Predicting Quality Attributes in Component-based Software Systems

Magnus Larsson
March 2004
Abstract

Building component-based applications adds new possibilities for reasoning about quality attributes. A component-based software architecture, facilitates reuse and control over the architectural parts in a software application. In general, reuse of software is very difficult to achieve and it has been shown that a systematic approach has to be taken when designing a system from reusable components or when designing components for later reuse. Further, reuse of functionality through components is limited if there are no means to know the behavior or quality attributes of the components, as well as their impact on the system.

A problem with software components and their reuse is that when the components are assembled together it is hard to predict the quality attributes of the resulting system. Usually the components are assembled according to some functional design description that is visible via the well-defined functional interfaces. However, the behavior, performance, scalability or reliability, of a final software system is not so easily determined.

Increased reuse would be possible if quality attributes could be specified and predicted. This is one motivation for introducing predictability. Another motivation is that predictability can reduce the testing of component-based applications by allowing the designers to reason about quality attributes before implementing and testing the applications.
This thesis investigates the possibility to develop component technologies that provide prediction mechanisms for quality attributes of assemblies, given quality attributes of the components. Moreover, a method is presented that can be used to build prediction enabled component technologies and its validation procedure is described. The method is demonstrated through discussion of two different attributes: latency and consistency. It is certain that not all types of attributes are suitable for prediction and the thesis discusses classification of different attributes from a prediction perspective.

If we can make predictions about quality attributes, does that imply that we can make critical design decisions based on the values obtained? Certainly, if there is no objective trust in the predicted knowledge, it will be hard to use the predictions in building safety critical systems. If it is known that prediction is not 100% accurate then it is important to make users aware about that uncertainty and that this uncertainty should be taken into consideration when designing and delivering the system. The question how to provide objective trust in component and assembly property predictions is also addressed in more detail in the thesis.
“Prediction is very difficult, especially about the future”.
Niels Bohr, 1885-1962
To
Christina
Emmy, Ida, Jacob and Thea
Acknowledgements

TBW
Published Material

I have authored or co-authored the following publications:

<table>
<thead>
<tr>
<th>Books</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Building Reliable Component-Based Software Systems</strong></td>
</tr>
<tr>
<td>Editors: Ivica Crnkovic, Magnus Larsson</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Journal Articles and Book Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Concerning Predictability in Dependable Component-based Systems: Classification of Quality Attributes</strong></td>
</tr>
<tr>
<td>Submitted to special issue on Architecting Dependable Systems II, Lecture Notes on Computer Science, December 2003, Springer Verlag</td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Challenges of Component-based Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISBN 0-7923-6823-1</td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson, Kung-Kiu Lau, Juliana K. Küster Filipe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thesis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Applying Configuration Management Techniques to Component-based Systems</strong></td>
</tr>
<tr>
<td>Licentiate Thesis</td>
</tr>
<tr>
<td>MRTC, December 2000</td>
</tr>
<tr>
<td>Author: Magnus Larsson</td>
</tr>
<tr>
<td>Conferences and workshops</td>
</tr>
<tr>
<td>---------------------------</td>
</tr>
</tbody>
</table>
| **Towards an Impact Analysis for Component Based Real-Time Product Line Architectures**  
Authors: Anders Wall, Magnus Larsson, Christer Norström |
| **Using Prediction Enabled Technologies for Embedded Product Line Architectures**  
In ICSE workshop on 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 2001 proceedings Opatija, Croatia, May 2001  
Authors: Ivica Crnkovic, Magnus Larsson |
| **Managing Complex Systems - Challenges for PDM and SCM**  
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**  
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**  
Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders |
| **The Different Aspects of Component Based Software Engineering**  
Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders |
<table>
<thead>
<tr>
<th>Conferences and workshops</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A Case Study: Demands on Component-based Development</strong></td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson</td>
</tr>
<tr>
<td><strong>Development Experiences of a Component-based System</strong></td>
</tr>
<tr>
<td>Authors: Magnus Larsson, Ivica Crnkovic</td>
</tr>
<tr>
<td><strong>Object-Oriented Design Frameworks: Formal Specification and Some Implementation Issues</strong></td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Juliana K. Küster Filipe, Magnus Larsson, Kung-Kiu Lau</td>
</tr>
<tr>
<td><strong>State of the Practice: Component-based Software Engineering Course</strong></td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson, Frank Lüders</td>
</tr>
<tr>
<td><strong>Component Configuration Management for Frameworks</strong></td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson, Kung-Kiu Lau</td>
</tr>
<tr>
<td><strong>Processing Requirements by Software Configuration Management</strong></td>
</tr>
<tr>
<td>In Euromicro 99, proceedings of the 25th EUROMICRO conference Milano, Italy, September 1999, IEEE Computer Society</td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Peter Funk, Magnus Larsson</td>
</tr>
<tr>
<td><strong>New Challenges for Configuration Management</strong></td>
</tr>
<tr>
<td>In System Configuration Management, SCM-9, proceedings Toulouse, France, August 1999, Springer Verlag</td>
</tr>
<tr>
<td>Authors: Magnus Larsson, Ivica Crnkovic</td>
</tr>
<tr>
<td><strong>Managing Standard Components in Large Software Systems</strong></td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson</td>
</tr>
<tr>
<td>Technical reports</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td><strong>Predictable Assembly of Substation Automation Systems: An Experiment Report</strong></td>
</tr>
<tr>
<td>Authors: Magnus Larsson, Kurt Wallnau, Scott Hissam, John Hudak, James Ivers, Mark Klein, Gabriel Moreno, Linda Northrop, Daniel Plakosh, Judith Stafford, William Wood</td>
</tr>
<tr>
<td><strong>PDM and SCM: Similarities and differences</strong></td>
</tr>
<tr>
<td>Technical Report, Sveriges Verkstadsindustrier, January 2002</td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Annita Persson Dahlqvist, Ulf Asklund, Magnus Larsson, Daniel Svensson</td>
</tr>
<tr>
<td><strong>Component-Based Development - a New Approach in Software Development,</strong></td>
</tr>
<tr>
<td>Technical Report, May 2001</td>
</tr>
<tr>
<td>Authors: Ivica Crnkovic, Magnus Larsson</td>
</tr>
<tr>
<td><strong>Component Based Software Engineering - State of the Art</strong></td>
</tr>
<tr>
<td>Technical Report, Internal, January 2000</td>
</tr>
<tr>
<td>Authors: Magnus Larsson, Ivica Crnkovic</td>
</tr>
</tbody>
</table>
Contents

1 INTRODUCTION ........................................................................................................... 1
  1.1 KEY RESEARCH QUESTIONS: THE PROPERTY PREDICTION PROBLEM ........... 7
  1.2 DELIMITATION SCOPE AND KEY ASSUMPTIONS ......................................... 10
  1.3 METHOD ..................................................................................................................... 11
  1.4 RELATED WORK ......................................................................................................... 13
  1.5 MAIN CONTRIBUTION ............................................................................................... 15
  1.6 ORGANIZATION ......................................................................................................... 25

2 COMPONENT-BASED SOFTWARE ENGINEERING .......... 26
  2.1 SCOPE AND GOAL OF CHAPTER ................................................................. 27
  2.2 CHALLENGES OF COMPONENT-BASED SOFTWARE ENGINEERING .... 28
  2.3 DIFFERENT REUSE CHALLENGES USING COMPONENTS .................... 31
    2.3.1 System Evolution
    2.3.2 Development of Reusable Components
    2.3.3 Maintaining Reusable Components
  2.4 COMPONENTS IN DEPENDABLE SYSTEMS ...................................................... 39
  2.5 COMPONENTS AND ASSEMBLIES ................................................................. 42
  2.6 COMPONENT MODELS IN THE LIGHT OF PREDICTION ............................ 47
    2.6.1 Prediction in .NET
    2.6.2 Prediction in Koala
    2.6.3 Prediction in Rubus
  2.7 SUMMARY AND CONCLUSION ............................................................................. 57
3 PREDICTABLE ASSEMBLY OF COMPONENTS ...............59

3.1 SCOPE AND GOAL OF CHAPTER ......................................................60
3.2 PREDICTABLE ASSEMBLY.................................................................60

3.2.1 Description of Accuracy of Property Prediction Theories

3.3 PREDICTION ENABLED COMPONENT TECHNOLOGY .......................69
3.4 BUILDING A PECT.................................................................73

3.4.1 Definition Phase
3.4.2 Implementation and co-refinement phase
3.4.3 Validation phase
3.4.4 Packaging phase

3.5 EMPirical VALIDATION........................................................................87

3.5.1 Define Validation Goal
3.5.2 Define Validation Process
3.5.3 Develop Measurement Apparatus
3.5.4 Collect Sample Data
3.5.5 Analyze Results

3.6 SUMMARY AND CONCLUSION ..................................................100

4 A LATENCY EXPERIMENT OF PECT ........................................101

4.1 SCOPE AND GOAL OF CHAPTER .............................................102
4.2 THE OPERATOR PECT..............................................................102

4.2.1 The Goal of the study
4.2.2 Description of Analytic study
4.2.3 The test environment
4.2.4 Sampling procedure
4.2.5 Select sample
4.2.6 Perform analyses

4.3 CONTROLLER PECT.........................................................................118

4.3.1 Sample selection and procedure
1 Introduction

We are witnessing an enormous expansion in the use of software in business, industry, administration, research, and not least in everyday life. Software is no longer marginal in technical systems but has become a central factor in many fields. Features based on software functionality, rather than other system characteristics, are becoming the most important factor in a competitive market. This trend increases the demands on software products such as enhanced usability, robustness, reliability, flexibility, adaptability, and simpler installation and deployment. At the same time as these demands are growing stronger, the complexity of processes that software manages is increasing as well as the demand for the integration of processes from different areas. Consequently, software is becoming increasingly large and complex. Complex entities such as a software system consists of many software programs or applications that have to cooperate flawless. A software system might be part of a whole system including both software and hardware.

The main challenge for software developers today is the ability to cope with complexity and to adapt quickly to changes. Traditionally, software development addressed the challenges of increasing complexity and dependence on external software by focusing on one system at a time and on satisfying delivery deadline and budget requirements without considering the evolutionary needs of the system.

Focusing on one application at a time and neglecting the forthcoming changes during the development, has led to a number of problems: the failure of the majority of projects to meet their deadline, budget, and quality requirements as well as the continued increase in the costs associated with software maintenance.
1. Introduction

One possible key to the solution of these problems is reusability. The idea of software reuse is not new. However, despite some successes, reusability has not become a driving force in software development. Many of the unsuccessful approaches to reuse did not satisfy the basic requirements of reusability [6]:

i) Reuse often requires some modification of the object being reused,

ii) Reuse must be integrated into specific software development.

The reuse concept can be used on different levels: On a low level, it is a reuse of source-code, and small-size components. More efficient reuse is obtained with larger components which encapsulate business functions. Finally, the integration of complete products in complex software systems is the highest level of reuse. On each level of reuse, there are specific demands on the reusable components, on the component management and on the integration process. A software component is defined a component model and there are an arbitrary number of component definitions made. Chapter 2 presents the definition of a software component used in this thesis.

In many approaches, reusability is not existing in the development process, what can be reused and what cannot be reused is not precisely defined, and how the changes can be introduced in the reusable parts is not formalized. The new rapidly emerging approach Component-based Development (CBD) re-establishes the idea of reuse and introduces new elements: In CBD, software applications are built by assembling components already developed and prepared for integration. CBD has many advantages including effective management of complexity, reduced time to market, increased productivity, improved quality, a greater degree of consistency, and a wider range of usability [15].

However, there are several disadvantages and risks in using CBD which can jeopardize its success.
• **Time and effort required for development of components** - Among the factors which can discourage the development of reusable components is the increased time and effort required to build reusable units [22,98].

• **Unclear and ambiguous requirement** - In general, requirements management is an important and complex phase in the development process, its main objective being to define consistent and complete component requirements. One of the major problems of software development in general comes from unclear, ambiguous, incomplete and insufficient requirements specifications. Reusable components are, by definition, to be used in different applications, some of which may yet be unknown and the requirements of which cannot be predicted. This applies to both functional and non-functional requirements. This makes it more difficult to identify the requirements properly and hence to design and build components successfully [61,69].

• **Conflict between usability and reusability** - To be widely reusable, a component must be sufficiently general, scalable and adaptable and therefore more complex (and thus more complicated to use), and more demanding of computing resources (and thus more expensive to use). A requirement for reusability may lead to another development approach, for example a design on a more abstract level, which may lose flexibility and finer tuning, but achieves greater simplicity [22,98].

• **Component maintenance cost** - While application maintenance costs can be lowered, component maintenance costs can be very high since the component must respond to the different requirements of different applications running in different environments, with different reliability requirements and perhaps requiring a different level of maintenance support [22].

• **Sensitivity to system evolution** - 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 the updating of operating system, the updating of other components, changes in the application), there is a risk that the change introduced will cause system failure [63].

- **Unknown system behavior** - Components are in general treated as black boxes with no or little information easily accessible. The information needed from the software components is about how they behave and their quality attributes. It is possible to get the syntactic information but hard to get access to the semantics. If the components behavior are not known it is even harder to deduce anything about the application behavior.

There are two possible ways to see the benefits of the component-based approach. The first is achieving advantages by reusing components and the second is the power of having well-defined software architecture interfaces, which can sustain changes by replacing components when needed. A component-based modular architecture is a means to use or reuse software architecture over time. However, there is a challenge to have a component technology that deals with reuse and exchange of components.

A problem that occurs is that even if the interfaces comply with a particular standard (a component model) a component-based system may fail due to mismatch of component and application behavior. This shows that it is not sufficient to secure the functional demand of an application but also that its behavior is important.

It is possible to secure the behavior and the functionality of an application through extensive testing but this is generally an expensive approach not very efficient. Much better approach is to predict behavior of the application that will be integrated from the components which build up the application.
The behavior is usually expressed in form of quality attributes. A quality attribute is a characteristic of software. For instance, satisfy security, safety, performance and dependability are attributes that are found in safety-critical systems.

Quality attributes are often called properties or non-functional or extra-functional attributes because they describe something about the quality of the component and not explicitly about the component functionality [4]. As the quality attributes describe the characteristics the property of a component is the concrete accessible value that represents the characteristics. A component can express quality attributes through its properties. Not all quality attributes can be expressed as properties at the component level but rather at the application level. In the following text, we use the term component property and quality attribute, but sometimes non- and extra functional attributes are used.

An analogy with the component properties is the specification sheet that comes with any electrical apparatus; the specification sheet describes for instance the electrical power consumption of the apparatus. The electrical power consumption is a property specified in the sheet. An example of a similar property for a software component is its memory consumption.

From a functional perspective, there might be no need to know the quality attributes, but by having visible properties it is possible to gain other values, such as trust and understanding in the software component. Having visible specification sheets with properties of the components for the user will put new demands on trust and certification. If design decisions are to be made using expressed properties of a component it is important that the properties actually fulfill what they claim.

In general, it is hard to know or achieve desired information about a component or a software system. The information could be packaged as properties of the components or as separate documents describing the properties. These kinds of properties or attributes
1. Introduction

also appear on the software system level but they are harder to be automatically derived from the component properties because of the complex interactions of components.

*Predictability* in a component-based approach is the ability to reason about application behavior knowing the quality attributes of the components and the components interconnections.

An inability of predicting the application behavior from the components’ properties is one of serious problems of component-based approach. This is in particular valid in domains in which quality attributes are of large importance (for example in safety or business critical systems)

Predictability influences how applications with higher quality can be built. Prediction can be used to reduce the cost of testing a system by allowing faults to be removed earlier in the development life cycle.

A weakness is that it is hard to make prophecies without any support; however, support can be obtained by building the prediction capabilities into the software systems. This is very much the same problem as with quality of software in general. Often an approach is taken, that it is possible to test a system after it is built to achieve the right quality; this is not true since the quality has to be designed into the system from the start and then tested. Testing is also very expensive and it is extremely hard to simulate the targeted end customer environment in which the program will execute. Often, it is the case that the software vendors have only recommendations on how the target configuration should look like.

The vendor is trying to make sure that their system will run on as many configurations as possible by testing on various configurations. However, the vendor can only provide recommendations to the end customer on what configurations of components are needed to run the particular product. The actual control of the customer configuration is in the hands of the customer himself and it often happen that the customer makes intended or
unintended changes to the installed configuration. Since the vendor cannot control the target system, it is hard to test a system fully before deployment. The systems and products must tolerate different environments or at least have predictable behaviors.

If a component-based approach could solve the prediction problem of application behavior from the component properties then this would be a strong argument for having components.

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 prediction of the final behavior of the software. There is a constant challenge to predict the cost, quality and the development time of a software product. When it comes to reality, it is very rare to have new green field development of products, instead there is often legacy software that has to be considered and taken into account during development.

A major problem with predictability is the need for measuring methods for the desired properties. If measurements of the component properties cannot be obtained then it is hard to reason about them. For many properties, it is necessary to measure other properties and deduce the value of the desired ones or just to estimate the value. In any case, it is important to have hard data on the value of a certain property that we want to reason about.

The rest of this chapter gives a description of the research questions addressed in this thesis and what method is used to answers those. Direct related work to the research questions is listed in Chapter 1.4. The main contribution of this thesis is made explicit in Chapter 1.5 and an overview of the whole thesis is presented.

1.1 Key Research Questions: The Property Prediction Problem

For component-based systems, there are many challenges as pointed out in the earlier section. A crucial challenge in relation to quality attributes is the problem to gain
1. Introduction

predictability of the software quality a priori its assembly and use. Given the system quality attributes, which properties of involved components are required? If there are reasoning techniques for a particular quality attribute, it is of importance that those theories are used to a wide extent to prevent reinvention. Levering on existing technologies gives a tool to focus on solving the problem at hand.

Another problem is to predict quality attributes in a software system, and to know what accuracy there is in the prediction of the quality attributes.

These and similar questions have been addressed at a series of CBSE workshops [25,95], and particular models of certain properties have been analyzed [40,94], but so far very little work has been done in the systematization and classification of quality attributes in accordance with the questions 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 lead to the main research question for this thesis:

*Can quality attributes be predicted in a component-based system 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 attributes, the cost might be too high. In order to answer to this question several sub questions should be considered to be able to predict properties of assemblies.

1. *Is it possible to develop a component technology that will support reasoning about system properties from component properties?*

2. *How can we verify the predictions to gain objective trust?*

3. *What properties or quality attributes are suitable for prediction?*
4. How can components provide information about their properties?

One possible approach to gain predictability is to take or design a component model that supports prediction. If it is possible to build or extend existing component models to support prediction then the question how to build prediction enabled component models or technologies follows.

The goal is not to build component technologies in general but rather to design 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 3, as a possible answer to this question.

What kinds of component properties and technologies are suitable for prediction and how can components provide information about their properties? Depending on the actual component, it might be possible to determine that prediction of certain attributes is not suitable. This question relates very much to the definition of what a component is, and different component definitions might or might not be suitable for prediction.

What if the definition of a component or the component model can guide if it is possible to introduce prediction or not? If the component can be determined not to suit prediction then no more effort is needed trying to introduce prediction for this case. In addition, the quality attribute itself determines the means of predictability; by their nature, certain quality attributes are easy to predict. Some, for instance complex attributes, are very difficult or even impossible to predict. A discussion concerning these questions can be found in Chapters 2 and 6.

To obtain predictability of a component-based system we must ensure several factors: (i) a component technology that provides means for specification of component properties, (ii) a property theory that identifies and specified the property, and provides a means for its calculation, and (iii) an interoperation mechanism which makes it possible to translate the component specification to the property theory. Using these elements, we
1. Introduction

can develop predictability of certain properties. If a prediction is developed, how can we verify that the prediction is valid?

To have the power to measure the desired attributes gives us the capability to verify predictions, but does this mean that all non-measurable properties are not suitable for prediction? A whole set of questions around the ability to detect, estimate or measure properties arises and are addressed in discussion and detail in chapters 3, 3.6 and 5.

1.2 Delimitation Scope and Key Assumptions

The questions that have been outlined in the previous section are addressed in this thesis in several sections. Each section addresses the questions and answers either by experiment or by discussion.

This thesis does not try to reason about assembly properties in a formal way, but rather in an empirical manner. Formal approaches complement software engineering but is not covered in the scope of this thesis.

Instead of taking the more formal approach, this thesis concentrates on software engineering. A result that this thesis is aiming for is to find best practices how to approach prediction of quality attributes. By having a more software engineering oriented approach we try to target for practicality, even as we shall see this is not easily achieved. A target is to know more about how to approach the problem of designing predictable components.

The area of knowing certain attributes on a system, software product or component is very complex and many methods can be used. We assume that the component model and technology is the basic parts where predictability needs to be introduced. The component model will be in focus for this thesis although we exemplify the theories with a component technology.
The other views on prediction, such as, prediction of success, development time, usability and cost of development are not covered largely in this thesis.

1.3 Method

A method used for conducting the research in this thesis is to take a real world practical problem and to solve it by transforming the problem into an idealized problem in a research setting. A research setting allows the researchers to work separate from product development and that relieves the researchers of the typical delivery line focus that is often present in an industry setting. Work in the research setting with the problem results in a research product that solves the idealized problem.

Taking a problem from a practical setting into a research setting can be facilitated if several case studies are performed. However, even as a practical problem can be idealized in a research setting, it does not guarantee that the solution is of any practical value, it is important to take the idealized solution and apply it as a practical solution. However, this is not easy without strong commitment from industry and willingness to spend time and money to apply the results. Our experiments are somewhat idealized since we do not operate on a real target system.

Figure 1: A method for doing research that target real world problems [90]

There are three phases defined by Shaw when doing research, question, strategy/result and validation [90]. The phases contain several building blocks that serve as guideline to
1. Introduction

what approach and what method is used for doing software engineering research. Definition of the research question can be of five types, feasibility, characterization, methods, generalization or selection. As strategy and type of result, the following five types can be used, qualitative model, technique, system, empirical model, or analytic model. Validation can be performed with, persuasion, implementation, evaluation, analysis or experience.

Each research question is approached different depending on the type of the question.

The main research question is to show feasibility and the strategy is to implement a system with the capability to predict a quality attribute. Validation of the research result is done by evaluating the implementation and drawing experiences.

Building the two component technologies demonstrates that it is possible to build automatic reasoning of system properties and this address the first research question. The first question is similar to the main question a feasibility question that is implemented as a system and validated with evaluation.

The second question is to build a method that can be used to verify predictions and this method is used to validate the results from the experiments. The research method is validated during the application of the method when building the component technologies.

The third question that characterizes attributes suitable for prediction is addressed by experimenting with measurable attributes such as latency and memory consumption. The research result is a quality model validated with evaluation.

During the implementation of the system, we show and try out different ways for components to describe their properties, thus addressing the forth research question.

All research results are evaluated and experiences are collected.
Creating experiments and drawing empirical conclusions have certain traps that ought to be avoided [32]. The conclusions must be based on empirical validation and not on intuition and advocacy. The experiment advocates a solution but it does not prove that the solution is generally correct. The experiments must have a good experimental design that allows others to reproduce the same experiment and that conclusions drawn are objective. Is the experiment a toy situation or a real world situation?

The experiments described in this thesis, target an idealized problem in a research setting of a real world practical problem. 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 pointed out strongly in [102,103]

Several research experiments in the area of component-based software engineering have been carried out to support the validity of the research results. These experiments address the research questions in a direct or indirect way. Some of these experiments contribute to the area of prediction while others have contributed more in general to the CBSE area and the area of product data management and software configuration management.

To strengthen the discussion about prediction we joined a research project about predictable assembly from certifiable components with the objective to study how prediction can be applied to an industrial problem. The experiment how to create a prediction enabled component technology was conducted at the Carnegie Mellon University/Software Engineering Institute. This project created two technologies with prediction capabilities that where validated empirically to certain extent and they are presented partly in this thesis and in full in [42].

1.4 Related work

This section outlines selected related work to the research questions. Additional related work is found in general in each chapter and specifically in chapter 2.
There are several research projects concerning the question about building a component technology that has predictability capabilities. The most prominent related work is the one conducted by SEI in the area of predictable assembly from certifiable components [19,40,42,44,54,74,112]. The work in this thesis is to a large extent based on this work.

In addition, the work from [84] is targeting how component technologies can be built or used to support prediction. This work is promising and strives to solve real world practical problems through intense industry collaborations.

The second research question how to verify and validate the predictions made with any theory to gain objective trust is addressed in the work by Moreno [74]. Statistical methods, which are used in this thesis, are presented and tried out in a research experimental setting.

The work by Wohlin [117] and Voas [106] outlines models and methods how to certify software components and certain of these ideas can be taken into account when building a certification framework for prediction theories or prediction enabled component technologies.

Properties suitable for prediction are addressed for certain specific attributes in the area of predictability. Reliability for component-based software architecture is addressed in [84,86,87]. By using parameterized contractual specifications based on state machines. The research is exemplified with an e-commerce example and a report from experiments in a reliability test bench. Reliability strongly depends 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 include the work done by Stafford and McGregor [94]. This work applies software reliability theories to component-based software.
1. Introduction

Scalability and performance prediction for component-based systems is addressed in [36,118] where an empirical investigation has been conducted on several COTS middleware products. This research presents scalability metrics depending on the performance of the system.

Memory consumption and the suitability for prediction is presented in [29]. This research uses the Koala component model [105] for experimenting with prediction of the memory demand from component compositions. Static memory evaluation techniques are used and a method is proposed that allows estimating the memory consumption.

Deadlock detection between components and the reasoning how to accomplish deadlock freeness can be found in the work [51]. This work presents a technique that allows connectors synthesis to prevent deadlock between components in a COM/DCOM [73] setting.

Work that addresses quality attributes at a more general level serves as a base for reasoning about suitability for prediction. For this there are several classifications, McCall proposes quality factors in [72], the ISO 9126 [53] standard lists certain quality attributes. In addition Bertoa analyses quality attributes for COTS components [8].

There are several approaches addressing the question of how to obtain the properties of components and get the property of their assembly [4,62,82]. These works are mostly focusing on the quality attributes as such and not on the quality attributes of software components and how to reason about assemblies of such components.

1.5 Main Contribution

This thesis delivers a structured approach how prediction of quality attributes, such as, latency or memory consumption, can be achieved. The answer to the main research question is, yes, it is possible to achieve predictions of attributes when components are assembled together. For instance, it is show 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 behavior.

There is however not a single simple “yes” answer for this research question, since prediction might mean imposing too many restrictions on a component technology to make it functionally useful.

A component property can be of many different types and classes and there are several quality attributes not suitable for prediction because of different causes. One example is scalability where it is very hard to say that a component has certain scalability. It is not until the components are put together, we know the scalability of the system or the assembly. An analysis of these harder to predict properties have been done and presented in this thesis.

When we know what properties we can predict and that it is possible to build prediction into a component technology, we would like to know if there is a method to accomplish this. From the SEI work, we have extracted a method that guides the prediction enabled component technology builder to reach a ready-made framework for prediction. This method is presented in Chapter 3.

It is of course of vast importance that we can get access to the component properties. The whole discussion about where and how the value of the property is stored is a result of this thesis. There are several options but there are two main approaches and the first one is that the component themselves store the value and the second is that there is a media outside the component that store and describe the property. Accessing properties can be done in several ways and the outline of analytic interfaces and property specifications is a contribution.

As the major contribution is the description of a method how a prediction enabled component technology is to be designed and developed, we need to know also how to validate such a technology. Together with the development method there is a
1. Introduction

presentation, in Chapter Error! Reference source not found., of how such a component technology and its interpretation for property theory is to be empirically validated.

Apart from the main contribution each chapter includes additional contributions and they are highlighted as follows:

• Chapter 2:
  - Identification of reuse challenges based on a case-study performed in an industrial setting.
  - Exemplification of a component model with capabilities for predicting consistency between components.

• Chapter 3:
  - Development of a method used to build prediction enabled component technologies (PECT)
  - Development of a validation method for a PECT

• Chapter 4:
  - Implementation of a PECT according to the defined method for a particular quality attributes.
  - Validation of the implemented PECT using the defined method.

• Chapter 5:
  - Exemplification of how to achieve predictability for another quality attribute.

• Chapter 6:
  - Classification of quality attributes with respect to prediction.

• Chapter 7:
  - An analysis of the suitability of predictability in industrial settings.

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

- Building Reliable Component-Based Software Systems


Editors Ivica Crnkovic and Magnus Larsson

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.

**Contribution:** Magnus Larsson contributes to this book in capacity of coeditor with Ivica Crnkovic. As editor for the book, Magnus defined included chapters and ordered them written by various scientists in the particular areas. Certain chapters are coauthored by Magnus who also reviewed and quality assured the other chapters. Chapter 2 in this thesis is based on the book and on those parts where Magnus contributed most.
1. Introduction

- **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

  **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.

  **Contribution:** Magnus Larsson took active part in the SEI PACC team producing 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. Magnus Larsson also took part in designing and building the validation infrastructure for the controller PECT. Chapters 3 and 3.6 are heavily based on this work.
• Concerning Predictability in Dependable Component-based Systems:
  Classification of Quality Attributes

  Submitted to special issue on Architecting Dependable Systems II, Lecture Notes on
  Computer Science, December 2003, Springer Verlag
  Authors: Ivica Crnkovic and Magnus Larsson

  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 properties 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 properties of the components and according 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.

  Contribution: Magnus Larsson co-authored this article together with Ivica Crnkovic and was responsible for the classification and assembly part of the article. Chapter 6 is more or less the original version of this paper.
1. Introduction

- **Challenges of Component-based Development**
  Presents experiences and background to 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

**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.

**Contribution:** Magnus Larsson describes the ABB case study and conclusions about the different reuse challenges in component based software engineering. Chapter 2 is partly based on this article.
Towards an Impact Analysis for Component Based Real-Time Product Line Architectures

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

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.

Contribution: Magnus Larsson applied a consistency property theory to the presented component model and provided underlying reasoning about how to achieve predictability in product line architectures for embedded systems. The results from this article about component dependencies and consistencies serve as a base for chapter 5.
1. Introduction

- 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

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 run-time 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.

**Contribution:** Magnus Larsson wrote and presented this thesis to achieve a licentiate degree in computer science. This work is based on several peer-reviewed papers. Ideas from this thesis serve as base for chapter 5.

### 1.6 Organization

Each chapter targets the research questions with support of state of the art where applicable. This thesis is organized in eight chapters of which the introduction part is covered in Chapter 1. Chapter 2 outlines the challenges of component-based software engineering and discusses three component models with respect to predictability. The underlying work of how to build a prediction enabled component technology is described in Chapter 3 while an example of a technology is provided in Chapter 3.6. Another quality attribute used to analyze the impact of installing or changing a component in a system is described in Chapter 5. Chapter 6 gives a classification of different quality attributes and their suitability for prediction. Ethical aspects of basing design decision on predicted values are discussed in Chapter 7 with a focus on catastrophic consequences of using wrongfully predicted data. The thesis is concluded in Chapter 7.6 with our own analyze, future work and lessons learnt.
2 Component-Based Software Engineering

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 [22,66]. 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 [23,38]:

- 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 system behavior. 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 [56,99]. 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 discusses 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 support the understanding of the practical problems faced in dealing with CBSE. The identified challenges are the results of performed case-studies and literature search.

An additional goal is to narrow the wider scope of CBSE down to more concrete definitions of components and assemblies of components. This is to provide a base for Chapter 3 and onwards. Moreover, this chapter looks at three different component models and pinpoints strengths and weaknesses from a prediction perspective.
2.2 Challenges of Component-Based Software Engineering

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 [78] 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 [23]; some of these are summarized here

Component specification - Although this problem has been addressed from 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 [16,68,89,91].
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 [13,30].

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 [38,59]. 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 [75].
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?” [114].

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 certificate 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 [76,106,109].

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 [21,64]. 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, run-time 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 [98]. 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 relative 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, part 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. 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 use of use of standard components implies less control on them [18,64,65], especially if the components are updated at run-
time. 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 [15].

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):

\[
R_C = R_{C0} + \sum a_i R_{pi} \quad 0 \leq a_i \leq 1
\]  

(1)

\(R_{C0}\) denotes direct requirements of the component, \(R_{pi}\) requirements of the products \(P_i\), \(a_i\) impact factors 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 run-time 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.](image)

The development time for these components was 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 became more stable and required less effort for new changes. In that period, the frequency of the releases has been lowered, and especially the effort has been significantly lower.

New efforts for further development of components appeared with migration of products on different platforms and newer platforms versions. Although the functions of the products and components did not changed significantly, a considerable amount of work was 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 [20,79]. 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](image)

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 were 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” [48]. 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 availability 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 [93].

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 behavior 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 run-time environment. The second question is how to predict the reliability of a system from the known reliabilities of its components?

Thane [101] 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 run-time 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 [9,107,108]. 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 [73], 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.

Szyperski [98] 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, 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 [27] 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 [2] explains this difference as follows: Firstly, components can be composed at run-time 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 [50], some of them developed.
internally by companies (for example Koala,[105], AspectObjects [37,80], or Object Modeller [31]). 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 [22].

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 run-time 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 exists and 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 run time 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.

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 in the Light of Prediction

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 (Figure 4).

![Component Model Diagram](image)

**Figure 4:** A component technology implements a component model

There are several component technologies and we will discuss if .NET [100] and Koala [105] are suitable as a base for making predictions.

In order to enable analyze 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 a notion of an 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 to many definitions.

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

The .NET component technology called the .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 downside from the predictability perspective is that the notion of assembly of components is missing and there are only applications using components that are denoted .NET assemblies.
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 [105]. 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 provides and requires 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 are called configurations and consists of a list of components and a list of connections between components. A configuration of components is shown in Figure 5 where also certain binding, gluing and switching components are shown.
Neither components nor configurations (assemblies) are binary and deployed as such. Components are source code with explicit interfaces and configurations are the blueprint 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 analysing the components and their interconnections [29,33]. 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 downside 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 Rubus

This section describes a component model that has version information built into it. The model is presented in detail in [110] and is developed as part of the RUBUS component technology. 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 [111].

In Figure 6, the example component meta-model is depicted in UML-fashion. Components have in and out ports, which resembles 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.
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 emphasize on the real-time properties. A formal definition of the constructive part of the component model depicted in Figure 6 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, $C$ 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 component’s 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.

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 a pair of tasks \( \langle \tau_i, \tau_j \rangle \in T \times T \), such that if \( \tau_i \rightarrow \tau_j \), then \( \tau_j \) may start its execution earliest at the end of \( \tau_i \)'s execution and \( i \neq j \).

A mutual exclusion relation, \( \otimes \), is a binary, symmetric relation among pair of tasks \( \langle \tau_i, \tau_j \rangle \in T \times T \), such that if \( \tau_i \otimes \tau_j \), then neither \( \tau_i \) nor \( \tau_j \) is permitted to execute while the corresponding party, or a transitively related party is executing and \( i \neq j \).

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 pair of ports on components, \( \langle c_i.i_x, c_j.o_y \rangle \in C.I \times C.O \), such that if \( c_i.i_x = c_j.o_x \), then \( c_j \)'s in port \( i_x \) is connected to \( c_j \)'s out port \( o_x \).
Figure 7: Four components with precedence, connections and version dependencies specified using constraints

Figure 7 shows an example where four components have been instantiated from the model presented in Figure 6. 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 dataflow 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.

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. This model
will be used in chapter 5 to illustrate reasoning about consistency between component
versions and variants in an assembly.

2.7 Summary and Conclusion

In this chapter we identify and present 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 Rubus 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 5.
2. Component-Based Software Engineering
3 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 behavior. One of the main challenges in constructing and maintaining a software architecture 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 [ref...] to extend the component specification.

If component can specify their quality attributes as properties, it gives a possibility to use that information to reason about assemblies of the 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 implantation.
3. Predictable Assembly of Components

Section 3.1 sets the scope and goal of this chapter. Predictable assembly is described in section 3.2 and the more concrete implementation a prediction enabled component technology is presented in section 3.3. The method how to build a component technology supporting prediction is given in section 3.4 and the validation procedure of such a technology in section Error! Reference source not found.. The chapter is concluded in section 3.6.

3.1 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 [42,45]. The SEI has developed the concepts of predictable assembly from certifiable components and prediction enabled component technologies. We have extended these concepts by development of a method used to build and validate prediction enabled component technologies. Our contribution with the methods is found in sections 3.4 and 3.5.

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.

3.2 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. 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. Each part in this approach is very much intervened with the other parts of software. With this approach, the application will probably work and the desired quality attributes will be achievable but when a later change occurs, it will be a tiresome exercise 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 possible 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 [88]. 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 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 [41,43,45,112].

To be able to predict the application properties from the component properties, SEI defines both a construct 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 constructive model deals with operational (functional) properties, the analytical model is used to define and reason about quality attributes.
3. Predictable Assembly of Components

Figure 8: 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 8). The constructive component model does not have to be consistent with the analytical model. It is sufficient to have the possibility transform a concrete assembly into a 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, it is desired to integrate the analytic technology with the component technology to achieve maximum degree of automation. A prediction enabled component technology (PECT) is an example of how such integration of analytic and component technologies can be done.

The fundamental principle of predictability is the ability to predict properties of components form 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.
3. Predictable Assembly of Components

To achieve predictable assembly a framework for reasoning about quality attributes is needed. Figure 9 display 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 limits on what we can reason about 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 has to be “ordered” from the technology vendor.
In a similar manner, the information that the commercial components can provide is limited and hence the level of flexibility is low. For example, if reasoning about the memory consumption is wanted 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 extendable 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 is measured the statistical distribution and confidence level of the measure is of interest for the user of the value. 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 3.5 and in the work from Moreno et al. [74].
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.

### 3.2.1 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 component 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 fall 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 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
3. Predictable Assembly of Components

this process. The goal is to determine the accuracy of the predictions made with the theory of a certain quality attribute.

Component properties can be calculated or measured. 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.

For measured 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 MRE is not applicable for quality attributes that are not representative with measurable properties.

\[
MRE = \frac{P' - P}{P}
\]  

(7)

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 is 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 10.
3. Predictable Assembly of Components

**Figure 10:** Example of a Statistical label for a component technology (With courtesy of Scott Hissam)

Figure 10 shows how a label can look like for an example component technology and the resemblance with a nutrition label on a Coca Cola can is noteworthy. Each component and component technology can express their quality attributes as properties with statistical labels if objective trust is desirable. Figure 10 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 2,952 jobs. The tasks and jobs are specific terms for this example component technology and is explained more in a Chapter 4 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%, 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 [41].

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.

### 3.3 Prediction Enabled Component Technology

Prediction Enabled Component Technology (PECT) 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 [40,41,43-45,112]. 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 11.
3. Predictable Assembly of Components

The constructive parts of a component technology are used to build and satisfy functional requirements. Components have to comply with the restrictions and rules set by the component technology. As the analytic part also is included in the 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 is given in Figure 22. Another example on how to express and describe assemblies is to use a composition language [54].

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 (Figure 12).

![Figure 12: Components are deployed into a component technology for later use](image)

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. One example is a component that shows a gauge metering something, this 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. And 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 (Figure 13). 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 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 behavior of the application. With the PECT concept, it is required that the components and their connections are explicitly described.

### 3.4 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.
3. Predictable Assembly of Components

We define the design of a PECT in four fundamental phases, they are definition, co-refinement, validation and package, as illustrated in Figure 14.

**Figure 14: An overview of the PECT method using RUP notation [60]**

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.
3. Predictable Assembly of Components

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 assemble**
  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) [60], 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.
3. Predictable Assembly of Components

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 Component 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 System 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.

3.4.1 Definition Phase

The definition phase has two parallel paths, 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 [92]
3. Predictable Assembly of Components

Figure 15: The definition phase with its two tracks

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.

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 10 at page 68. Table 1 summarizes the activities, artifacts and workers for the definition phase.
### 3. Predictable Assembly of Components

#### Table 1: Definition phase activities, artifacts and workers

<table>
<thead>
<tr>
<th>Activity</th>
<th>Artifact</th>
<th>Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define functional requirements</td>
<td>PECT requirements</td>
<td>Customer</td>
</tr>
<tr>
<td>Define assembly property type</td>
<td>Property definition</td>
<td>Customer</td>
</tr>
<tr>
<td>Define normative statistical goal</td>
<td>Statistical goal</td>
<td>Customer</td>
</tr>
</tbody>
</table>

#### 3.4.2 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 [41].

During the implementation the most important part is the creation of the component model, analysis model and measurement framework. As shown in Figure 14, 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.
3. Predictable Assembly of Components

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 [32].

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 16: The integration of the different models

Figure 16 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.
3. Predictable Assembly of Components

I.e. the analysis tool work on concrete components while the predictor tool work 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 14 the constructive model or PECT instance is validated both empirically and functionally after the integration step. The empirical validation is covered in section 3.5 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 2: Co-refinement and implementation phase activities, artifacts and workers

<table>
<thead>
<tr>
<th>Activity</th>
<th>Artifact</th>
<th>Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create or define component model</td>
<td>Component model</td>
<td>Component model specialist</td>
</tr>
<tr>
<td>Define analytic model</td>
<td>Analytic model</td>
<td>Attribute specialist</td>
</tr>
<tr>
<td>Define property theory</td>
<td>Property theory</td>
<td>Attribute specialist</td>
</tr>
<tr>
<td>Create measurement apparatus for component and assemblies</td>
<td>Measurement framework</td>
<td>Measurement specialist</td>
</tr>
<tr>
<td>Embed property theory in constructive model</td>
<td>Automated reasoning capability</td>
<td>PECT designer</td>
</tr>
<tr>
<td>Integrate component and analytic model with the measurement framework</td>
<td>Constructive technology</td>
<td>PECT designer</td>
</tr>
<tr>
<td>Develop analysis tool</td>
<td>Analysis tool</td>
<td>Attribute specialist</td>
</tr>
<tr>
<td>Develop runtime</td>
<td>PECT runtime</td>
<td>PECT designer</td>
</tr>
<tr>
<td>Specify description schemas</td>
<td>Component and assembly description schemas</td>
<td>PECT designer</td>
</tr>
<tr>
<td>Develop a tool that can create predictions given assembly and component descriptions</td>
<td>Predictor tool</td>
<td>Attribute specialist</td>
</tr>
</tbody>
</table>
3. Predictable Assembly of Components

3.4.3 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 3.5 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 process is not sufficient to produce the wanted results. If it is not sufficient, then we go back to phase 2 to refine the technology and experimental process 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 3: Validation phase activities, artifacts and workers

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

3.4.4 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 in the target environment. Also the framework can help the application developer to test the application to a greater extent.

Table 4: Packaging phase activities, artifacts and workers

<table>
<thead>
<tr>
<th>Activity</th>
<th>Artifact</th>
<th>Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create deployable PECT</td>
<td>Packaged PECT</td>
<td>PECT designer</td>
</tr>
<tr>
<td>Make measurement framework available</td>
<td>Packaged measurement apparatus</td>
<td>Measurement specialist</td>
</tr>
</tbody>
</table>

3.5 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 mean 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 will be described by assigning values to certain variation points. The variation points are different dimensions in which the assembly may vary.

• 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 [41].

The empirical validation procedure is described in more detail in the following five subsections and the steps shown in Figure 17:
3. Predictable Assembly of Components

Figure 17: Process of empirical validation

3.5.1 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
3. Predictable Assembly of Components

component in question. If the component properties can be 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 7 on page 67).

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 [83].

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.

### 3.5.2 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, all different variation points have to be identified. There are suggestions of how this can be done in the ATAM for architectures [58]. 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” [58]. 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 be removed. It included the variation point will only introduce unnecessary complexity. Other variation points might be size and performance. All variation points V0, V1 to Vn, are dimensions in the variation hyperspace which can be described with a tuple V = ⟨v0, v1, …, vn⟩.

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.
As shown in Figure 18 there are several assemblies that are possible to create using the PECT, but using judgment sampling there is knowledge that defines 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.

![Figure 18: An example of how samples are selected in a non-feasible way](image)

Using judgment sampling, we can achieve higher confidence in the property theories since the validation aim for the realistic subset of possible assemblies. Figure 19 illustrate that the lion part of the sample set of assemblies is 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 have 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.
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.

3.5.3 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.
3. Predictable Assembly of Components

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.
3. Predictable Assembly of Components

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. 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 and 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.

3.5.4 Collect Sample Data

The procedure of how each sample shall be executed shall be described. This is a more detailed description addressing only a part of the overall analytic study process. Where the analytic study process describes how all samples shall be validated together to achieve the statistical labels stated in the goal, the sampling procedure focus on how each sample shall be treated. In addition, the sampling procedure describes how the selection of samples shall be performed.

The reasoning framework with the property theory will be 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.

### 3.5.5 Analyze Results

When the all sample data is collected the results can be analyzed. Normal statistical 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.

### 3.6 Summary and Conclusion

This chapter describe the approach of having predictable assembly of components. The approach taken is to defined a prediction enabled component technology that has capabilities to do predictions built in. When designing a PECT it is possible to use existing component technologies but that 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 and another method is provided to perform validation of such.
4 A Latency Experiment of PECT

Latency is the time it takes to execute an assembly from point A to point B. Measurements points are defined as pins on a component where measures can be taken as shown in Figure 20.

![Diagram of Latency Experiment](image)

**Figure 20:** Measurement points for latency are defined on the pins of components

The time it takes to execute the path from point A to point B is the latency for the assembly of the four components.

In this example, there are two different PECT implementations, the operator and controller PECT. 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...
4. A Latency Experiment of PECT

(HMI). The controller is responsible for controlling input and output signals to machinery or devices and the operator display is part of the HMI.

A typical problem addressed in this example is the one to predict the latency from that a button is pressed in a HMI until a specific user control is updated. Another latency problem addressed is the time it takes from a signal is received by the controller until the controller has created a corresponding output signal.

It is shown later that the HMI latency 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.

4.1 Scope and Goal of Chapter

TBW

4.2 The operator PECT

This section outlines the operator PECT with the purpose to predict latency for human machine interface (HMI) applications and components. First, an overview of the operator PECT is given and later how this PECT was empirically validated.

The execution model is simple with only one thread of execution which gives no parallelism. Invocations between graphical user components are carried out in a synchronous manner which gives a straight line of execution once a component has been invoked.

The operator is monitoring and controlling the automation system through use of the HMI which is communicating with the controller for getting data to monitor or setting execution orders for control.
The example of an assembly of display components is shown in Figure 21 where two buttons, Select and close, are connected to a switch components that carries out the order. A meter component constantly pulls data from the underlying controller and pushes the result to a visible gauge. The operator can use the buttons to control the process and the gauge to monitor its state.

![Diagram of a display with buttons connected to a switch that is metered by a gauge](image)

**Figure 21:** A display with buttons connected to a switch that is metered by a gauge

The button components are connected with a connector to the switch component, since this is a direct connection decided upon at design time; it is called a constructive connector. The analytic connector is merely used to illustrate that it is possible to insert measurement points in the assembly of components. In Figure 21, the analytic connector indicates that latency, from when a button is pressed until the gauge shows a result, is measured.

The goal of the operator PECT is to be able to predict latency with a magnitude of relative error (MRE) less than 10% with a 95% confidence.

The constructive design of the component technology is built on .Net technology using Windows 2000. The component model is Pin as described in [42].
4. A Latency Experiment of PECT

For the analytic part we have chosen a simple theory for assembly latency in the operator PECT on the assumption that execution times for each component could be summed, i.e. the latency for each component in the path defined by the assembly is added. The latency for an operator assembly with respect to a execution path, $L(A, P)$ is:

$$L(A, P) = \sum_{\forall C_i \in P} L(C_i)$$

(8)

, where $P$ is the path of components that will execute in the assembly.

Taking the summation of the components execution time might not be the most representative theory of the reality but it is a theory good as a starter. As shown earlier 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 might need improvement. This will also be shown later in this example. Validation results and their corresponding samples should be described to know the limits of the validation. Such a prediction theory might work in the area of assemblies used for validation but not for another area.

After the constructive part was designed together with the infrastructure for taking measures, the first instance of a PECT was available. The PECT was validated following the defined steps for empirical validation.

XML was 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 of one operator PECT assembly described in XML can be found in Figure 22.
4. A Latency Experiment of PECT

The figure shows that there are four components of different component types. Each component has 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. The assembly description also shows 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 because of space restrictions. The same assembly can be found in Figure 21 where the connection from the close button to the switch is shown.

The components 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.

```xml
c<?xml version='1.0' ?>
<Assembly xmlns="PinTeKXML.xsd">
  <Components>
    <Component name="Gauge" type="SEI.Components.SEIAngularGauge">
      <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.SEISwitch">
      <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 21**

The components 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.
4. A Latency Experiment of PECT

4.2.1 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 $\text{MRE} \leq 0.10$ and confidence level $\gamma = 0.95$. This means that a prediction stated for an assembly will have a MRE less than 10% with a confidence of 95% in that prediction.

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

4.2.2 Validation Process

Characteristics of the assemblies are defined for the operator PECT as a tuple with the following 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 needed 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. $v_1 = \text{Set of canonical topology rules}$
2. $v_2 = \text{Size (\#Components, \#Connections)}$
3. $v_3 = \text{Performance (lower execution time, upper execution time)}$

The canonical topology rules accommodate loops and branch and there are several topologies of assemblies, which can be seen as patterns. These patterns are used to identify how complex the topology of an assembly is. An example pattern is the regulator with feedback (Figure 23) which has a loop in it. All assemblies with a loop according to this pattern are said to belong the same category of topology.
4. A Latency Experiment of PECT

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 shall be defined for the analyzed PECT. An example of what is not allowed in PIN is to connect a source of a component to its own sink as shown in Figure 24.

An assembly can inhabit many different patterns at the same time. For example, an assembly might have both loops and branches as shown in Figure 25. Our way to represent each topology pattern is to number each pattern using bit representation. 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, a loop pattern would be represented with 0x1 and a branch pattern with 0x2.
4. A Latency Experiment of PECT

Figure 25: Which one of the sources that react on the sink is up to the logic of the component.

There are different means to specify reactions to what happens on sink pin and one approach is to use a CSP [46] notation. A CSP specification as the one below describes the reaction for the evaluator component in Figure 25:

\[
\text{EVALUATOR} = \text{value?}x:N \rightarrow \text{if } x>15 \\
\quad \text{then valid } \rightarrow \text{EVALUATOR} \\
\quad \text{else notvalid } \rightarrow \text{EVALUATOR}
\]

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. Either the valid or invalid source pin will be triggered but not both.

There are also size and performance rules that define variation points and 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 26. Average execution time is obtained by measuring the components in a test environment.
4. A Latency Experiment of PECT

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. Figure 27 shows that the measured execution times for the five components C1 to C5 is roughly 10 ms and this average execution time will be used when predictions are later carried out. 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.

```xml
<?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 27: Example XML describing the latency property of several components

The measurements were carried out assuming only one thread of execution, communication over synchronous calls and that components have only one sink.
4. A Latency Experiment of PECT

To facilitate generation and execution of different assemblies we have built a special-purpose component of type (named SEI.Component.SEITest). This component has 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 sample. The component used in these assemblies will be of the SEI.Component.SEITest type.

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

4.2.3 Measurement Apparatus

A test environment and measurement infrastructure is designed to be able to carry out the experiments. The target machine is a P4 class machine with at least 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 component properties are stored on XML files describing
each component. These properties are then feed into the prediction theory and analytic model to provide a prediction.

After the prediction is done, each sample assembly evaluated is run 1000 times to gather the measured latency.

4.2.4 Collect Sample Data

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

\( \langle 0x0, \langle 4, 1 \rangle, \langle 10, 10 \rangle \rangle \) 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.

\( \langle 0x2, \langle 4, 2 \rangle, \langle 10, 10 \rangle \rangle \) 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.

\( \langle 0x2, \langle 20, 2 \rangle, \langle 60, 60 \rangle \rangle \) is a set of 20 components with 40 connections accommodating at least one branch. The execution time is 60 ms.

\( \langle 0x2, \langle 30, 4 \rangle, \langle 10, 100 \rangle \rangle \) is a set of 30 components with 40 connections accommodating at least one branch. The execution time is distributed randomly in the range from 10 to 100 ms.

\( \langle 0x0, \langle n, 1 \rangle, \langle 10, 10 \rangle \rangle \) 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.

4.2.5 Analyze Results

The different samples where executed and analyzed and after the first set of executed samples it was possible to get feedback if the sampling procedure is working and if there are any immediate errors. The results from three different sample sets are shown in Table
4. A Latency Experiment of PECT

5, Table 6, and Table 7. In the first two sets, the number of components varies while the other variation points are fixed. By having all variation points fixed but one, it is easier to analyze what is being performed in the PECT. For instance in the case where the number of components is changed while the topology is a straight line of components and 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.

Table 5 and Figure 28 show a case where the number of components is increased but there is an unexpected phenomenon with a growing MRE. The table shows that the predicted latency is in line with 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.

Figure 28 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.
Table 5: Sample where the size parameter is changed but the error grows unexpectedly\(^1\)

<table>
<thead>
<tr>
<th>#components</th>
<th>Predicted (ms)</th>
<th>Measured (ms)</th>
<th>MRE</th>
<th>STD</th>
<th>ERR</th>
<th>REL ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,042</td>
<td>9,988</td>
<td>0.54%</td>
<td>0.439</td>
<td>0.096</td>
<td>0.96%</td>
</tr>
<tr>
<td>5</td>
<td>50,675</td>
<td>50,180</td>
<td>0.99%</td>
<td>0.773</td>
<td>0.277</td>
<td>0.55%</td>
</tr>
<tr>
<td>10</td>
<td>100,410</td>
<td>103,819</td>
<td>-3.28%</td>
<td>1.671</td>
<td>0.598</td>
<td>0.58%</td>
</tr>
<tr>
<td>20</td>
<td>198,845</td>
<td>215,441</td>
<td>-7.70%</td>
<td>1,811</td>
<td>0.648</td>
<td>0.30%</td>
</tr>
<tr>
<td>50</td>
<td>489,980</td>
<td>611,949</td>
<td>-19.93%</td>
<td>3,148</td>
<td>1,126</td>
<td>0.18%</td>
</tr>
<tr>
<td>100</td>
<td>978,577</td>
<td>1468,827</td>
<td>-33.38%</td>
<td>6,200</td>
<td>2,219</td>
<td>0.15%</td>
</tr>
<tr>
<td>1000</td>
<td>10000</td>
<td>59572,878</td>
<td>-83.21%</td>
<td>No Calc</td>
<td>No Calc</td>
<td>No Calc</td>
</tr>
</tbody>
</table>

![Predicted vs measured](image)

Figure 28 Predicted versus measured latency where large deviation is depending on design of load function

In the operator PECT experiment a sample assembly is measured at least 30 times to obtain confidence in the measured value. The STD is the standard deviation for the measurements and ERR is the average absolute error in comparison with the average

\(^1\) The sample with 1000 components has a rough estimate of the predicted value and only one measured value, hence no STD, ERR and REL ERR.
measured latency. REL ERR is the relative error of the measurement and should not be mistaken with the MRE which is the magnitude of relative error of the prediction compared to the measured value.

Table 6: Same experiment as shown in Figure 28 but with error corrected in the PECT validation infrastructure

<table>
<thead>
<tr>
<th>#Components</th>
<th>Predicted (ms)</th>
<th>Average Measured (ms)</th>
<th>MRE</th>
<th>STD</th>
<th>ERR</th>
<th>REL ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9,469</td>
<td>9,780</td>
<td>-3.18%</td>
<td>0.017</td>
<td>0.006</td>
<td>0.06%</td>
</tr>
<tr>
<td>5</td>
<td>46,743</td>
<td>46,994</td>
<td>-0.54%</td>
<td>1.025</td>
<td>0.367</td>
<td>0.78%</td>
</tr>
<tr>
<td>10</td>
<td>94,191</td>
<td>92,858</td>
<td>1.44%</td>
<td>2.275</td>
<td>0.814</td>
<td>0.88%</td>
</tr>
<tr>
<td>20</td>
<td>188,158</td>
<td>182,519</td>
<td>3.09%</td>
<td>2.375</td>
<td>0.850</td>
<td>0.47%</td>
</tr>
<tr>
<td>50</td>
<td>469,725</td>
<td>453,506</td>
<td>3.58%</td>
<td>3.609</td>
<td>1.291</td>
<td>0.28%</td>
</tr>
<tr>
<td>100</td>
<td>939,985</td>
<td>904,596</td>
<td>3.91%</td>
<td>3.630</td>
<td>1.299</td>
<td>0.14%</td>
</tr>
<tr>
<td>1000</td>
<td>9300</td>
<td>9102,020</td>
<td>2.18%</td>
<td>140,731</td>
<td>50,360</td>
<td>0.55%</td>
</tr>
</tbody>
</table>

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 of the number of times 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 29 and Table 6 shows what the measurements where with the new load generating function in place.

---

2 The predicted latency for an assembly with 1000 components is rough and not calculated with any tool.
Figure 29: 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 7 and Figure 30 shows the results when the number of connections has been changed and variation takes place in the number of components.

Table 7: Each component has five source pins connected to different components and the size is varied

<table>
<thead>
<tr>
<th>#components</th>
<th>Predicted (ms)</th>
<th>Average Measured (ms)</th>
<th>MRE</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,46</td>
<td>9,96</td>
<td>4,9%</td>
<td>0,04</td>
</tr>
<tr>
<td>5</td>
<td>1210,11</td>
<td>1070,82</td>
<td>13,0%</td>
<td>6,03</td>
</tr>
<tr>
<td>5</td>
<td>3996,28</td>
<td>3553,21</td>
<td>12,5%</td>
<td>36,76</td>
</tr>
<tr>
<td>10</td>
<td>7471,76</td>
<td>7242,73</td>
<td>3,2%</td>
<td>62,50</td>
</tr>
</tbody>
</table>
4. A Latency Experiment of PECT

Figure 30: 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%. Interesting to note is that the MRE improves with more components but no further investigation was carried out to understand this.

The overall results from the last sample give the following descriptive statistics:

Table 8: Descriptive statistics of the last operator sample set

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples (N)</td>
<td>11</td>
</tr>
<tr>
<td>Mean MRE</td>
<td>4.8%</td>
</tr>
<tr>
<td>Median MRE</td>
<td>3.2%</td>
</tr>
<tr>
<td>Standard Deviation (SD)</td>
<td>4.11%</td>
</tr>
</tbody>
</table>
| 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’s correlation test does not make assumptions about the distribution of the data which is useful in our example where we do not have a normal distribution of the data. I.e. 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
4. A Latency Experiment of PECT

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 8 the probability is very small and neglectable. The Shapiro-Wilk normality test shows whether the distribution is normal or not and a p-value greater than 0.05, which is generally accepted level, indicates that the data follow a normal distribution. In the operator PECT example, we see 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 found out to be very simple in terms of complexity of execution model 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 does 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 strange 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.
4. A Latency Experiment of PECT

4.3 Controller PECT

The controller model problem is taken from the substation automation domain and it was first described by Preiss in [81] and served as the basis for the problem tackled in [41] and in this thesis. Figure 31 shows how substation logic could look like and it depicts components of types specified in the IEC 61850 standard [47]. 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.

![Diagram of substation model problem](image)

Figure 31: The substation model problem represented using the IEC 61850 standard components

The MMXU calculates the effect using current and voltage. Over-current protection is implemented in the PIOC component and it opens the circuit breaker in case of too 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 serve as a test bed with somewhat the same behavior as a substation switch. Of course the current allowed where several magnitude less than in the real case and from that perspective this was more a toy, 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 see that the breaker is tripped by software respectively hardware. For actual schematics of the SEI switch hardware see the experience report [41].

![Image of SEI toy switch for lab purpose](image)

**Figure 32: The SEI toy 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.
4.3.1 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 \( \gamma = 0.99 \). 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.

4.3.2 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. 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 later.

The method for drawing inferences is based on input from [85]. 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 that the controller PECT inhabits are captured by having a tuple of variation points $V = \langle v_0, v_1, \ldots, v_n \rangle$. Each tuple defines the characteristics for one randomly generated assembly. Also, boundaries where set on each dimensional variation point. If each variation point represents a dimension then all possible assemblies can be found in the corresponding hyperspace. 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. One boundary example is, no assembly was considered for testing if it had more than 50 components. The following variation points were defined and Table 10 shows the samples chosen:

- **number of clocks**— the total number of Clock components (discussed above) that are used as a stimulus to other components within an assembly
- **number of components**— the total number of components, in this case, synthetic components, that make up an assembly
- **number of connections per source pin**— the maximum number of connections allowed for a single component’s source pin to be connected to other components’ sink pins
- **minimum load factor**— the minimum execution time for a component in the assembly. This number could range from 5 to 20 and had to be less than or equal to the maximum load factor
- **maximum load factor**— the maximum execution time for a component in the assembly. This number could range from 5 to 20
- **harmonic period**— determination of whether the clocks in the system have the same period
- **minimum clock period**— the minimum clock period for the clocks in the assembly. This value must be less than or equal to the maximum clock period
- **maximum clock period**— the maximum clock period for the clocks in the assembly
• communication type—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

• percent blocking—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 have values or even multidimensional values 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 500 ms 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 \( \langle \# \text{Components}, \# \text{Connections to source pin}, \# \text{Clocks}\rangle \). 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.

### 4.3.3 Measurement Apparatus

As the test and execution environment directly affect the execution time of components, it is important to record what software is running on the machine.
4. A Latency Experiment of PECT

Table 9: Software and hardware specification

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

Figure 33 shows how an assembly generator can be used to generate the samples. The generator takes as input the different defined characteristics and produces constructively valid assemblies using synthetic test components. The synthetic component does not perform any realistic task other than consuming CPU or performing other dummy tasks. 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.
Figure 33: Overview of an example analytic study experiment

The confidence on 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. One exception is the PECT when it is going to be used to develop products of a software product line [17]. In a product line, variation points are well defined together with the software architecture, making it easier to generate assemblies’ representative of future products.

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 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.

### 4.3.4 Collection of Sample Data

The sample sets covers different characteristics points and for each point several assemblies are generated. Table 10 shows what the characteristics of the samples are.

**Table 10: Variation points for the sample assemblies**

<table>
<thead>
<tr>
<th>Assembly Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Tuple</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
4. A Latency Experiment of PECT

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>2</th>
<th>8</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>N</th>
<th>20</th>
<th>100</th>
<th>M</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>4</td>
<td>35</td>
<td>2</td>
<td>15</td>
<td>20</td>
<td>Y</td>
<td>500</td>
<td>2000</td>
<td>A</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>35</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>500</td>
<td>2000</td>
<td>M</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>200</td>
<td>400</td>
<td>M</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>N</td>
<td>200</td>
<td>400</td>
<td>M</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>15</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>Y</td>
<td>200</td>
<td>400</td>
<td>M</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>15</td>
<td>3</td>
<td>20</td>
<td>20</td>
<td>Y</td>
<td>400</td>
<td>1200</td>
<td>M</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

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 assemblies generated were pure synthetic and provided no functionality except consuming CPU. One example of an assembly is found in Figure 34.

![Figure 34: An example assembly generated from a variation point](image)

The assembly shows four components and two clocks that have a period of 400 and 200 ms, respectively. It is an assembly that fits the eighth assembly space tuple in Table 10, 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 indicate a constant that is feed into the component during execution and for the controller target environment a factor of five resembles roughly five ms. 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.

![Figure 35: The analytic view of the generated assembly with two tasks and their execution](image)

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. Figure 35 shows an example where task 1 releases four components 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. 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 [42]. The predictor takes as input the tasks it should analyze. The tasks have priorities, starting offset, subtasks and
4. A Latency Experiment of PECT

periods. Each subtask has priority and execution time. After the predictor has run the simulation, a prediction for each job with in a task is presented as shown in Table 11.

Table 11: The predicted values for the example assembly

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Task</th>
<th>Job</th>
<th>Predicted Job Latency (ms)</th>
<th>Predicted AVG Task Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>21,285</td>
<td>21,32</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>26,509</td>
<td>15,85</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>5,224</td>
<td></td>
</tr>
</tbody>
</table>

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 (9) 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:

$$L^{n+1}(c_i) = c_i.wcet + B(c_i) + \sum_{\forall c_j \in hp(c_i)} \left[ \frac{L^n(c_i)}{c_j.T} \right] c_j.wcet \quad (9)$$

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

Worst-case latency was not the goal and to achieve average latency simulation has been chosen as the mean to calculate it. 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 12), which is defined as the average of each job in a hyper-period, is used for comparison with the predicted one.
Table 12: Average measured latency is achieved for each task in the assembly

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Task</th>
<th>Job</th>
<th>Number of measurements</th>
<th>Average Measured Job Latency</th>
<th>Standard Deviation</th>
<th>Average Measured Task Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>21,317</td>
<td>0,0122</td>
<td>21,317</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>30</td>
<td>26,442</td>
<td>0,0047</td>
<td>15,845</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>30</td>
<td>5,248</td>
<td>0,0045</td>
<td></td>
</tr>
</tbody>
</table>

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 16 and Table 17. Each predicted average task latency is compared with the measured average task latency by calculating the magnitude of relative error (see equation 7).

### 4.3.5 Analyze results

The first raw results of measuring the predicted 65 measurement points are displayed in Table 16. The standard deviation correlates to the average measured latency for each sample. It is noteworthy to point out that having synthetic components with well-defined behavior results in little surprises and deviation of the measured latency. The overall results from the last sample give the following descriptive statistics:

Table 13: Descriptive statistics of the first controller sample set

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples (N)</td>
</tr>
<tr>
<td>Mean Absolute MRE</td>
</tr>
<tr>
<td>Median Absolute MRE</td>
</tr>
<tr>
<td>Standard Deviation (SD)</td>
</tr>
<tr>
<td>Spearman rank correlation of predicted latency and average measured latency</td>
</tr>
<tr>
<td>p-value &lt; 0.0001</td>
</tr>
<tr>
<td>Shapiro-Wilk Normality test for MRE</td>
</tr>
<tr>
<td>p-value &lt; 0.05</td>
</tr>
</tbody>
</table>
4. A Latency Experiment of PECT

The Shapiro-Wilk test shows a p value much less than the accepted 0.05 and that indicates that the distribution is not normal. For a distribution that is non-normal, distribution free statistical intervals must be used.

![Scatterplot (controller 1ed.sta 10v*65c)](image)

Average Latency (ms) = 2,9851 + 0,9219 * x

Predicted Latency (ms): Average Latency (ms): $r^2 = 0.9846; r = 0.9923, p = 0.0000$

**Figure 36: A regression analysis of the first sample set**

For the sample, the correlation between the predicted and measured value is wanted. 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 correlation analysis, which allows us to assess the strength of the linearly relation between two variables—in our case, predicted and observed assembly latency. The result of such analysis is the coefficient of determination, $0 \leq R^2 \leq 1$, where 0 represents no
relation, and 1 represents a perfect linear relation. In a perfect prediction model, predicted and observed latency would be identical; therefore, the goal for the model builder is a linear relation.

Figure 36 shows how well the predicted value is corresponding to the average measured value. The correlation is high for smaller latency but when for larger latency figures the graph indicates worse correlation. To get a better understanding of whether the error is growing we have a residual graph in Figure 37.

![Predicted vs. Residual Scores](image)

**Figure 37: Residual analyze of the first sample**

The residual graph shows that there is a definite indication that the size of the predicted error depends on the size of the predicted latency. I.e. The number of outliers grows with the size of the latency.

To get an understanding of the distribution we plot a normal probability plot, this plot is shown in Figure 38. In the plot we have the MRE as the observed value on the x-axis and the expected normal value on the y-axis. If the observed values (plotted on the x-axis)
are normally distributed, then all values should fall onto a straight line in the plot. If the values are not normally 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 outliers, a normal distribution of the error. For positive MRE errors we get another shaped normal distribution. It seems that when predictions 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.

![Normal Probability Plot of MRE Signed (controller 1ed.sta 10v*65c)](image)

**Figure 38:** Normal probability plot that indicates that we have two distributions in the first sample

To get an overview of how big the errors are we plot a histogram (shown in Figure 39). This however shows that the error is distributed both on the positive and the negative side. 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 40).
4. A Latency Experiment of PECT

**Figure 39:** The histogram showing the magnitude of relative error

**Figure 40:** 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 possibly indicate an error in the measurement framework, constructive or the analytic model. A further analysis of the data is shown in [41], and it was concluded that an error in the constructive component model allowed inconsistencies between the analytic and the constructive model. The error was corrected and a new round of empirical validation was executed.

This time the results were much more encouraging. The results are shown in Table 14 and in [42].

Table 14: Descriptive statistics of the second controller sample set

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples (N)</td>
</tr>
<tr>
<td>Mean MRE</td>
</tr>
<tr>
<td>Standard Deviation (SD)</td>
</tr>
<tr>
<td>Spearman rank correlation of predicted latency and average measured latency</td>
</tr>
<tr>
<td>Shapiro-Wilk Normality test of MRE</td>
</tr>
</tbody>
</table>

As for the first sample set we want to look at the correlation of the predicted value and the measured value. Figure 41 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. By looking at the residuals it is possible to deduce this information. Figure 42 shows that it seems to still be a dependency on the size but it is not so big as in the first sample set where errors were present. 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.
4. A Latency Experiment of PECT

Predicted vs. Observed Values
Dependent variable: Average Measured

Figure 41: Linear regression test for the second sample

Predicted vs. Residual Scores
Dependent variable: Average Measured

Figure 42: Residual analysis show that there is one big outlier in the second sample set
If we take a look at the histograms of the second sample we can see that there are two outliers on 8% respectively -7% that clutter the graph, if these are removed it is possible to see the distribution of the rest of the errors. Such a graph is shown in Figure 43. A closer look at the outliers reveals that sample number 155 and task 3 consist of six different jobs with huge variation in latency. Depending on where in the hyper period a job for task 3 is executed the latency is from 18 ms. to 445 ms.. 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.
4. A Latency Experiment of PECT

It is interesting that the underestimated latency predictions with MRE values below zero are dominating. To see the relationships with the over versus underestimation of latency we use the normal probability plot. This is shown in Figure 44 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.
4. A Latency Experiment of PECT

Figure 44: The normal probability plot shows the same characteristics as in the first sample set

To demonstrate the absolute magnitude of relative error we plot the histogram as shown in Figure 45. 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.
4. A Latency Experiment of PECT

The histogram below shows the distribution of absolute MRE values for the 2nd sample set. The absolute MRE is calculated as

\[ \text{Absolute MRE} = 156 \times 0.01 \times \text{normal}(x; 0.0051; 0.0098) \]

The histogram indicates that 87% of the observations are within ±1%, while 11% are within ±2%. The mean absolute MRE is 0.0050812236, with a standard deviation of 0.00984134425, maximum of 0.0807630951, and minimum of 0.000000636354824.

### Figure 45: Histogram of the absolute MRE for the 2nd sample set

#### 4.4 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 mean for analyzing how good the prediction theory is. In addition, the method gives a mean 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.
4. A Latency Experiment of PECT

It is questionable if the statistical labels produced are of greater use since the data produced is weak from different perspectives. We have not been able to show normality in the error 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.

Another weak spot in the 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 in constructive component model or the analytic model that are found in the validation phase using this validation technique does not have to be found in the test phase or even worse at the customer site. The earlier, in the development cycle, faults are identified and removed the lesser the cost for the development.
5 A Dependency Experiment

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 [64]. There might be a check 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. Managing complexity in software systems is also addressed in [79] and the experiences from product data management and software configuration management can be used to discuss the dependency problem.

To illustrate the problems we highlight two relevant cases and propose solutions to the dependence 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 37 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. This change of characteristics is perhaps acceptable
5. A Dependency Experiment

for a particular product, but the consequences for the rest of the software products are unknown.

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. 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.

There is generally a lack of information available for identifying components in systems. 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 the component models that differ provided from required interface implicitly identify dependencies. Dependency information between components is necessary to predict the effects of updating the system with new components.

In an approach where variants of components are of importance, as for instance in product lines [17], the handling of consistency is a 2-dimensional problem (Figure 46). 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 46: The 2-dimensional version-variant concept](image)

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 a property is introduced on the assembly. The consistency property, $A.\text{consistent}$, 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 in the product line. 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 idea of having version dependencies is very similar to how .NET assemblies use meta-data to describe dependencies to other assemblies [100]. Dependencies can be expressed and assured using object constraint language (OCL [115]) constraints for the
5. A Dependency Experiment

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.

For the purpose of predicting variant- and version consistence on an assembly, a property that expresses the dependencies must be introduced. The analytical property $c.depends$ is used to host the 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 $<C, \text{variant}, \text{version}>$ identifies a variant version of a component $C$.

The consistency of all variants and versions in an assembly can be calculated with the following definition (equation 10). The property consistent is of type Boolean with possible values true and false and has the following definition:

\[
\text{An assembly } A \text{ is variant- and version consistent, } A.\text{consistent} \text{ iff: } \\
A.\text{consistent} = \forall \langle \langle c_i, \text{variant, x} \rangle, \langle c_i, \text{variant, y} \rangle \rangle \in V \times V: x=y \\
\text{where } V = \bigcup_{c_j \in C(A)} c_j.\text{depends}, \\
C(A)= \text{the set of all components in an assembly } A, \\
\text{variant is a component variant and } x,y \text{ are versions.}
\]

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 don’t 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.
5. A Dependency Experiment

5.1 Scope and Goal of Chapter

5.2 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 constitutes a component’s data interface [96].

We illustrate the problem of adding a new component to a software system with a new component C4. 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 7 and Figure 47). 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.

```
«constructive»
C4 : Component
inports = {I4, I5}
outports = {}
periodTime : Time = 40
priority : int = ?
deadline : uint = 15

«precondition»
{C3.n_executed > C4.n_executed}

«precondition»
{C3.O3 = C4.I4,
C2.O2 = C4,I5 }

«precondition»
{C4.depends.includes(<C3,1,2>, <C2,1,1>)}
```

Figure 47: A new component c4 is added to represent a new feature of a product.

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.
5. A Dependency Experiment

When component C4 is added to the assembly, it is possible to check the version consistency by using equation 10 above.

\[ V = \bigcup_{c_i \in C(A)} c_i.\text{depends}, c_i \in C(A) = \]
\{<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. The next section outlines strategies for what to do if such dependencies are not available in the component model.

### 5.3 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 [63]. 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 with a working prototype. The prototype is able to browse the dependencies in a system and to store them under version control. The prototype incorporates ideas from [64,65]. It is also used to gather information about
changes made between two 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.
5. A Dependency Experiment

Experiments have shown 1993 components and 8936 edges in a Windows 2000 workstation configured for software development. It took 2 minutes and 4 second to compute all the dependencies on an Intel Celeron 300 Mhz cpu. 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 48 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, only direct dependencies have been measured before the transitive closure has been calculated. If the trend is decreasing as show with the solid line in Figure 48, it is a measurement that the system is less
complex compared with a system having an increasing trend of dependencies between components.

A graph similar to that shown in Figure 48 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 reversed before performing this kind of analysis. 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 configuration theory can be applied when the dependencies are discovered but not when there are dynamic dependencies.

Another 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 run-time. A client uses
5. A Dependency Experiment

A unique key to find the server component in the registry and then the COM run-time 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.

If we do not know which components a program uses at run-time, 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 49). 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 : IUnknown
{
    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 49: 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. 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.

### 5.4 Summary and Conclusions

Functional 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 all dependency analyses, there is a need to identify a component and its version. 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 [35] to wrap components with a version information interface. Hoek [104] 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 [64].
Component-based development (CBD) is of great interest to the software engineering community and it has achieved considerable success in many engineering domains.

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.

Some of the main advantages of CBD 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 properties are required of the components concerned?
6. Classification and Capabilities of Quality Attributes

- 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 [26], and particular models of certain properties have been analyzed [74,94], 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 [28,52,82,97], 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.

In this chapter, our intention is to demonstrate the diversity of quality attributes and the different methodologies which can be used for predicting system behavior 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, i.e. the ability to predict the properties of component assemblies. Such a classification indicates the feasibility of the component-based approach for building dependable systems.

The chapter is organized as follows: Section 6.1 identifies the types of properties according to the principles for predicting the properties of component assemblies.
Section 6.3 discusses the possibility of defining component composition in a recursive way. Section 6.4 provides a list of many quality attributes and classifies them according to the compositional principles discussed in the previous sections. Finally, section 6.5 exemplifies the reasoning by showing quality attributes for safety-critical systems.

### 6.1 Scope and Goal of Chapter

TBW

### 6.2 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: Run-time 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 shall use a generic term Assembly \( (A) \) which simply denotes a set of interacting components. Such an assembly can be a part of a software system (for example a functional unit, or a subsystem), or the entire system. The only characteristic we want to relate to an assembly is a set of integrated 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.

\[
P = \text{property}, A = \text{assembly}, c = \text{component} \\
A = \{c_i\} \\
P(A) = f(P(c_i)); \; i \in N
\]  

(11)

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 (11). 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:
The function $M(c_i)$ is different for different technologies. For example in the case of the separation of composition time from run-time 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 [105], 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 (12) 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.

$$M(A) \leq \sum_{i=1}^{n} M_{\text{max}}(c_i)$$

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.
6. Classification and Capabilities of Quality Attributes

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.

\[ SA = \text{software architecture}, \quad x_k = \text{connections} \]
\[ P(A) = f(P(c_i), SA(c_i, x_k)); \quad i, k \in N \] (14)

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 50. 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 50 is related to the number of clients and the number of server components. 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 50: A typical multi-tier architecture with client and servers variability points affecting the performance quality attribute

According to [55,118] the time per transaction $T/N$ expressed in (15) 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.

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

$T/N = \text{execution time per transaction}$

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

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

\[
P(A) = f(P_1(c_i), P_2(c_i), \ldots, P_k(c_i));
\]
\[
i, k \in N
\]
\[
P = \text{assembly property}
\]
\[
P_1, P_k = \text{component properties}
\]

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 [23,41,110]. 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 51. 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.

![Composition of port-based components](image)

**Figure 51: 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.

\[
P(A, U_k) = f(P(c_i, U'_{i,k})); \quad i, k \in N
\]

\[P = \text{property for a particular usage profile}\]

\[U_k = \text{assembly usage profile}\]

\[U'_{i,k} = \text{component usage profile}\]

The behavior 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 (\( U_{i,k} \)) 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 [23]. 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 52).
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.

\[ U_1 \subseteq U_k = P_{k_{\text{min}}}(A, U_1) \leq P(A, U_1) \leq P_{k_{\text{max}}}(A, U_1) \]  

(18)

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 52 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), 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.
6. Classification and Capabilities of Quality Attributes

\[ P_k(S, U_k, E_l) = f(P_k(c_i, U_{i,k}), E_l); \quad i, k, l \in N \]

\[ U_k \quad \text{System usage profile;} \]

\[ E_l = \text{Environment context} \]

\[ S = \text{System} \]

\[ U'_{i,k} = \text{Component usage profile} \]  \( (19) \)

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 behavior of a system. Properties that depend on the environment in which a system is deployed are generally hard to derive from the component properties.

6.3 Assemblies and Systems

We have used the generic term “assembly” for a set of integrated components. In the previous section, it is shown that 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, something that understands and can run this 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 [23]. 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 [105], 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, means 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, 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 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 6.1 will be derived in the following way:
6. Classification and Capabilities of Quality Attributes

\[ P(A_a) = f(P(A_k) = f(f(P(c_i))); i, k \in N \]
\[ A_a = \{A_k\}; A_k = \{c_i\} \]
\[ P = \text{property}, A_a = \text{assembly of assemblies} \]
\[ A_k = \text{assemblies}, \quad c_i = \text{components} \]

For the memory consumption case in equation (12), we have:

\[ M(A_a) = \sum_{i=1}^{k} M(A_i) = \sum_{i=1}^{k} \sum_{j=1}^{n} M(c_{ij}) \]

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.

### 6.4 Composition of Quality Attributes

The literature contains different names for system (and component) properties: non-functional properties, extra-functional properties, or quality attributes. When identifying a particular property of a system or of a component we use the term property, and when referring to these properties in general, we use the term quality attribute. 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. Complex attributes are, for example, dependability or performance while tangible attributes could be, for example, memory footprint, scalability or availability. The
complex attributes consist of several more tangible attributes, and consequently the complex attributes are harder to classify in one category only. 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 run time or not. Preiss et al show such a mapping of quality attributes from the runtime observable perspective [82].

To demonstrate that there are different quality attributes in the various categories we have assembled a table (Table 15) showing the classification of the attributes. The quality attributes are mainly taken from [97] and a short description of each attribute is to be found in Table 18 and [3,7,57,82]. According to [52] certain attributes grouped together constitute complex attributes. As these attributes are vaguely defined, and we have in some cases slightly modified them to achieve better consistency, they are in the table designated as concerns. Some inconsistencies remains, for example, “maintainability” is defined as a concern and as an attribute. Such inconsistencies are the result of having different requirements, priorities, development processes and focus when performing the categorization.

The table shows the dominant type of composition and possibly another related composition type. For each attribute there is one dominant type of composition, marked with an “xx” that has the greatest impact on the particular attribute. Other related composition types, not as dominant, 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 15: The different quality attributes and their classification

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Quality Attribute</th>
<th>Directly composable</th>
<th>Architecture-related</th>
<th>Derived</th>
<th>Usage-dependent</th>
<th>System environment context</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usability</strong></td>
<td>Accessibility</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Administrability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Understandability</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Generality</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Simplicity</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td>Mobility</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nomadicity</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hardware independence</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Software independence</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Accuracy</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Footprint</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Responsiveness</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scalability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Schedulability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Timeliness</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU utilization</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Latency</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transaction Throughput</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Concurrency</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Efficiency</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. Classification and Capabilities of Quality Attributes

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Quality Attribute</th>
<th>Directly composable</th>
<th>Architecture-related</th>
<th>Derived</th>
<th>Usage-dependent</th>
<th>System environment context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Flexibility</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changeability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Evolvability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Extensibility</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modifiability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tailorability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Upgradeability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Expandability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integibility</td>
<td>data consistency</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Version consistency</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adaptability</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Composability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interoperability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Openness</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heterogeneity</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integrability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>Audibility</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Completeness</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conciseness</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Correctness</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Traceability</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Coherence</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Analyzability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modularity</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployability</td>
<td>Configurability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Distributeability</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Ease of creation</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependability</td>
<td>Availability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Confidentiality</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maintainability</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Safety</td>
<td>x</td>
<td>xx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Security</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business</td>
<td>Cost</td>
<td>xx</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. Classification and Capabilities of Quality Attributes

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Quality Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projected Lifetime</td>
<td>Directly composable: x</td>
</tr>
<tr>
<td></td>
<td>Architecture-related: x</td>
</tr>
<tr>
<td></td>
<td>Derived: xx</td>
</tr>
<tr>
<td></td>
<td>Usage-dependent: x</td>
</tr>
<tr>
<td></td>
<td>System environment: x</td>
</tr>
<tr>
<td>Targeted Market</td>
<td>Directly composable: x</td>
</tr>
<tr>
<td></td>
<td>Architecture-related: xx</td>
</tr>
<tr>
<td></td>
<td>Derived: x</td>
</tr>
<tr>
<td></td>
<td>Usage-dependent: x</td>
</tr>
<tr>
<td></td>
<td>System environment: x</td>
</tr>
<tr>
<td>Time to market</td>
<td>Directly composable: x</td>
</tr>
<tr>
<td></td>
<td>Architecture-related: xx</td>
</tr>
<tr>
<td></td>
<td>Derived: x</td>
</tr>
<tr>
<td></td>
<td>Usage-dependent: x</td>
</tr>
<tr>
<td></td>
<td>System environment: x</td>
</tr>
<tr>
<td>Affordability</td>
<td>Directly composable: x</td>
</tr>
<tr>
<td></td>
<td>Architecture-related: xx</td>
</tr>
<tr>
<td></td>
<td>Derived: x</td>
</tr>
<tr>
<td></td>
<td>Usage-dependent: x</td>
</tr>
<tr>
<td></td>
<td>System environment: x</td>
</tr>
<tr>
<td>Development time</td>
<td>Directly composable: x</td>
</tr>
<tr>
<td></td>
<td>Architecture-related: xx</td>
</tr>
<tr>
<td></td>
<td>Derived: xx</td>
</tr>
<tr>
<td></td>
<td>Usage-dependent: x</td>
</tr>
<tr>
<td></td>
<td>System environment: x</td>
</tr>
</tbody>
</table>

The table and Figure 53 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 tangible concrete and related to technologies, while fewer are complex. 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.
6. Classification and Capabilities of Quality Attributes

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 have affect many quality attributes.

6.5 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 [3]. 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 complex 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 properties are emerging or derived system properties, which are both system and component properties?
- In a component-based system how are these properties related to component properties?
- To which extent (and how) can these properties can be determined from component properties?
- To which extent can the uncertainty of the predictability of these properties be minimized and how much is it related to the uncertainty of the component properties?

The following is a short analysis and classification of the dependability properties.
6. Classification and Capabilities of Quality Attributes

6.5.1 Reliability

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

\[
MTTF(A) = \frac{1}{P_f(A)}
\]

\[A = \text{module (component, assembly, module)}\]

\[P_f = \text{probability that module } A \text{ fails per time unit}\]

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 (14) and, because of the fact that the reliability is measured under stated conditions, a usage profile must be specified, which is expressed by (16). 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 [86,87]:

- 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 [87,117]. 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).

### 6.5.2 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):

\[
\text{Availability}(A) = \frac{\text{MTTF}(A)}{\text{MTTF}(A) + \text{MTTR}(A)}
\]

(23)

\[A = \text{module (component, assembly, module)}\]

From (23) 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
avaliability 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 run-time 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.

6.5.3 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.
6.5.4 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 [3] 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.

6.5.5 Maintainability

Maintainability is defined as the capacity to undergo repairs and modifications. Maintenance is of three different types, corrective, adaptive and perfective maintenance [3]. 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 15 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 [71]). 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.

6.6 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 behavior. 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
6. Classification and Capabilities of Quality Attributes

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.
7. Possible Implications of Design Decisions Based on Predictions

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 is 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.

7.1 Scope and Goal of Chapter

TBW
7. Possible Implications of Design Decisions Based on Predictions

7.2 Introduction to Moral Problems

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.
7. Possible Implications of Design Decisions Based on Predictions

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 [24,116].

Quality attributes should not be allowed to be a happening; 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 of that they can very seldom be 100% accurate; usually there is a different level of confidence of the predictability. 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 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 [39]:

---

Predicting Quality Attributes in Component-based Software Systems 185
7. Possible Implications of Design Decisions Based on Predictions

- 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.

This section addresses the morality of decision making based on empirically validated software.

![Figure 54: A brief overview of ethics in computing [77]](image)

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. 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 [14]. 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 that 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 [39,70]. 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 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.

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

---

3 ‘Deon’ = Latin for duty
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.

![Graph showing relationship between utilitarianism and virtue ethics.](image)

**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.
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 [49] and the ten commandments of computer ethics [5]. 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 CPSR: 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 Prediction of quality attributes

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. 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 [42,86] 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) [67]. 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 70ties that programming is fun and that the quality part of the work is not considered fun [14]. 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 [34,39], 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.

In the predictable assembly approach the aim is to predict the component and to some extent the system behavior with a 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.

7.6 Summary and Conclusion

TBW
8. Conclusion and Future Work

Predictability of quality attributes represented as properties of components and on assemblies increase the ability to reason about software behavior well before the testing phase of a software application. Having predictability would also provide means for addressing evolution of architecture or changes to execution environment. Before a change is made, it is possible to predict certain quality attributes to see if they withstand the change. If components are reused or exchanged a precise specification of component properties would support management of components.

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 ease the reasoning about quality attributes. Having this approach, leads to the following main research question:

*Can quality attributes be predicted in a component-based system given the attributes of the components?*

It is shown in Chapter 3.6 with the latency experiments performed that it is possible to predict a quality attribute, latency, in a component-based software system. However there are some open issues that have to be addressed before we have a general solution to this question. It has been shown in two experiments that it is possible to predict the latency quality attribute. Other parallel work to this thesis has shown that model checking can be
applied, according to the same principles about prediction enabled component technology, to predict safety attributes [54,112,113].

We have considered these research questions on two different quality attributes using different component-based systems and component models. The thesis do not emphasize the component models, or quality attributes themselves (this was done in the same research project but is not the focus in this thesis), but emphasize is on the process of validation and adjustments between component models and property theories.

The reasoning, the experiments analysis, and relations to industrial setting have shown that certain quality attributes of systems can be predicted from properties of components. It was also shown that the introduction of that process is not simple, but it enables different levels of accuracy according to the specification and restrictions of component models.

In order to answer the main research question several sub questions has been considered:

*Is it possible to develop a component technology that will support reasoning about system properties from component properties?*

It is shown possible in chapter 3.6 to develop a component technology supporting reasoning about latency attributes at an assembly level. To reach that goal several restrictions on the component model had to be applied. Since many component models today are not designed to provide prediction capabilities it is possible that prediction will imply restrictions on the component models or even restrictions on the use of the component model.

This subtle difference in restriction does have a big impact. If we restrict the component model, we also actively prevent users to build components that violate the rules. On the other hand, if we restrict the use of the component technology then there is still a possibility to misuse the underlying component technology. It is very much as
saying to a developer that he or she is only allowed to use certain functions of the underlying operating system.

In that case, we rely on the discipline of the developer but if the underlying technology does not allow misusage then the discipline of the programmer is not primary concern. Nevertheless, of course software developers must have certain amount of discipline when they work in such a open en free world as software development. The degrees of freedom are usually more than needed by a developer and this often causes late changes etc. that affect the final quality of the program. The discussion about late changes and engineering discipline is not covered in this thesis, even if it is interesting.

A more serious problem could be different requirements for different 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 [58].

How these restrictions are imposed is of importance since it is probably not until we know how to design our component model that we can achieve prediction capabilities in our systems. An analogy is that we cannot test an already faulty designed software product to perfection; the faults must be removed already at the design phase. Another approach is that prediction capabilities of component models will restrict the component model and hence the component model must be designed with prediction in mind.

There are different component technologies – they conform differently to the different prediction models. The question is how to improve the predictability? A) by improving component models, B) by introducing different restrictions on the existing component models c) a combination of this. Further research on this subject is needed.

More future work would be to implement the approach for a particular component-model in industrial settings and build a number of property theories with their relations to
the component model. Doing this would gain confidence that we can take research results to solve a problem in a practical industrial setting.

More research is needed to find out how it is possible to build up the component model to enable predictability of more and more quality attributes, i.e. to have an extendable component model that can be used to reason about many different future quality attributes.

A Lesson learnt by addressing this research question is that even if it is possible to develop component technologies, that support reasoning of quality attributes, the restrictions imposed or needed might cripple the functional design of such a technology. There is a trade-off between functional and analytic capability of such a component technology. It is important that what is produced have enough functional capabilities to build and apply real world practical problems. If we fail to do that, the technologies developed will never leave the research setting.

*How can we verify the predictions to gain objective trust?*

A theory how the assembled components affect the quality attributes of a system is valuable or even a must if we want to predict the attributes. If we have such a theory then it is possible to reason about the whole by knowing the parts. The theory can be implemented for automatic reasoning or a manual task. A desired goal is to automatically get the wanted predictions just after the components have been assembled. By having the information about the assembly and the component properties this should be possible. If we want to achieve prediction in practice we should strive to have the theories automated. This also suggests that the reasoning theory has to be designed into the component technology from the start.

Any prediction theory must be validated to show that the theory holds. It is shown in chapter 3.6 that a sample set for the validation must be taken carefully using
argumentation from the theory and from the experimental results. It is important that the sample set is a good representative of the population of all possible assemblies.

As long as we cannot show that the sample set is a representative of the population it is hard to obtain objective trust in the validation. However, it is possible to show how the validation is done and by having domain knowledge a certain degree of subjective trust is possible to gain. This trust might be sufficient depending on the criticality of the target applications. Chapter 7 discusses more in details implications of basing design decisions on predicted quality attributes.

As future work we propose to investigate more how to achieve object trust in attributes of a component and also in the predictions made for those attributes. More quality attributes should be tried out if possible to automate the reasoning about them.

Lesson learnt is that it is important to show how the objective trust has been obtained. Hence it is vital to keep experiment data and also to demonstrate the method how the validation or certification was conducted.

**What properties or quality attributes are suitable for prediction?**

Chapter 6 shows may 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 components are put together. A reasoning theory should exists about the particular attribute in question to make it possible to predict attributes of an assembly knowing only the component attributes.

If quality attributes are hard to get on the components then they are not suitable for prediction. The first step to achieve predictability at the assembly level is to know the theory of the attribute on the component level. An exception is derived attributes that are a function of several other attributes at the component level. For instance concurrency or
deadlock free, where a component cannot be deadlock free by itself, it has to be deadlock free acting with other components.

We can conclude that attributes not that are depending on the system environment, usage profile are very hard to predict since it is hard to obtain and specify the system or usage information for the reasoning framework.

The classification of quality attributes could be developed further and to come up with analysis pattern and strategies for achieving predictability of the different attributes would be very valuable. Quality attributes in general software systems is being investigated and found hard, but using a component-based technology should ease the effort of reasoning about quality attributes. Component-models specifies the composition rules via interfaces and rules, by restricting and reusing rules it is possible to reuse reasoning - in many case it will be possible to automate the procedures.

Future work in this area is to do more investigation about 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 it would be of interest to classify attributes according to certain domains. 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 [3].

More research is needed to find out what kinds of component model restrictions are possible to introduce and which ones are impossible (or very difficult). This would give us guidelines that would ease the process of building prediction enabled component technologies.

Lessons learnt is that more research is needed in this area to be able to provide guidelines how to tackle different quality attributes from a prediction perspective. Reasoning frameworks or patterns for each attribute or class of attributes should be developed.
How can components provide information about their properties?

There are several ways that components can provide information about their properties and their values. For example, the component can have an extra interface that can be accessed to get the properties. To have the interface on the component implies that the component itself has the ownership and storage of the property data which might in some cases is very inflexible. Another example is to have a specification list separate from the component but that specifies the component properties. An analogy from the hardware industry is that physical properties are listed in a specification. A third possibility is to have configurable/flexible components that may include extra specifications in particular phases of their lifecycle (for example deployment), or which can be configured for particular component use.

The component model defines how components provide information about their quality attributes and future research should investigate the possibilities of creating or modifying component models for achieving PECT. It would be possible to make research studies for different component models to find out what approach should be taken to gain predictability.

More research is needed to see how introduction of predictability affect component models. For instance, consider evaluation of a component model to acquire more predictability, how does this effect maintainability of software systems? Will introduction of predictability with its restriction actually decrease maintainability? These questions are proposed future work.

A lesson learnt is that there are several ways to provide information about the properties, and depending on the problem domain, different approaches are appropriate. There are two main categories and the first is that components contain the property information and the second is that there is a specification, external to the component, of the component properties. With the first approach, it is easier to maintain consistency.
9 References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:


9. References:
10 Appendix

This appendix contains measurement data from the sample sets used in the controller experiment and a description of selected quality attributes.

Table 16: Recorded results of predicted and average measure latency for the first controller sample set

<table>
<thead>
<tr>
<th>Sample</th>
<th>Assembly</th>
<th>Task</th>
<th>Job</th>
<th>Predicted Latency (ms)</th>
<th>Measured Average Latency (ms)</th>
<th>MRE</th>
<th>Standard Deviation (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>12,157</td>
<td>12,181</td>
<td>-0,20%</td>
<td>0,0032</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>152,203</td>
<td>142,800</td>
<td>6,59%</td>
<td>0,0025</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>223,730</td>
<td>212,652</td>
<td>5,21%</td>
<td>0,0011</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>3</td>
<td>1</td>
<td>146,122</td>
<td>136,769</td>
<td>6,84%</td>
<td>0,0019</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>3</td>
<td>2</td>
<td>108,058</td>
<td>105,056</td>
<td>2,86%</td>
<td>0,0019</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>108,058</td>
<td>105,078</td>
<td>2,84%</td>
<td>0,0020</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>3</td>
<td>4</td>
<td>108,058</td>
<td>105,063</td>
<td>2,85%</td>
<td>0,0020</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>4</td>
<td>1</td>
<td>295,343</td>
<td>288,478</td>
<td>2,38%</td>
<td>0,0004</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>179,671</td>
<td>175,116</td>
<td>2,60%</td>
<td>0,0007</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>4</td>
<td>3</td>
<td>179,671</td>
<td>175,123</td>
<td>2,60%</td>
<td>0,0006</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>179,671</td>
<td>175,122</td>
<td>2,60%</td>
<td>0,0006</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>36,480</td>
<td>36,373</td>
<td>0,30%</td>
<td>0,0045</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>12,160</td>
<td>12,150</td>
<td>0,25%</td>
<td>0,0024</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>1</td>
<td>1</td>
<td>106,665</td>
<td>98,740</td>
<td>8,03%</td>
<td>0,0031</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>140,991</td>
<td>128,770</td>
<td>9,49%</td>
<td>0,0037</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>3</td>
<td>1</td>
<td>38,779</td>
<td>37,785</td>
<td>2,63%</td>
<td>0,0007</td>
</tr>
<tr>
<td>17</td>
<td>14</td>
<td>3</td>
<td>2</td>
<td>19,798</td>
<td>18,844</td>
<td>5,07%</td>
<td>0,0033</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>4</td>
<td>1</td>
<td>72,320</td>
<td>68,429</td>
<td>5,69%</td>
<td>0,0017</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>53,339</td>
<td>49,534</td>
<td>7,68%</td>
<td>0,0031</td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>48,077</td>
<td>47,991</td>
<td>0,18%</td>
<td>0,0020</td>
</tr>
<tr>
<td>21</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>6,078</td>
<td>6,090</td>
<td>-0,27%</td>
<td>0,0048</td>
</tr>
<tr>
<td>22</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td>6,078</td>
<td>6,088</td>
<td>-0,25%</td>
<td>0,0044</td>
</tr>
<tr>
<td>23</td>
<td>19</td>
<td>1</td>
<td>1</td>
<td>6,078</td>
<td>6,099</td>
<td>-0,37%</td>
<td>0,0036</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>593,455</td>
<td>634,291</td>
<td>-6,31%</td>
<td>0,0347</td>
</tr>
<tr>
<td>Sample</td>
<td>Assembly</td>
<td>Task</td>
<td>Job</td>
<td>Predicted Latency (ms)</td>
<td>Measured Average Latency (ms)</td>
<td>MRE</td>
<td>Standard Deviation (ms)</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>------</td>
<td>-----</td>
<td>------------------------</td>
<td>-------------------------------</td>
<td>-----</td>
<td>------------------------</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>995,948</td>
<td>995,237</td>
<td>0.07%</td>
<td>0.0004</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>754,928</td>
<td>754,640</td>
<td>0.04%</td>
<td>0.0003</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>553,665</td>
<td>460,407</td>
<td>20.26%</td>
<td>0.0002</td>
</tr>
<tr>
<td>28</td>
<td>21</td>
<td>1</td>
<td>1</td>
<td>6,078</td>
<td>6,086</td>
<td>-0.31%</td>
<td>0.0055</td>
</tr>
<tr>
<td>29</td>
<td>22</td>
<td>1</td>
<td>1</td>
<td>24,051</td>
<td>23,980</td>
<td>0.45%</td>
<td>0.0070</td>
</tr>
<tr>
<td>30</td>
<td>22</td>
<td>2</td>
<td>1</td>
<td>14,563</td>
<td>14,548</td>
<td>0.19%</td>
<td>0.0012</td>
</tr>
<tr>
<td>31</td>
<td>22</td>
<td>2</td>
<td>2</td>
<td>8,485</td>
<td>8,485</td>
<td>-0.38%</td>
<td>0.0041</td>
</tr>
<tr>
<td>32</td>
<td>23</td>
<td>1</td>
<td>1</td>
<td>72,517</td>
<td>72,180</td>
<td>0.47%</td>
<td>0.0027</td>
</tr>
<tr>
<td>33</td>
<td>23</td>
<td>2</td>
<td>1</td>
<td>41,100</td>
<td>41,004</td>
<td>0.27%</td>
<td>0.0011</td>
</tr>
<tr>
<td>34</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>65,423</td>
<td>62,560</td>
<td>0.09%</td>
<td>0.0016</td>
</tr>
<tr>
<td>35</td>
<td>24</td>
<td>2</td>
<td>1</td>
<td>6,079</td>
<td>6,101</td>
<td>-0.48%</td>
<td>0.0042</td>
</tr>
<tr>
<td>36</td>
<td>25</td>
<td>1</td>
<td>1</td>
<td>69,194</td>
<td>66,032</td>
<td>4.79%</td>
<td>0.0039</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>2</td>
<td>1</td>
<td>16,965</td>
<td>16,945</td>
<td>0.13%</td>
<td>0.0012</td>
</tr>
<tr>
<td>38</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>233,001</td>
<td>233,775</td>
<td>-6.30%</td>
<td>0.0582</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>2</td>
<td>1</td>
<td>304,572</td>
<td>279,491</td>
<td>10.18%</td>
<td>0.1280</td>
</tr>
<tr>
<td>40</td>
<td>27</td>
<td>3</td>
<td>1</td>
<td>445,338</td>
<td>429,370</td>
<td>3.72%</td>
<td>0.0002</td>
</tr>
<tr>
<td>41</td>
<td>27</td>
<td>3</td>
<td>2</td>
<td>140,767</td>
<td>137,264</td>
<td>2.55%</td>
<td>0.0001</td>
</tr>
<tr>
<td>42</td>
<td>27</td>
<td>4</td>
<td>1</td>
<td>161,388</td>
<td>155,648</td>
<td>3.69%</td>
<td>0.0004</td>
</tr>
<tr>
<td>43</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>12,156</td>
<td>12,105</td>
<td>0.78%</td>
<td>0.0136</td>
</tr>
<tr>
<td>44</td>
<td>28</td>
<td>2</td>
<td>1</td>
<td>31,941</td>
<td>31,872</td>
<td>0.24%</td>
<td>0.0009</td>
</tr>
<tr>
<td>45</td>
<td>28</td>
<td>2</td>
<td>2</td>
<td>25,863</td>
<td>25,798</td>
<td>0.29%</td>
<td>0.0015</td>
</tr>
<tr>
<td>46</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>12,157</td>
<td>12,191</td>
<td>-0.32%</td>
<td>0.0031</td>
</tr>
<tr>
<td>47</td>
<td>30</td>
<td>1</td>
<td>1</td>
<td>207,821</td>
<td>172,140</td>
<td>21.24%</td>
<td>0.0771</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>1</td>
<td>2</td>
<td>70,788</td>
<td>67,999</td>
<td>4.10%</td>
<td>0.0018</td>
</tr>
<tr>
<td>49</td>
<td>30</td>
<td>2</td>
<td>1</td>
<td>187,239</td>
<td>192,602</td>
<td>-5.95%</td>
<td>0.0263</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
<td>3</td>
<td>1</td>
<td>166,675</td>
<td>135,748</td>
<td>22.78%</td>
<td>0.0018</td>
</tr>
<tr>
<td>51</td>
<td>30</td>
<td>4</td>
<td>1</td>
<td>146,124</td>
<td>117,977</td>
<td>23.86%</td>
<td>0.0005</td>
</tr>
<tr>
<td>52</td>
<td>31</td>
<td>1</td>
<td>1</td>
<td>1087,457</td>
<td>1032,781</td>
<td>5.29%</td>
<td>0.0009</td>
</tr>
<tr>
<td>53</td>
<td>31</td>
<td>2</td>
<td>1</td>
<td>1036,500</td>
<td>788,620</td>
<td>31.43%</td>
<td>0.0009</td>
</tr>
<tr>
<td>54</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>6,078</td>
<td>6,092</td>
<td>-0.33%</td>
<td>0.0039</td>
</tr>
<tr>
<td>55</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>12,908</td>
<td>12,921</td>
<td>-0.17%</td>
<td>0.0024</td>
</tr>
<tr>
<td>56</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>20,555</td>
<td>20,531</td>
<td>0.24%</td>
<td>0.0024</td>
</tr>
<tr>
<td>57</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>20,555</td>
<td>20,507</td>
<td>0.33%</td>
<td>0.0024</td>
</tr>
<tr>
<td>58</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>20,555</td>
<td>20,519</td>
<td>0.28%</td>
<td>0.0024</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>20,555</td>
<td>20,510</td>
<td>0.33%</td>
<td>0.0025</td>
</tr>
<tr>
<td>60</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>20,555</td>
<td>20,515</td>
<td>0.29%</td>
<td>0.0025</td>
</tr>
<tr>
<td>61</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>48,552</td>
<td>48,502</td>
<td>0.14%</td>
<td>0.0008</td>
</tr>
<tr>
<td>62</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>58,063</td>
<td>58,013</td>
<td>0.10%</td>
<td>0.0006</td>
</tr>
<tr>
<td>63</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>37,516</td>
<td>37,452</td>
<td>0.20%</td>
<td>0.0009</td>
</tr>
<tr>
<td>64</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>64,903</td>
<td>64,928</td>
<td>-0.21%</td>
<td>0.0019</td>
</tr>
<tr>
<td>65</td>
<td>9</td>
<td>2</td>
<td>1</td>
<td>32,412</td>
<td>32,435</td>
<td>-0.14%</td>
<td>0.0017</td>
</tr>
</tbody>
</table>
Table 17: 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%  

Predicting Quality Attributes in Component-based Software Systems 219
## Appendix

### Predicting Quality Attributes in Component-based Software Systems

<table>
<thead>
<tr>
<th>Sample</th>
<th>Assembly</th>
<th>Task</th>
<th>Predicted AVG Task Latency (ms)</th>
<th>Measured AVG Task Latency (ms)</th>
<th>MRE</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>26</td>
<td>1</td>
<td>20,90</td>
<td>20,91</td>
<td>-0,05%</td>
</tr>
<tr>
<td>40</td>
<td>26</td>
<td>2</td>
<td>10,45</td>
<td>10,49</td>
<td>-0,36%</td>
</tr>
<tr>
<td>41</td>
<td>27</td>
<td>1</td>
<td>54,69</td>
<td>54,45</td>
<td>0,43%</td>
</tr>
<tr>
<td>42</td>
<td>27</td>
<td>2</td>
<td>14,14</td>
<td>14,14</td>
<td>0,00%</td>
</tr>
<tr>
<td>43</td>
<td>28</td>
<td>1</td>
<td>17,30</td>
<td>17,26</td>
<td>0,24%</td>
</tr>
<tr>
<td>44</td>
<td>28</td>
<td>2</td>
<td>22,87</td>
<td>22,62</td>
<td>1,08%</td>
</tr>
<tr>
<td>45</td>
<td>28</td>
<td>3</td>
<td>11,00</td>
<td>11,02</td>
<td>-0,18%</td>
</tr>
<tr>
<td>46</td>
<td>29</td>
<td>1</td>
<td>203,39</td>
<td>201,13</td>
<td>1,12%</td>
</tr>
<tr>
<td>47</td>
<td>29</td>
<td>2</td>
<td>55,10</td>
<td>53,49</td>
<td>3,01%</td>
</tr>
<tr>
<td>48</td>
<td>29</td>
<td>3</td>
<td>123,08</td>
<td>121,36</td>
<td>1,41%</td>
</tr>
<tr>
<td>49</td>
<td>29</td>
<td>4</td>
<td>17,56</td>
<td>16,25</td>
<td>8,08%</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
<td>1</td>
<td>15,87</td>
<td>15,85</td>
<td>0,09%</td>
</tr>
<tr>
<td>51</td>
<td>30</td>
<td>2</td>
<td>23,90</td>
<td>23,91</td>
<td>-0,05%</td>
</tr>
<tr>
<td>52</td>
<td>31</td>
<td>1</td>
<td>204,20</td>
<td>201,66</td>
<td>1,26%</td>
</tr>
<tr>
<td>53</td>
<td>31</td>
<td>2</td>
<td>344,02</td>
<td>338,83</td>
<td>1,53%</td>
</tr>
<tr>
<td>54</td>
<td>31</td>
<td>3</td>
<td>58,96</td>
<td>58,16</td>
<td>1,38%</td>
</tr>
<tr>
<td>55</td>
<td>31</td>
<td>4</td>
<td>48,19</td>
<td>47,49</td>
<td>1,47%</td>
</tr>
<tr>
<td>56</td>
<td>32</td>
<td>1</td>
<td>5,22</td>
<td>5,24</td>
<td>-0,29%</td>
</tr>
<tr>
<td>57</td>
<td>33</td>
<td>1</td>
<td>214,89</td>
<td>214,51</td>
<td>0,18%</td>
</tr>
<tr>
<td>58</td>
<td>33</td>
<td>2</td>
<td>34,43</td>
<td>34,45</td>
<td>-0,05%</td>
</tr>
<tr>
<td>59</td>
<td>33</td>
<td>3</td>
<td>88,24</td>
<td>88,22</td>
<td>0,02%</td>
</tr>
<tr>
<td>60</td>
<td>33</td>
<td>4</td>
<td>148,64</td>
<td>148,53</td>
<td>0,07%</td>
</tr>
<tr>
<td>61</td>
<td>34</td>
<td>1</td>
<td>10,45</td>
<td>10,46</td>
<td>-0,16%</td>
</tr>
<tr>
<td>62</td>
<td>35</td>
<td>1</td>
<td>7,07</td>
<td>7,09</td>
<td>-0,27%</td>
</tr>
<tr>
<td>63</td>
<td>35</td>
<td>2</td>
<td>13,22</td>
<td>13,21</td>
<td>0,06%</td>
</tr>
<tr>
<td>64</td>
<td>36</td>
<td>1</td>
<td>437,79</td>
<td>437,83</td>
<td>-0,01%</td>
</tr>
<tr>
<td>65</td>
<td>36</td>
<td>2</td>
<td>17,21</td>
<td>17,25</td>
<td>-0,23%</td>
</tr>
<tr>
<td>66</td>
<td>36</td>
<td>3</td>
<td>33,51</td>
<td>33,52</td>
<td>-0,05%</td>
</tr>
<tr>
<td>67</td>
<td>36</td>
<td>4</td>
<td>68,25</td>
<td>68,22</td>
<td>0,05%</td>
</tr>
<tr>
<td>68</td>
<td>37</td>
<td>1</td>
<td>10,45</td>
<td>10,47</td>
<td>-0,18%</td>
</tr>
<tr>
<td>69</td>
<td>38</td>
<td>1</td>
<td>202,29</td>
<td>202,42</td>
<td>-0,06%</td>
</tr>
<tr>
<td>70</td>
<td>38</td>
<td>2</td>
<td>229,65</td>
<td>229,39</td>
<td>0,11%</td>
</tr>
<tr>
<td>71</td>
<td>38</td>
<td>3</td>
<td>12,29</td>
<td>12,34</td>
<td>-0,35%</td>
</tr>
<tr>
<td>72</td>
<td>38</td>
<td>4</td>
<td>39,96</td>
<td>39,94</td>
<td>0,07%</td>
</tr>
<tr>
<td>73</td>
<td>39</td>
<td>1</td>
<td>10,45</td>
<td>10,46</td>
<td>-0,15%</td>
</tr>
<tr>
<td>74</td>
<td>40</td>
<td>1</td>
<td>7,99</td>
<td>8,01</td>
<td>-0,25%</td>
</tr>
<tr>
<td>75</td>
<td>40</td>
<td>2</td>
<td>8,81</td>
<td>8,82</td>
<td>-0,06%</td>
</tr>
<tr>
<td>76</td>
<td>41</td>
<td>1</td>
<td>222,58</td>
<td>222,66</td>
<td>-0,04%</td>
</tr>
<tr>
<td>77</td>
<td>41</td>
<td>2</td>
<td>98,37</td>
<td>98,30</td>
<td>0,07%</td>
</tr>
<tr>
<td>78</td>
<td>41</td>
<td>3</td>
<td>114,67</td>
<td>114,60</td>
<td>0,06%</td>
</tr>
<tr>
<td>Sample</td>
<td>Assembly</td>
<td>Task</td>
<td>Predicted AVG Task Latency (ms)</td>
<td>Measured AVG Task Latency (ms)</td>
<td>MRE</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>------</td>
<td>--------------------------------</td>
<td>-------------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>79</td>
<td>41</td>
<td>4</td>
<td>94,68</td>
<td>94,63</td>
<td>0,06%</td>
</tr>
<tr>
<td>80</td>
<td>42</td>
<td>1</td>
<td>10,45</td>
<td>10,47</td>
<td>-0,21%</td>
</tr>
<tr>
<td>81</td>
<td>43</td>
<td>1</td>
<td>10,64</td>
<td>10,65</td>
<td>-0,05%</td>
</tr>
<tr>
<td>82</td>
<td>44</td>
<td>1</td>
<td>229,52</td>
<td>228,74</td>
<td>0,34%</td>
</tr>
<tr>
<td>83</td>
<td>44</td>
<td>2</td>
<td>23,36</td>
<td>22,89</td>
<td>2,06%</td>
</tr>
<tr>
<td>84</td>
<td>44</td>
<td>3</td>
<td>31,35</td>
<td>31,34</td>
<td>0,04%</td>
</tr>
<tr>
<td>85</td>
<td>44</td>
<td>4</td>
<td>33,51</td>
<td>33,51</td>
<td>-0,02%</td>
</tr>
<tr>
<td>86</td>
<td>45</td>
<td>1</td>
<td>5,22</td>
<td>5,24</td>
<td>-0,32%</td>
</tr>
<tr>
<td>87</td>
<td>46</td>
<td>1</td>
<td>41,16</td>
<td>40,87</td>
<td>0,70%</td>
</tr>
<tr>
<td>88</td>
<td>46</td>
<td>2</td>
<td>26,82</td>
<td>26,70</td>
<td>0,47%</td>
</tr>
<tr>
<td>89</td>
<td>47</td>
<td>1</td>
<td>9,23</td>
<td>9,23</td>
<td>-0,01%</td>
</tr>
<tr>
<td>90</td>
<td>47</td>
<td>2</td>
<td>6,15</td>
<td>6,16</td>
<td>-0,21%</td>
</tr>
<tr>
<td>91</td>
<td>48</td>
<td>1</td>
<td>6,45</td>
<td>6,47</td>
<td>-0,22%</td>
</tr>
<tr>
<td>92</td>
<td>48</td>
<td>2</td>
<td>9,84</td>
<td>9,86</td>
<td>-0,18%</td>
</tr>
<tr>
<td>93</td>
<td>49</td>
<td>1</td>
<td>473,42</td>
<td>470,87</td>
<td>0,54%</td>
</tr>
<tr>
<td>94</td>
<td>49</td>
<td>2</td>
<td>5,22</td>
<td>5,25</td>
<td>-0,58%</td>
</tr>
<tr>
<td>95</td>
<td>49</td>
<td>3</td>
<td>6,97</td>
<td>6,98</td>
<td>-0,21%</td>
</tr>
<tr>
<td>96</td>
<td>49</td>
<td>4</td>
<td>9,58</td>
<td>9,58</td>
<td>-0,08%</td>
</tr>
<tr>
<td>97</td>
<td>50</td>
<td>1</td>
<td>85,14</td>
<td>84,68</td>
<td>0,54%</td>
</tr>
<tr>
<td>98</td>
<td>50</td>
<td>2</td>
<td>49,10</td>
<td>48,65</td>
<td>0,93%</td>
</tr>
<tr>
<td>99</td>
<td>50</td>
<td>3</td>
<td>90,37</td>
<td>89,44</td>
<td>1,04%</td>
</tr>
<tr>
<td>100</td>
<td>51</td>
<td>1</td>
<td>95,98</td>
<td>95,21</td>
<td>0,81%</td>
</tr>
<tr>
<td>101</td>
<td>51</td>
<td>2</td>
<td>5,22</td>
<td>5,25</td>
<td>-0,55%</td>
</tr>
<tr>
<td>102</td>
<td>51</td>
<td>3</td>
<td>26,31</td>
<td>26,27</td>
<td>0,18%</td>
</tr>
<tr>
<td>103</td>
<td>51</td>
<td>4</td>
<td>21,09</td>
<td>21,06</td>
<td>0,14%</td>
</tr>
<tr>
<td>104</td>
<td>52</td>
<td>1</td>
<td>362,96</td>
<td>363,22</td>
<td>-0,07%</td>
</tr>
<tr>
<td>105</td>
<td>52</td>
<td>2</td>
<td>343,90</td>
<td>343,66</td>
<td>0,07%</td>
</tr>
<tr>
<td>106</td>
<td>53</td>
<td>1</td>
<td>10,45</td>
<td>10,47</td>
<td>-0,18%</td>
</tr>
<tr>
<td>107</td>
<td>54</td>
<td>1</td>
<td>211,52</td>
<td>211,17</td>
<td>0,16%</td>
</tr>
<tr>
<td>108</td>
<td>54</td>
<td>2</td>
<td>233,03</td>
<td>232,76</td>
<td>0,12%</td>
</tr>
<tr>
<td>109</td>
<td>54</td>
<td>3</td>
<td>53,19</td>
<td>53,15</td>
<td>0,07%</td>
</tr>
<tr>
<td>110</td>
<td>54</td>
<td>4</td>
<td>27,98</td>
<td>28,03</td>
<td>-0,18%</td>
</tr>
<tr>
<td>111</td>
<td>55</td>
<td>1</td>
<td>10,45</td>
<td>10,48</td>
<td>-0,27%</td>
</tr>
<tr>
<td>112</td>
<td>56</td>
<td>1</td>
<td>15,87</td>
<td>15,90</td>
<td>-0,21%</td>
</tr>
<tr>
<td>113</td>
<td>57</td>
<td>1</td>
<td>194,01</td>
<td>192,85</td>
<td>0,60%</td>
</tr>
<tr>
<td>114</td>
<td>57</td>
<td>2</td>
<td>26,90</td>
<td>26,93</td>
<td>-0,10%</td>
</tr>
<tr>
<td>115</td>
<td>57</td>
<td>3</td>
<td>129,44</td>
<td>128,48</td>
<td>0,74%</td>
</tr>
<tr>
<td>116</td>
<td>57</td>
<td>4</td>
<td>148,50</td>
<td>148,04</td>
<td>0,31%</td>
</tr>
<tr>
<td>117</td>
<td>58</td>
<td>1</td>
<td>10,45</td>
<td>10,46</td>
<td>-0,13%</td>
</tr>
<tr>
<td>118</td>
<td>59</td>
<td>1</td>
<td>38,63</td>
<td>38,48</td>
<td>0,38%</td>
</tr>
<tr>
<td>Sample</td>
<td>Assembly</td>
<td>Task</td>
<td>Predicted AVG Task Latency (ms)</td>
<td>Measured AVG Task Latency (ms)</td>
<td>MRE</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>119</td>
<td>59</td>
<td>2</td>
<td>15,25</td>
<td>15,27</td>
<td>-0,09%</td>
</tr>
<tr>
<td>120</td>
<td>60</td>
<td>1</td>
<td>9,84</td>
<td>9,85</td>
<td>-0,12%</td>
</tr>
<tr>
<td>121</td>
<td>60</td>
<td>2</td>
<td>11,99</td>
<td>11,99</td>
<td>-0,03%</td>
</tr>
<tr>
<td>122</td>
<td>61</td>
<td>1</td>
<td>6,15</td>
<td>6,17</td>
<td>-0,28%</td>
</tr>
<tr>
<td>123</td>
<td>61</td>
<td>2</td>
<td>8,55</td>
<td>8,57</td>
<td>-0,20%</td>
</tr>
<tr>
<td>124</td>
<td>62</td>
<td>1</td>
<td>189,48</td>
<td>189,41</td>
<td>0,04%</td>
</tr>
<tr>
<td>125</td>
<td>62</td>
<td>2</td>
<td>132,19</td>
<td>132,13</td>
<td>0,05%</td>
</tr>
<tr>
<td>126</td>
<td>62</td>
<td>3</td>
<td>123,17</td>
<td>123,09</td>
<td>0,07%</td>
</tr>
<tr>
<td>127</td>
<td>62</td>
<td>4</td>
<td>25,42</td>
<td>25,46</td>
<td>-0,15%</td>
</tr>
<tr>
<td>128</td>
<td>63</td>
<td>1</td>
<td>237,92</td>
<td>235,61</td>
<td>0,98%</td>
</tr>
<tr>
<td>129</td>
<td>63</td>
<td>2</td>
<td>62,94</td>
<td>62,05</td>
<td>1,44%</td>
</tr>
<tr>
<td>130</td>
<td>63</td>
<td>3</td>
<td>5,22</td>
<td>5,25</td>
<td>-0,53%</td>
</tr>
<tr>
<td>131</td>
<td>63</td>
<td>4</td>
<td>6,53</td>
<td>6,55</td>
<td>-0,29%</td>
</tr>
<tr>
<td>132</td>
<td>64</td>
<td>1</td>
<td>31,54</td>
<td>31,55</td>
<td>-0,03%</td>
</tr>
<tr>
<td>133</td>
<td>64</td>
<td>2</td>
<td>10,45</td>
<td>10,48</td>
<td>-0,35%</td>
</tr>
<tr>
<td>134</td>
<td>65</td>
<td>1</td>
<td>172,34</td>
<td>171,93</td>
<td>0,24%</td>
</tr>
<tr>
<td>135</td>
<td>65</td>
<td>2</td>
<td>38,12</td>
<td>38,17</td>
<td>-0,11%</td>
</tr>
<tr>
<td>136</td>
<td>66</td>
<td>1</td>
<td>10,45</td>
<td>10,47</td>
<td>-0,26%</td>
</tr>
<tr>
<td>137</td>
<td>67</td>
<td>1</td>
<td>171,85</td>
<td>171,95</td>
<td>-0,06%</td>
</tr>
<tr>
<td>138</td>
<td>67</td>
<td>2</td>
<td>217,35</td>
<td>217,20</td>
<td>0,07%</td>
</tr>
<tr>
<td>139</td>
<td>67</td>
<td>3</td>
<td>41,20</td>
<td>41,23</td>
<td>-0,07%</td>
</tr>
<tr>
<td>140</td>
<td>67</td>
<td>4</td>
<td>215,51</td>
<td>215,33</td>
<td>0,08%</td>
</tr>
<tr>
<td>141</td>
<td>68</td>
<td>1</td>
<td>10,64</td>
<td>10,65</td>
<td>-0,08%</td>
</tr>
<tr>
<td>142</td>
<td>69</td>
<td>1</td>
<td>10,64</td>
<td>10,65</td>
<td>-0,07%</td>
</tr>
<tr>
<td>143</td>
<td>70</td>
<td>1</td>
<td>32,55</td>
<td>32,34</td>
<td>0,68%</td>
</tr>
<tr>
<td>144</td>
<td>70</td>
<td>2</td>
<td>14,09</td>
<td>14,00</td>
<td>0,68%</td>
</tr>
<tr>
<td>145</td>
<td>71</td>
<td>1</td>
<td>396,22</td>
<td>393,78</td>
<td>0,62%</td>
</tr>
<tr>
<td>146</td>
<td>71</td>
<td>2</td>
<td>436,70</td>
<td>433,99</td>
<td>0,62%</td>
</tr>
<tr>
<td>147</td>
<td>71</td>
<td>3</td>
<td>219,27</td>
<td>217,94</td>
<td>0,61%</td>
</tr>
<tr>
<td>148</td>
<td>71</td>
<td>4</td>
<td>142,90</td>
<td>142,25</td>
<td>0,46%</td>
</tr>
<tr>
<td>149</td>
<td>72</td>
<td>1</td>
<td>10,64</td>
<td>10,65</td>
<td>-0,07%</td>
</tr>
<tr>
<td>150</td>
<td>73</td>
<td>1</td>
<td>10,64</td>
<td>10,65</td>
<td>-0,06%</td>
</tr>
<tr>
<td>151</td>
<td>74</td>
<td>1</td>
<td>32,55</td>
<td>32,34</td>
<td>0,67%</td>
</tr>
<tr>
<td>152</td>
<td>74</td>
<td>2</td>
<td>14,09</td>
<td>14,15</td>
<td>-0,41%</td>
</tr>
<tr>
<td>153</td>
<td>75</td>
<td>1</td>
<td>396,22</td>
<td>393,63</td>
<td>0,66%</td>
</tr>
<tr>
<td>154</td>
<td>75</td>
<td>2</td>
<td>436,70</td>
<td>433,99</td>
<td>0,62%</td>
</tr>
<tr>
<td>155</td>
<td>75</td>
<td>3</td>
<td>219,27</td>
<td>237,63</td>
<td>-7,72%</td>
</tr>
<tr>
<td>156</td>
<td>75</td>
<td>4</td>
<td>142,90</td>
<td>142,25</td>
<td>0,46%</td>
</tr>
</tbody>
</table>
Table 18: Short description of selected quality attributes

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Quality Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>Accessability</td>
<td>The ability to be accessible</td>
</tr>
<tr>
<td></td>
<td>Administrability</td>
<td>The ability to be easily administrated (data are easy to administrate)</td>
</tr>
<tr>
<td></td>
<td>Understandability</td>
<td>The degree of easiness to understand (the function, protocol, interaction)</td>
</tr>
<tr>
<td></td>
<td>Generality</td>
<td>The breadth of potential application of program components.</td>
</tr>
<tr>
<td></td>
<td>Operability</td>
<td>The ease of operation of a program.</td>
</tr>
<tr>
<td></td>
<td>Simplicity</td>
<td>The degree to which a program can be understood without difficulty.</td>
</tr>
<tr>
<td>Mobility</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Portability</td>
<td>Nomadicty</td>
<td>The ability to move operation between different nodes</td>
</tr>
<tr>
<td></td>
<td>Hardware independence</td>
<td>The degree to which the software is decoupled from the hardware on which it operates.</td>
</tr>
<tr>
<td></td>
<td>Software system independence</td>
<td>The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints.</td>
</tr>
<tr>
<td>Performance</td>
<td>Accuracy</td>
<td>The precision of computations and control.</td>
</tr>
<tr>
<td></td>
<td>Footprint</td>
<td>Size of memory used</td>
</tr>
<tr>
<td></td>
<td>Responsiveness</td>
<td>Time to response to an input</td>
</tr>
<tr>
<td></td>
<td>Scalability</td>
<td>The ability of a system to support modifications that dramatically increase the size of the system.</td>
</tr>
<tr>
<td></td>
<td>Schedulability</td>
<td>Ability to schedule an operation</td>
</tr>
<tr>
<td></td>
<td>Timeliness</td>
<td>The ability to perform a task at the correct time</td>
</tr>
<tr>
<td></td>
<td>CPU utilization</td>
<td>The percentage of CPU execution time</td>
</tr>
<tr>
<td></td>
<td>Latency</td>
<td>Time between input and output</td>
</tr>
<tr>
<td></td>
<td>Transaction Throughput</td>
<td>Number of transaction processed per unit of time</td>
</tr>
<tr>
<td></td>
<td>Concurrency</td>
<td>Ability to perform operations in parallel</td>
</tr>
</tbody>
</table>
### 10. Appendix

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Quality Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>The amount of computing resources and code required by a program to perform its function.</td>
<td></td>
</tr>
<tr>
<td>Flexibility</td>
<td>The effort required to modify an operational program, or, the ease with which the systems can be adapted to changes.</td>
<td></td>
</tr>
<tr>
<td>Evolvability</td>
<td>The ability to continuously being changed</td>
<td></td>
</tr>
<tr>
<td>Extensibility</td>
<td>The ability for adding new functionality</td>
<td></td>
</tr>
<tr>
<td>Modifiability</td>
<td>The ability of a system to be extended to accomplish additional functionality</td>
<td></td>
</tr>
<tr>
<td>Upgradeability</td>
<td>The ability of a system to be updated</td>
<td></td>
</tr>
<tr>
<td>Expandability</td>
<td>The degree to which architectural, data, or procedural design can be extended.</td>
<td></td>
</tr>
<tr>
<td>Data consistency</td>
<td>The absence of contradictory data in the system. The use of uniform design and documentation techniques throughout the software development project.</td>
<td></td>
</tr>
<tr>
<td>Version consistency</td>
<td>Ability of avoiding version mismatch in a configuration</td>
<td></td>
</tr>
<tr>
<td>Adaptability</td>
<td>The ability for adaptation to new requirements or new environment or similar</td>
<td></td>
</tr>
<tr>
<td>Composeability</td>
<td>The ability to be integrated with the parts of a system</td>
<td></td>
</tr>
<tr>
<td>Interoperability</td>
<td>The ability of a system to work with another system</td>
<td></td>
</tr>
<tr>
<td>Openness</td>
<td>Ability to integrate new functions developed by third party</td>
<td></td>
</tr>
<tr>
<td>Heterogenity</td>
<td>Ability to integrated heterogeneous elements (software and hardware)</td>
<td></td>
</tr>
<tr>
<td>Integrability</td>
<td>The ability to make the separately developed components of a system work correctly together</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>Audibility</td>
<td>The ease with which conformance to standards can be checked</td>
</tr>
<tr>
<td>Completeness</td>
<td>The degree to which full implementation of required function has been achieved</td>
<td></td>
</tr>
<tr>
<td>Conciseness</td>
<td>The compactness of the program in terms of lines of code.</td>
<td></td>
</tr>
<tr>
<td>Concerns</td>
<td>Quality Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Correctness</td>
<td>The extent to which a program satisfies its specification and fulfills the customer's mission objectives.</td>
<td></td>
</tr>
<tr>
<td>Testability</td>
<td>The effort required to test a program to ensure that it performs its intended function.</td>
<td></td>
</tr>
<tr>
<td>Traceability</td>
<td>The ability to trace a design representation or actual program component back to requirements.</td>
<td></td>
</tr>
<tr>
<td>Coherence</td>
<td>The degree of consistency in the design.</td>
<td></td>
</tr>
<tr>
<td>Analyzability</td>
<td>The ability to analyze particular properties.</td>
<td></td>
</tr>
<tr>
<td>modularity</td>
<td>The functional independence of program components.</td>
<td></td>
</tr>
<tr>
<td>Reusability</td>
<td>The extent to which a program [or parts of a program] can be reused.</td>
<td></td>
</tr>
<tr>
<td>Configureability</td>
<td>The ability to configure the artifact during deployment.</td>
<td></td>
</tr>
<tr>
<td>Distributeability</td>
<td>The ability to distribute the artifact.</td>
<td></td>
</tr>
<tr>
<td>Ease of creation</td>
<td>The easiness of constructing the system. Often measured in labor hours.</td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td>The probability that the system functions correctly.</td>
<td></td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Absence of unauthorized disclosure of information.</td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td>The extent to which access to software or data by unauthorized persons can be controlled.</td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td>The effort required to locate and fix an error in a program (this is a very limited definition).</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>The ability of the system to sustain operations. A common measure is mean time between failures.</td>
<td></td>
</tr>
<tr>
<td>Safety</td>
<td>Absence of catastrophic consequences on the users and the environment.</td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>The availability of mechanisms that control of protect programs and data.</td>
<td></td>
</tr>
<tr>
<td>Business</td>
<td>Cost</td>
<td></td>
</tr>
<tr>
<td>Concerns</td>
<td>Quality Attribute</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Projected Lifetime</td>
<td></td>
<td>Depending on the lifetime of the product, scalability, maintainability and portability becomes important</td>
</tr>
<tr>
<td>Targeted Market</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time to market</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Affordability</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>