	363.22	-0.07%
105	52	2	343.90	343.66	0.07%
106	53	1	10.45	10.47	-0.18%
107	54	1	211.52	211.17	0.16%
108	54	2	233.03	232.76	0.12%
109	54	3	53.19	53.15	0.07%
110	54	4	27.98	28.03	-0.18%
111	55	1	10.45	10.48	-0.27%
112	56	1	15.87	15.90	-0.21%
113	57	1	194.01	192.85	0.60%
114	57	2	26.90	26.93	-0.10%
115	57	3	129.44	128.48	0.74%
116	57	4	148.50	148.04	0.31%
117	58	1	10.45	10.46	-0.13%
118	59	1	38.63	38.48	0.38%
119	59	2	15.25	15.27	-0.09%
120	60	1	9.84	9.85	-0.12%
121	60	2	11.99	11.99	-0.03%
122	61	1	6.15	6.17	-0.28%
123	61	2	8.55	8.57	-0.20%
124	62	1	189.48	189.41	0.04%
125	62	2	132.19	132.13	0.05%
126	62	3	123.17	123.09	0.07%
127	62	4	25.42	25.46	-0.15%
128	63	1	237.92	235.61	0.98%
129	63	2	62.94	62.05	1.44%
130	63	3	5.22	5.25	-0.53%

Sample	Assembly	Task	Predicted AVG Task Latency (ms)	Measured AVG Task Latency (ms)	MRE
131	63	4	6.53	6.55	-0.29%
132	64	1	31.54	31.55	-0.03%
133	64	2	10.45	10.48	-0.35%
134	65	1	172.34	171.93	0.24%
135	65	2	38.12	38.17	-0.11%
136	66	1	10.45	10.47	-0.26%
137	67	1	171.85	171.95	-0.06%
138	67	2	217.35	217.20	0.07%
139	67	3	41.20	41.23	-0.07%
140	67	4	215.51	215.33	0.08%
141	68	1	10.64	10.65	-0.08%
142	69	1	10.64	10.65	-0.07%
143	70	1	32.55	32.34	0.68%
144	70	2	14.09	14.00	0.68%
145	71	1	396.22	393.78	0.62%
146	71	2	436.70	433.99	0.62%
147	71	3	219.27	217.94	0.61%
148	71	4	142.90	142.25	0.46%
149	72	1	10.64	10.65	-0.07%
150	73	1	10.64	10.65	-0.06%
151	74	1	32.55	32.34	0.67%
152	74	2	14.09	14.15	-0.41%
153	75	1	396.22	393.63	0.66%
154	75	2	436.70	433.99	0.62%
155	75	3	219.27	237.63	-7.72%
156	75	4	142.90	142.25	0.46%

Appendix B

Table 21: Short description of selected quality attributes

Higher level attribute	Quality attribute	Description
Usability	Accessability	The ability to be accessible
	Administrability	The ability to be easily administrated (data are easy to administrate)
	Understandability	The degree of easiness to understand (the function, protocol, interaction)
	Generality	The breadth of potential application of program components.
	Operability	The ease of operation of a program.
	Simplicity	The degree to which a program can be understood without difficulty.
Portability	Mobility	The effort required transferring the program from one hardware and/or software system environment. The ability of a system to execute on different hardware and software platforms.
	Nomadicity	The ability to move operation between different nodes
	Hardware independence	The degree to which the software is decoupled from the hardware on which it operates.
	Software system independence	The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints.
Performance	Accuracy	The precision of computations and control.
	Footprint	Size of memory used
	Responsiveness	Time to response to an input
	Scalability	The ability of a system to support modifications that dramatically increase the size of the system.
	Schedulability	Ability to schedule an operation
	Timeliness	The ability to perform a task at the correct time
	CPU utilization	The percentage of CPU execution time
	Latency	time between input and output

Higher level attribute	Quality attribute	Description
	Transaction Throughput	Number of transaction processed per unit of time
	Concurrency	ability to perform operations in parallel
	Efficiency	The amount of computing resources and code required by a program to perform its function.
Maintainability	Flexibility	The effort required to modify an operational program, or, the ease with which the systems can be adapted to changes.
	Evolvability	The ability to continoulsy being changed
	Extensibility	The ability for adding new functonality
	Modifiability	The ability of a system to be extended to accomplish additional functionality
	Upgradeability	The ability of a system to be updrated
	Expandability	The degree to which architectural, data, or procedural design can be extended.
Integrability	Data consistency	The absence of contradictory data in the system. The use of uniform design and documentation techniques throughout the software development project.
	Version consistency	Ability of avoiding version mismatch in a configuration
	Adaptability	The ability for adaptation to new requirements or new environment or similar
	Composeability	The ability to be integrated with the parts of a system
	Interoperability	The ability of a system to work with another system
	Openness	Ability to integrate new functions developed by third party
	Heterogenity	Ability to integrated heterogeneous elements /software and hardware)
	Integrability	The ability to make the separately developed components of a system work correctly together
Design	Audibility	The ease with which conformance to standards can be checked
	Completeness	The degree to which full implementation of required function has been achieved.
	Conciseness	The compactness of the program in terms of lines of code.
	Correctness	The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

Higher level attribute	Quality attribute	Description
	Testability	The effort required to test a program to ensure that it performs its intended function.
	Traceability	The ability to trace a design representation or actual program component back to requirements.
	Coherence	The degree of consistency in the design
	Analyzability	The ability to analyze particular properties
	modularity	The functional independence of program components.
	Reusability	The extent to which a program [or parts of a program] can be reused
Deployability	Configureability	The ability to configure the artifact during deployment
	Distributeability	The ability to distribute the artifact
	Ease of creation	The easiness of constructing the system. Often measured in labor hours.
	Availability	The probability that the system functions correctly.
	Confidentality	Absence of unauthorized disclosure of information
	Integrity	The extent to which access to software or data by unauthorized persons can be controlled.
	Maintainability	The effort required to locate and fix an error in a program (this is a very limited definition).
	Reliability	The ability of the system to sustain operations. A common measure is mean time between failures.
	Safety	Absence of catastrophic consequences on the users and the environment
	Security	The availability of mechanisms that control of protect programs and data.
Business	Cost	The cost of the product
	Projected Lifetime	Depending on the lifetime of the product, scalability, maintainability and portability becomes important
	Targeted Market	The targeted market volume
	Time to market	The time it takes to make the product available on the market
	Affordability	How affordable is the component
	Development time	The time it takes to develop a component

Table 22 presents the results of the survey performed with the researchers. Where 60% of the researchers are in agreement that is indicated with the majority result, 0 or 1. A value of 0 indicates no relation and 1 indicates relation with the classification. Disagreement is displayed with a –1. Confidence is in the range from 0 to 5 where 0 is no confidence and 5 is great confidence.

Table 22: Classification according to the survey with researchers

Higher level attribute	Quality attribute	a) Directly composable	b) Architecture-related	c) Derived	d) Usage-dependent	e) System environment context	Level of agreement	Level of confidence 2 and above
Usability	Accessibility	0	1	-1	1	1	80%	67%
	Administrability	-1	1	0	-1	1	60%	75%
	Generality	1	0	-1	-1	0	60%	50%
	Operability	0	1	1	1	1	100%	75%
	Simplicity	0	1	1	-1	-1	60%	92%
	Understandability	0	1	1	0	-1	80%	92%
Portability	Hardware independence	1	1	0	0	1	100%	83%
	Mobility	1	1	-1	0	1	80%	92%
	Nomadicity	-1	1	0	0	-1	60%	75%
	Software system independence	1	1	0	0	0	100%	83%
Performance	Accuracy	1	1	1	0	-1	80%	92%
	Concurrency	-1	1	0	0	0	80%	75%
	CPU utilization	1	1	-1	0	0	80%	92%
	Efficiency	1	1	1	-1	-1	60%	83%
	Footprint	1	1	-1	-1	0	60%	92%
	Latency	-1	1	-1	-1	1	40%	92%
	Responsiveness	-1	1	1	0	1	80%	92%
	Scalability	0	1	-1	0	1	80%	75%
	Schedulability	-1	1	0	0	1	80%	83%
	Timeliness	0	1	-1	1	1	80%	92%
	Transaction Throughput	0	1	1	0	1	100%	83%
Maintainability	Evolvability	-1	1	1	0	-1	60%	83%
	Expandability	0	1	1	-1	-1	60%	58%

Higher level attribute	Quality attribute	a) Directly composable	b) Architecture-related	c) Derived	d) Usage-dependent	e) System environment context	Level of agreement	Level of confidence 2 and above
	Extensibility	-1	1	1	0	-1	60%	83%
	Flexibility	-1	1	1	0	-1	60%	83%
	Modifiability	-1	1	1	0	-1	60%	75%
	Upgradeability	-1	1	1	-1	1	60%	75%
Integrability	Adaptability	-1	1	1	-1	1	60%	83%
	Composeability	-1	1	1	0	-1	60%	75%
	data consistency	0	1	1	0	-1	80%	83%
	Heterogeneity	0	1	0	0	0	100%	50%
	Integrability	-1	1	-1	0	0	60%	75%
	Interoperability	-1	1	1	-1	1	60%	83%
	Openness	0	1	-1	1	-1	60%	75%
	Version consistency	-1	-1	0	0	-1	40%	75%
Design	Analyzability	-1	-1	-1	0	0	40%	75%
	Audibility	1	1	-1	-1	-1	40%	67%
	Coherence	0	1	-1	0	-1	60%	92%
	Completeness	1	1	-1	-1	-1	40%	75%
	Conciseness	1	0	0	0	0	100%	83%
	Correctness	0	1	0	1	1	100%	83%
	modularity	1	1	-1	0	0	80%	75%
	Reusability	1	1	-1	0	0	80%	75%
	Testability	1	1	-1	-1	-1	40%	92%
	Traceability	1	1	0	0	0	100%	92%
Deployability	Configureability	1	1	0	1	-1	80%	83%
	Distributeability	1	1	-1	1	1	80%	83%
	Ease of creation	-1	1	0	-1	-1	40%	92%

Higher level attribute	**Quality attribute**	**a) Directly composable**	**b) Architecture-related**	**c) Derived**	**d) Usage-dependent**	**e) System environment context**	**Level of agreement**	**Level of confidence 2 and above**
Dependability	Availability	-1	1	0	-1	-1	40%	92%
	Confidentiality	-1	1	0	1	-1	60%	92%
	Integrity	-1	1	0	1	1	80%	92%
	Maintainability	1	1	-1	0	-1	60%	92%
	Reliability	1	1	-1	1	1	80%	83%
	Safety	0	1	0	1	1	100%	83%
	Security	1	1	-1	-1	-1	40%	83%
Business	Affordability	-1	1	0	1	-1	60%	67%
	Cost	-1	1	0	0	0	80%	67%
	Development time	-1	1	0	1	0	80%	67%
	Projected Lifetime	1	1	0	1	1	100%	67%
	Targeted Market	-1	-1	0	1	-1	40%	58%
	Time to market	-1	1	0	1	-1	60%	58%
Level of agreement		57%	95%	66%	74%	56%		