IMPERFECT INFORMATION IN SOFTWARE DESIGN PROCESSES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W. H. M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 5 juli 2007 om 15.00 uur

door
Johannes Albertus Rudolf Noppen
geboren op 26 februari 1978
te Doetinchem

Dit proefschrift is goedgekeurd door:

Prof. Dr. Ir. M. Aksit (promotor)
Dr. P. M. van den Broek (assistent-promotor)

Imperfect Information in Software Design Processes

Joost Noppen

Dissertation Committee:

Prof. Dr. Ir. A. J. Mouthaan

Prof. Dr. Ir. M. Aksit (promotor, University of Twente)
Dr. P. M. van den Broek (assistant-promotor, University of Twente)

Prof. Dr. H. Brinksma (University of Twente)
Dr. P. C. Clements (Carnegie Mellon University)
Prof. Dr. A. Finkelstein (University College London)
Prof. Dr. W. Pedrycz (University of Alberta)
Prof. Dr. M. A. Rashid (University of Lancaster)
Dr. Ir. A. Rensink (University of Twente)

Joost Noppen

Imperfect Information in Software Design Processes
PhD Thesis, University of Twente, 2007
ISBN 978-90-365-2511-4

IPA Dissertation Series 2007-11
CTIT PhD Thesis Series (ISSN 1381-3617) 07-99

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and within the context of the Centre for Telematics and Information Technology (CTIT)

Printed by Gildeprint BV, Enschede, the Netherlands
Cover design by Lotte Nijkamp

IMPERFECT INFORMATION IN SOFTWARE DESIGN PROCESSES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W. H. M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 5 juli 2007 om 15.00 uur

door
Johannes Albertus Rudolf Noppen
geboren op 26 februari 1978
te Doetinchem

Dit proefschrift is goedgekeurd door:

Prof. Dr. Ir. M. Aksit (promotor)
Dr. P. M. van den Broek (assistent-promotor)

In loving memory of

Albert Oude Nijhuis

and

Ben Wubbels

I dearly wish you would have been able to see me complete this work.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

During secondary education, most dutch children of my age have taken a “Algemene Oriëntatie Beroepskeuze“-test. This test assesses and evaluates which professions are possible and lie within the area of interest of each child. While I distinctly recall thinking the results of the test not to be too interesting and relevant, I had forgotten the actual results of the test. However, when I recently stumbled upon the report of the test, I was amazed to see it stated that I had a distinct preference for scientific research activities in a technical and theoretical area. Little did the people who devised the test, and even less so me, know that some 15 years later I would be writing the acknowledgements of my Ph.D. thesis on a plane back from a conference in Norway. Looking back, I can only say that while not every step was planned to achieve this goal, I am convinced this is the best choice I could have made. The people I have met, the places I have seen and, perhaps most importantly, the things I have learned have made this period an incredible experience.

The possibility to pursue a Ph.D. was offered to me by Mehmet Aksit and I am tremendously grateful for this chance. Over the years, I have gotten to know Mehmet as an inspired researcher, always full of ideas and always more than willing to discuss research-related as well as non-research related topics. Mehmet’s guidance was always aimed at challenging yourself and becoming more than you are, a state of mind I hope to keep in the future. My other major influence during this period is my daily supervisor, Pim van den Broek. With his background in physics, I always tend to see Pim as a proper scientist. He taught me to always be very meticulous about the steps I am taking in my research, and I admire his persistence in finding the right formulas and specifying them correctly. His door was always open for a short discussion on research as well as mutual interests such sports or even quantum mechanics. I feel that the supervision provided by Mehmet and Pim was a very potent mix, and our cooperation stretched beyond the boundaries of our research, as was underlined when we visited the workshop on the origin of the universe out of general interest. I sincerely hope in the future this will remain.

I would like to thank my committee members, Ed Brinksma, Paul Clements, Anthony Finkelstein, Witold Pedrycz, Awais Rashid and Arend Rensink for participating in the committee. I very much value the effort taken to read the concept version and provide feedback for its improvement. I am honored that you have allocated time and traveled long distances to participate in my defense. I would also like to thank my colleagues of the fifth floor and some from other floors (in no particular order): Lodewijk, Klaas, Boris, Celim, Ali, Marcos, Piotr, Axel, André, Tom, Pascal, Wilke, Gurcan, Theo, Rom, Wouter, Hicham, Iovka, Val, Tomas, Dino, Rik, Patrick, Pablo, Henrik, Suzana, and Arda. You have created an inspiring research environment as well as a very friendly atmosphere. There was always great willingness for discussions on a plethora of topics, with the relevance rapidly dropping off during the notorious BOCOM, every Friday at five o‘clock.

I am very thankful to our secretaries, Ellen and Joke. I am convinced that for any research group, the secretaries are the most important people next to obviously the professor. Ellen and Joke have made this clear to me by always helping me out on any subject for which I required assistance. You have always been there for the serious as well as the no-so-serious time. Then, there are a number of people that had to endure me on a frequent basis: thanks to Harmen for the fitness, Mariëlle and Laura for the bbq-organizing fun, Anne for the talks over coffee, Ivan for the guidance during the writing process, Georgios for the BOCOM and dodgy pub crawls and Arend for all the crazy events such as a 13-hour Star Wars marathon.

In these acknowledgements I reserve a special place for my roommates over the years, Bedir, Christian, Hasan, Ismenia, Istvan and Machiel. You have had the worst of me over the years, since you had to cope with a noisy colleague every day. We have had so many nice discus-

sions, shared so many jokes and have done so many fun things that you all have become special to me. You have made it easy for me to go to work and sometimes difficult to go home. I am positive we will stay in touch in the future, in one way or the other.

I am also thankful for all the support I have received from my friends. I would like to mention in particular Eelco, Jeroen and Joost for the fun, beer drinking nights and excursions to the Oktoberfest. There are more stories there than the world could handle. I would like to thank Luitzen and Tom for their ever present enthusiasm and support. In the end, you both have beaten me in the completion of our respective work. I also want to thank Lotte for all the support over the years. You have been there during the good and the bad times, and in addition you have designed the beautiful cover of the thesis. I am proud you agreed to be one of my paranimphs. Also I would like to thank Dennis, Frank and Mark for the technical discussions and fun during the dreaded “ISpec“ meetings. I sincerely hope we never start a foundation again. And I would like to thank Marina and Sander for being such good friends. Your constant interest and support is very much appreciated and I have so much enjoyed all the things we did over the years, from birthdays and moving to refereeing at the European championship and the Enschede Open. I am sure there is much more to come.

Finally, I would like to thank my family for their never-ending support and interest, even while the topic of my research was perhaps too technical for most. I would like to express special thanks to Rieneke, Jay and Brechje, for being there for me during all these years. For all the fun and laughs during the holidays and other times, and also for the confidence you had in me. And last, but certainly not least, I would like to thank my father and mother. You have always supported me in my choices and believed in me in times of doubt. You have given me the opportunity, preparation, confidence and tools to complete this work, and for this I consider myself very lucky. I am so pleased we can all celebrate the completion of this work together.

All in all, I will use a single sentence in my regional dialect, Twents, to express my thanks to everyone who has helped me over the years: *Leu, onmeunig bedankt!!*

ABSTRACT

The process of designing high-quality software systems is one of the major issues in software engineering research. Over the years, this has resulted in numerous design methods, each with specific qualities and drawbacks. For example, the Rational Unified Process is a comprehensive design process, which is proposed to support the major phases in the software engineering life-cycle. Agile processes, like for instance Extreme Programming, aim at flexibility, since the design steps are not defined rigidly. Although the current software methods have largely proven their applicability and there exists a plethora of different design processes to be used, the current methods naturally suffer from the existence of imperfect information.

Imperfection during software design is the occurrence of information, which is uncertain or incomplete to a certain degree. This can have many different causes, such as for instance incomplete information sources or an imprecise view of what the system should do. The existence of imperfection makes the design processes difficult to apply, since such information typically has one or more elements that are ambiguous in their interpretation. When a system is designed by using only one of the possible interpretations, there is a risk that the interpretation turns out to be wrong in due time. This can lead to redesigning the system, and consequently to very high costs.

Unfortunately, current design methods neglect the existence of imperfect information. For example, modern software design processes explicitly require crisp and, preferably, complete requirement specifications, although it is generally acknowledged that this is difficult to accomplish. Similarly, imperfect information during design activities is also commonly neglected. However, imperfection is an inherent problem of almost every design process as well. Rather than trying to model the imperfection that inevitably exists, imperfection is generally neglected by making explicit and crisp assumptions, which may eventually result in wrong design decisions. The impact of neglecting imperfection in requirement specifications or design activities is higher in the early phases of software development. As the design process progresses, both the software engineers and the stakeholders may come to have new insights about what the system is supposed to do and which steps are to be taken. Because this information only becomes gradually available along with the design process, in the early phases it is not possible to transform imperfect information into a perfect one.

In this thesis, we identify the problems in the two areas in which imperfect information can manifest itself, namely in contextual information (predominantly requirement specifications) and in software design activities. We analyze the types of imperfection, and the way in which the imperfection should be interpreted. Based on this analysis, we propose generic extensions to software design processes for modeling imperfect information. By this way, different interpretations of a design choice may be captured and considered according to their appropriateness without committing to one of them too early. By modeling alternative interpretations of a design choice, the design becomes more flexible. This is because the new insights which are gained during the software development process can be taken into account in the decision process more conveniently, without a need to redesign the system.

In our proposed method we have combined the techniques used in various disciplines such as software architecture design, probability theory and fuzzy set theory, to ensure that we capture the relevant properties of both the software engineering process as well as the nature of the imperfect information. For a real-world application our approach can become very labor-intensive. In order to aid the software engineer, tooling support is considered essential. For this purpose the proposed methods have been implemented in a prototype. The tools relieve the user from the mathematical computation and optimization effort, and ensure that the user only has to focus on providing the relevant input. The effectiveness and ease of use of the tools are evaluated by means of a pilot study.

CHAPTER

1

"I am a theater of processes, he told himself. I am a prey to the imperfect vision, to the race consciousness and its terrible purpose."

- From Frank Herbert's Dune [Herbert2005]

INTRODUCTION

1.1 Introduction

Nowadays, there exists consensus among the software engineering community that designing even a medium size software system is a complex task [Lethbridge2005] [Pfleeger1998] [Pressman1997]. There are many causes for this, such as inherent complexity of the problems to be solved, ambiguous and evolving requirements, difficulty of taking the right design decision at the right time, and so on. Although there may be also specific reasons why software design projects do not accomplish their original goals, coping with *imperfect information* is a common problem of all projects and possibly the origin of many practical failures. Despite its importance, the imperfect information problem has not been studied in the software engineering literature satisfactorily. In this chapter, we introduce the problem of imperfect information in software engineering and we define a generic approach to resolve this problem.

1.2 Problem Statement

In this thesis we focus on an important problem during software development: the existence of imperfect information in design processes. The difficulty of defining or attaining unambiguous information at the time it is needed, can be encountered in many stages of software development. In addition, the consequences for software systems and design processes can range from considerable refactoring to complete redesign, in the case that imperfect information is not recognized and considered accordingly.

The design of software systems is a very complex activity, which requires a considerable amount of information to be done effectively. Ideally, the required information must be of perfect quality, i.e. clear and accurate, since it will be used in decisions that determine the design of the system. However, in practice it has proven to be very difficult to define or attain accurate information when it is required. As a result, the design activities generally are performed with descriptions that only partially provide the information with the desired quality. The usefulness of information can therefore be limited as a result of ambiguities, incompleteness and vagueness among others. We refer to such information as *imperfect information*.

An imperfect description of what is expected from a software system is fairly common during software design. In the case that the variability of the context in which the system will be used is known, the design can be fitted with mechanisms to handle the variability. For example, most modern web browsers accommodate a plug-in mechanism to facilitate the various media standards that are available in the market. However, this approach can only be used when the extent to which a particular system part can vary is known. If this is not the case, the variability can turn out to stretch beyond the capabilities of the implemented mechanisms. As a result, for software development it is important that the requirement specification accurately describes the expected behavior and capabilities of the software system to be designed.

Most software development methods acknowledge the difficulty of defining requirement specifications with desired quality, and indicate advice to minimize the consequences of imperfect information. Generally, it is advised to specify requirements of the software system as precise and complete as possible. However, the failure of the waterfall model as a practical development method has demonstrated that this it is very difficult to come to such specifications. In response to this failure, software development methods were extended with refinement cycles, an approach that allowed software engineers to revisit and adjust designs in an iterative manner. Nowadays, most modern software design processes, such as Rational Unified Process [Jacobson1999], incorporate iterative mechanisms. Agile processes, such as Extreme Programming [Poppendieck2003], have made the iterative cycle an integral part of their software development process.

However, the negative consequences of dealing with imperfect information are still noticeable, even in iterative methods. Requirements generally include assumptions, which can not be precisely verified at the time of their definition. For example, there may be requirements derived from market estimations, user expectance and satisfaction, etcetera. We term this as *imperfect requirements*; requirements that are affected by imperfect information. Obviously, requirements have direct impact on the designs that are defined at the subsequent phases. Quite naturally, imperfect requirements may result in imperfect designs. If during the design step the software engineer happens to obtain more information on the assumptions that he previously had to make, he may iterate to adjust the original requirement and carry out the design steps again. In the ideal case, this may eliminate the problems created by imperfect information. However, in practice it is likely that this can not be accomplished for the following reasons:

1. For a non-trivial system, it is considered unrealistic to assume that imperfect information as a whole can become perfect at the same design moment.
2. For a non-trivial system, it is considered unrealistic to assume that requirements are frozen and will not change.

3 The corrective action may have undesired side-effects.

4 There are generally time and financial restrictions that constrain the number of iterations in design.

Moreover, imperfect information is not limited to the definition of requirement specifications. During the design process, the software engineer makes assumptions on the result of the design process. Typical assumptions are made on for example performance, adaptability or user satisfaction. When the design is mature, the validity of these assumptions can be assessed. If the system is not satisfactory at this moment, the design process can be partially repeated. In the ideal case, this will eliminate the problems caused by wrong assumptions on the final design. However, this is difficult to accomplish, because:

1 The corrective action is also likely to be based on some assumptions on the result of the design process.

2 It is likely that new requirements will be imposed during the design, which are expected to be influenced by imperfect information.

3 The corrective action may have undesired side-effects.

4 There are generally time and financial restrictions that constrain the number of iterations in design.

We conclude that imperfection is an inherent property of the information used during software development, even when it is not always recognized as such. In the previous paragraphs, we have identified four problems for resolving imperfect requirements. These four problems apply equally to resolving other imperfect information during software development. From the problems we have identified, it can be seen that the success of iterative design with respect to correcting the consequences of imperfect information depends on whether the following four requirements are fulfilled:

All-Isolation Requirement: For the design to be corrected effectively with iterative design, all concerns must be orthogonal; otherwise concerns that are influenced by imperfect information cannot be isolated and made perfect eventually.

All-Always Requirement: For the design to be corrected effectively with iterative design, all requirements must be frozen; All requirements must always stay the same.

All-at-Once Requirement: For the design to be corrected effectively with iterative design, all unknowns of a dependent design part must be resolved at the same time, otherwise there will be always some influence of imperfect information

All-Infinite Requirement: For the design to be corrected effectively with iterative design, all the required resources must be infinitely available, otherwise iterations cannot be applied until the imperfections are resolved.

Naturally, these requirements will not be fulfilled in a realistic setting, which means design processes can not rely only on iteration to ensure the timely delivery of systems with acceptable quality. While imperfection is not necessarily problematic during software design, development methods need to become aware of the imperfection that exists in information that is used. In addition, development methods need to understand the nature of the imperfection, such as conflict, ambiguity or tolerance, since this directly influences the way in which the information should be used.

1.3 Approach

To understand the problems caused by imperfect information in software development methods, we study the nature of imperfection as well as sources and locations from which imperfect

information originates. We propose to explicitly model and consider imperfection during software development activities. We define models that capture the relevant properties based on the character of the imperfection, which are based on *probability theory* and *fuzzy set theory*.

We study the way in which design activities must be extended to make software development methods capable of working with information that contains imperfection models. To facilitate the compatibility with modern software design processes, we propose three reasoning approaches, that can evaluate imperfect information in a similar manner to perfect information: the *design tree model*, the *artifact trace model* and the *resource allocation model*. The mathematical basis underlying these models ensures the evaluation of imperfect information is performed in accordance with the models that are used to capture it.

To be able to perform software development activities uniformly and correctly, we study the common operations that are (implicitly) performed during the course of a software development process. We propose extensions to these common operations to ensure the uniform treatment of both perfect and imperfect information, without losing the added value of imperfection models during software development. In addition, we use the extra information in the imperfection models and the tracing models to provide feedback to the development process on how to proceed with incremental steps of the design.

Finally, to explore and evaluate the applicability of the techniques we propose, we perform a pilot study based on the example cases described in this thesis. This is done by use of a toolset that implements both the imperfection models and the optimization models that have been proposed. The novelty of the approach lies in the possibility to model and analyze information that does not contain the level of detail that is desirable. By including the nature of the imperfection in the decisions that are taken, the risk of not having the desired information can be assessed and the development process can be adjusted to minimize the risk.

1.4 Contributions

In this thesis we describe the following contributions:

1. *The concept of fuzzy functional requirements that enables the inclusion of alternative requirement interpretations in software design processes.*

Chapter 3 identifies the danger of committing to a single interpretation when a software engineer is faced with imperfect functional requirements. An extension of crisp functional requirements is proposed, which the inclusion of alternative interpretations by means of fuzzy sets. The extension enables the inclusion multiple interpretations of a single imperfect requirement, and addresses these interpretations as normal requirements in the development process. The fuzzy requirement concept is combined with the Artifact Trace Model to support the analysis of the resulting design based on different trade-offs, such as cost minimization or relevance maximization.

2. *An extension to numerical expressions in quality requirements and estimations that enables the specification and evaluation of imperfect information in design decisions.*

Chapter 4 presents an approach for specifying numerical expressions in quality requirements and quality estimations that are subject to imperfection, by means of probability distributions and fuzzy sets. This approach overcomes the inability of current methods to capture imperfection in such numerical expressions, which enables the software engineer to model for example tolerance in quality requirements and partial knowledge on the expected behavior of the completed system. The approach is completed with the definition of comparison operators that are needed to evaluate imperfect estimations with imperfect requirements. Combined with the

Design Tree Model, the approach forms a decision support model for design decisions with support for imperfect information.

3. Reasoning and optimization models that ensure the proper usage of the proposed imperfection models and explore the added possibilities that are offered by the imperfection.

In chapters 3 and 4, two tracing approaches are presented that are used to provide decision support for the software engineer during the software development process. The first model traces functional requirements to their respective components. In addition, it provides optimization operations that enable the software engineer to make trade-offs between the provided functionality and particular stakeholder interests. The second model captures the sequence of the design decisions that have been taken, and the alternatives that have been considered. By means of configurable design strategies, the software engineer is provided with advice on how to continue the software development process.

4. An optimization model for scheduling the implementation order of an application framework and product line components with respect to probabilistic changes in market demands.

Chapter 5 motivates the explicit consideration of probabilistic changes in market demands during scheduling of implementation activities in software design processes. A graph-based technique is defined for modeling both the possible market demand scenarios that can occur and the production plans that are considered during the development period. This technique provides the software project manager with scheduling advice based on the current workstate and the events that have occurred. Additionally, the project manager is provided with the possibility to explore worst-case situations and "what-if" scenarios as a means to understand the risks posed by the imperfect market demand expectations.

1.5 Outline of the Thesis

The map of the thesis with the chapters and relations among them is depicted in Figure 1.1.

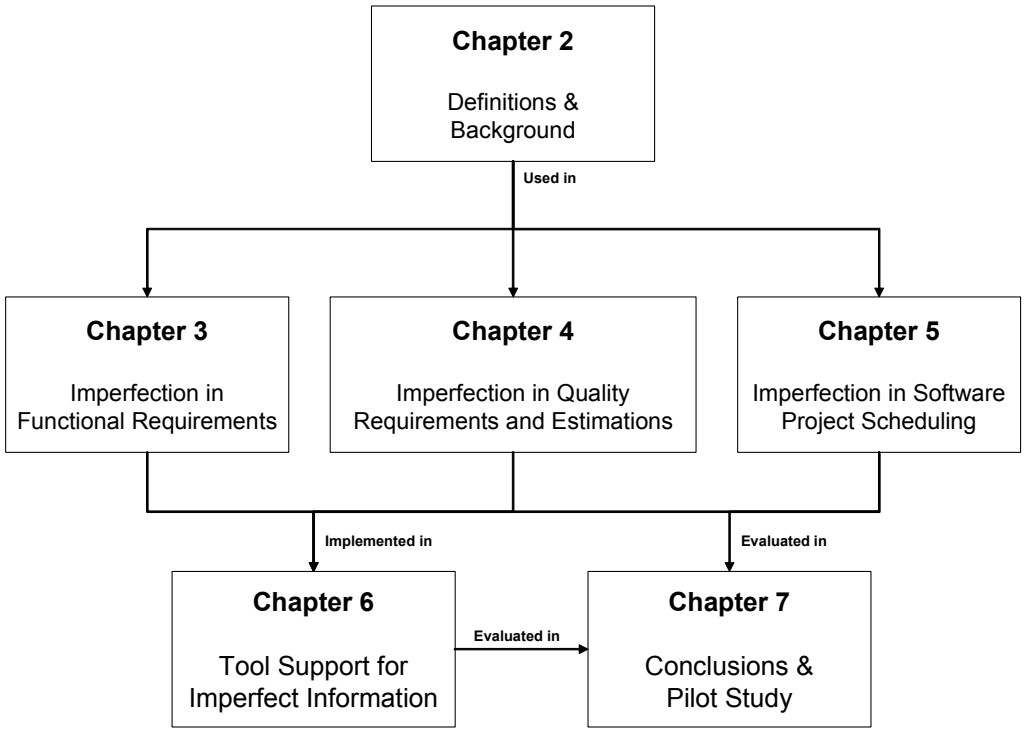


Figure 1.1 Thesis Map

The thesis consists of the following chapters:

Chapter 2 introduces the background knowledge and the definitions that will be used in this thesis. It describes the core concepts of probability theory, fuzzy set theory and fuzzy probability theory, which are used to capture and reason with imperfect information within the software development process, as defined in chapters 3 and 4. Additionally, it gives a short introduction into the optimization theory underlying the decision support models defined in chapters 3, 4 and 5.

Chapter 3 describes the Artifact Trace Model, which is used for tracing functional requirements to the components that fulfil them. The resulting artifact trace is used to trade-off system functionality and stakeholder interests. The chapter also introduces the concept of fuzzy functional requirements into the Artifact Trace Model. Fuzzy functional requirements are presented as an extension to crisp requirements that facilitate the introduction of alternative interpretations of imperfect specifications into the software design process. Within the Artifact Trace Model the alternative interpretations are treated as normal requirements, which means the software engineer is not hampered by the imperfect requirements. The Artifact Trace Model can be used as a means to derive multiple system designs based on particular stakeholder interests, such as relevance of urgency. This chapter is a revised version of the work described in [Noppen2004] and [Noppen2007].

Chapter 4 introduces the Design Tree Model, which captures design decisions that are taken during the software development process. In a design tree a trace is made of the decision, the contemplated solutions and their expected quality. By means of configurable design strategies, the resulting tree structure is used to steer the design process based on quality evaluations.

Subsequently, the chapter extends the expressiveness of quality requirements and quality estimations with definitions for probabilistic, fuzzy and fuzzy probabilistic requirements and estimations. With these extensions, a number of imperfection types are described, such as tolerance and variance. To facilitate the use of these imperfection models, this chapter defines comparison operators for the evaluation of imperfect quality requirements and imperfect quality estimations in a uniform manner. The extensions are integrated into the design tree model to enable design decision support, while dealing with imperfect information. The work described in this chapter is a revision of the work in [Noppen2007a], [Noppen2005] and [Noppen2005a].

Chapter 5 proposes a resource allocation approach for the implementation of components and assets of application frameworks and product lines under uncertain market demands. Graph-based models are proposed to describe the demands scenarios that can occur during the course of the software project and to describe the allocation strategies that are considered. The approach results in conditional scheduling advice, which returns the optimal resource allocation schedule based on the current demand and the work that has been completed. This chapter is an extension of the work described in [Noppen2004a].

Chapter 6 introduces a of set of tools, which have been implemented to support the application of our approach within software design processes. The toolset offers decision support for resource allocation, trade-off analysis and design decisions with full support for imperfect information as described in chapters 3, 4 and 5. This chapter describes the architectural design and user interface of the toolset, and it explores a number of points of interest with respect to its implementation.

Chapter 7 gives conclusions and an evaluation of the contributions in this thesis In addition, this chapter describes a pilot study that has been performed to assess the applicability of the approach and the toolset within a controlled setting. Early insights on the usefulness of our approach are given based on the results of this pilot study, as well as an outline with points of interest for experimental validation of the proposed models.

CHAPTER

2

"Arrakis teaches the attitude of the knife--chopping off what's incomplete and saying: "Now, it's complete because it's ended here."

- From Frank Herbert's Dune [Herbert2005]

DEFINITIONS AND BACKGROUND

2.1 Introduction

In this chapter, we give an overview of the basic concepts used in this thesis. We aim at selecting a consistent set of definitions and background information that supports the understanding of the subsequent chapters. We introduce the notion of imperfect information and we examine the support for imperfect information in state-of-the-art design processes, such as the Rational Unified Process [Kroll2003] [Kruchten1999] and Agile Processes [Martin2003] [Poppendieck2003]. Based on this analysis, we classify imperfect information into a number of types and we introduce the core concepts of probability theory, fuzzy set theory, fuzzy probability theory and optimization theory, which are used in this thesis to facilitate the description of imperfect information in software development processes.

The chapter is structured as follows. Section 2.2 introduces the concept of imperfect information and in Section 2.3 we analyze the capabilities of a number of well-known design processes with respect to supporting imperfect information during software development. In section 2.4, we distinguish a number of imperfect information types in the field of software engineering. The core concepts of the mathematical models we use to capture imperfect information, such as probability theory and fuzzy set theory, are described in section 2.5. In section 2.6, we give an overview of decision support systems, which are used in this thesis to support reasoning with imperfect information. In particular, we explore the core concepts of decision support systems based on optimization theory. An overview on existing approaches for supporting imperfect information is given in section 2.7 and the chapter is concluded in section 2.8.

2.2 Imperfect Information in Software Design Processes

2.2.1 Introduction

In every day life, almost all the information we encounter is imperfect. Generally, the absence of perfection in the information we receive does not limit us in performing most of our tasks. However, when the complexity of the task to be performed increases the influence of the imperfection becomes difficult to understand and manage. As a result, imperfect information can severely hamper very complex activities, such as developing a software system. In the field of software engineering, imperfection is not yet accepted as a natural state for available information. Nevertheless, software engineers frequently encounter information during the software design process that contains imperfection. Among others, software engineers can encounter:

- ambiguous requirement specifications;
- stakeholders with conflicting interests;
- incomplete descriptions of the desired functionality;
- changing requirement specifications;
- tolerance with respect to project delays and budgets;
- changes in the demand for products;
- uncertain expectations of software quality;
- required information that is missing

The occurrence of imperfect information in software design processes generally results from the fact that information is needed at a time at which it is not (yet) available. For example, software engineers may be forced to decompose a system into a certain modular structure to manage complexity, already in the early phase of software development. In practice, it may be preferable to defer this decision to a later phase, when the interactions among components are known. This allows grouping of densely interacting components into the same module, for the purpose of improving performance and cohesion. However, given the fact that the decomposition must be performed early in the design process, the software engineer will only have an imperfect view of the interactions between components. Software engineers and stakeholders therefore frequently encounter imperfect information, but they are seldom given effective tools to retain the oversight and minimize its impact on the software development process. Nonetheless, software engineers have to deal with the imperfection, especially in the early phases of software development.

During the last 30 years, a considerable number of design methods have been introduced, such as Structural design [Yourdon1979] and Rational Unified Process [Jacobson1999]. These approaches generally differ from each other with respect to the adopted models (functional, data-oriented, object-oriented, etc.). The methods propose a process which is guided by a large set of explicit and implicit heuristics rules. A method may distinguish itself from the others by introducing and emphasizing its own design heuristics. In [Tekinerdogan2002], based on their heuristics, architecture design methods are classified as artifact-driven, use-case-driven and domain-driven.

All software development processes acknowledge the difficulty of defining concise and sufficiently accurate requirement specifications, and offer their own approach to ensure the quality of the resulting software system. Through iteration and heuristics the software engineer is enabled to assess and adjust the software under design, which is structured depending on the type of system that is being designed. Most environments provide model editing, consistency checking, version management and code generation facilities. Incremental design is typically

well-suited to address changes in requirements, which can for instance result from information that is incomplete at the start of the development process. However, as has been identified in chapter 1, for successful corrective design steps through iteration, requirements must be fulfilled that can not be satisfied in a realistic setting. Therefore both the imperfection that is present in design inputs as the triggers to start a design increment should be better understood. Despite a considerable amount of research on process modeling [Kaiser1994] [Finkelstein1994], only a few environments provide a process support that explicitly considers imperfection that is present in the process inputs. Rather, this problem is left to be adjusted by iteration and the experience of software engineers. Formalizing design heuristics and providing some sort of expert system support during the design process is not exploited well. In this section, we examine four software development methods, to evaluate their approach for dealing with imperfect information. First we take a look at the waterfall model, which essentially was defined to demonstrate the impact of imperfect information on sequential software design. Next we examine Rational Unified Process and Agile Processes as examples of a complete and an agile development process respectively. We look at Synbad as an example of synthesis-analysis based software development approaches and the Architecture Trade-off Analysis Method as an evaluation method at the architectural level of software design.

2.2.2 The Waterfall Model

One of the most well-known, and criticized, process models for software development is the waterfall model, introduced by Royce [Royce1970]. In this model, the software development process is divided into a sequence of discrete phases. Schematically, the waterfall model is depicted as follows:

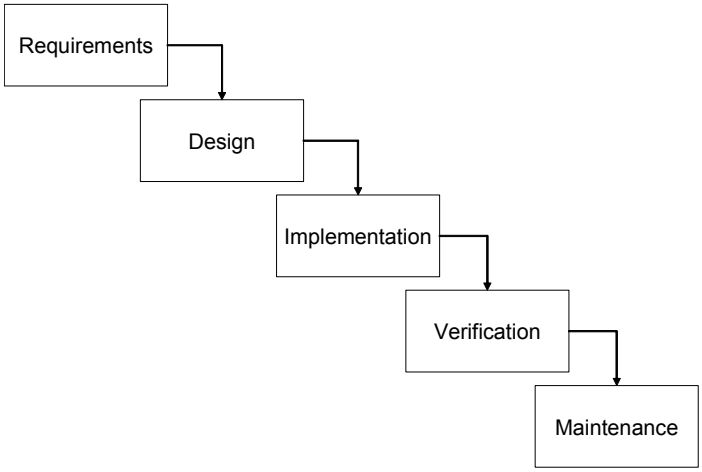


Figure 2.1 The Waterfall Model

The phases depicted in Figure 2.1 are performed in the waterfall model in a pure sequential manner, which means that it is not possible to revisit previous phases after they have been completed. Obviously, such a rigid sequence of defined states only results in software of high quality, when the initial requirement specification accurately captures the desires of the stakeholders. In the case that the requirement specification fails to achieve this, and this is not noticed until later in the design process, software engineers are forced to start from the beginning. The influence of imperfect information on the waterfall model is therefore prevalent, especially since the occurrence of imperfection in subsequent phases can lead to invalidation of the software design, in a manner identical to imperfect requirement specifications.

In response to the obvious disadvantages of the waterfall model, the concept of *iterative design* was introduced. In this approach, design is done by means of a cyclic process of prototyping,

testing, analyzing and refining. Iterative design is better suited to work with imperfect inputs, since it enables software designers to adjust designs and decisions during new iterations of the development process. In typical models such as the spiral model [Boehm1986], these iterations were envisioned to last for a period of six months to a year. However, recently agile processes have shortened the time between iterations considerably.

2.2.3 Rational Unified Process

The Rational Unified Process (RUP) [Jacobson1999] is a complete and well-defined software development method, which has gained considerable popularity in recent years. It is defined to be a customizable framework, which makes it adaptable to specific companies and settings. The RUP is frequently described as being *iterative*, *architecture-centric* and *use-case-driven*, which means that the RUP supports iterative development while using use-cases as its primary input. In Figure 2.2 the iterative character of the RUP is depicted as it is described in [Kroll2003].

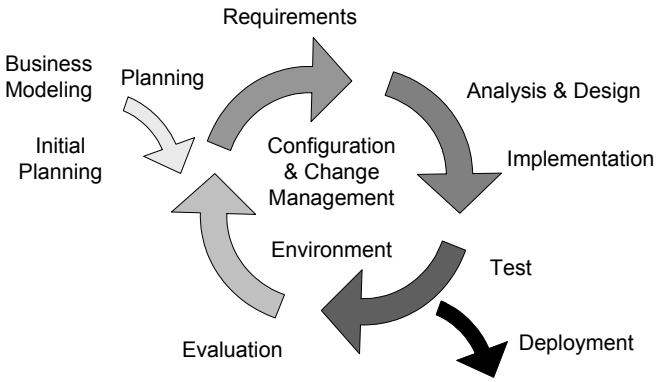


Figure 2.2 Iteration in the Rational Unified Process

In this figure, the software development process is initiated by modeling the business information, after which the final product is built in a number of increments. Each iteration step builds on the results of the previous iteration, in which new insights and adjustments can be considered. This iterative character is maintained throughout the four phases of the RUP, being inception, elaboration, construction and transition. A schematic depiction that is commonly used to characterize the RUP, is depicted in Figure 2.3.

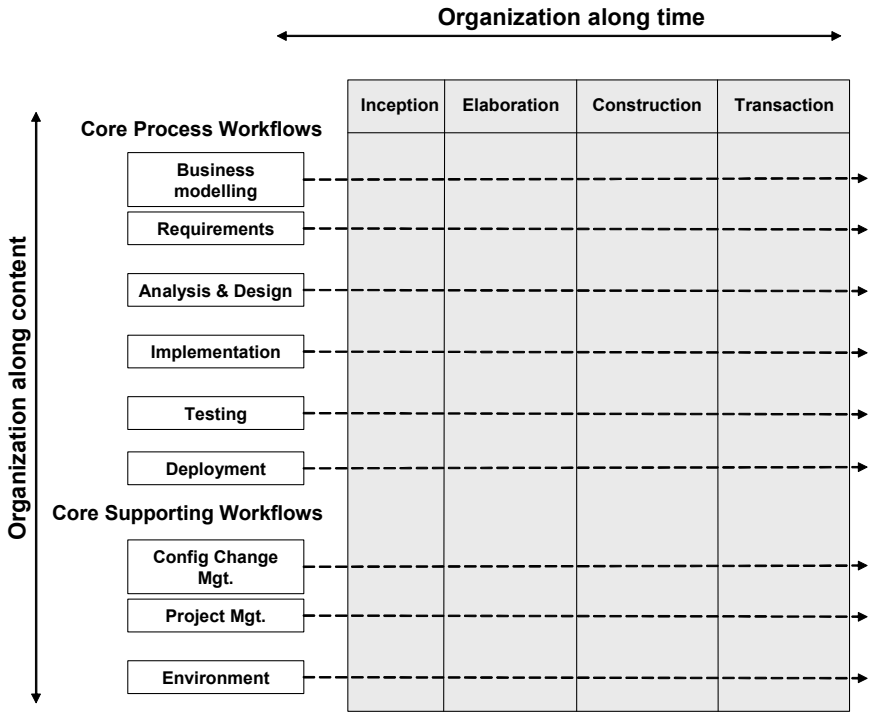


Figure 2.3 Schematic representation of the RUP

In this figure, the RUP is described by two dimensions, the *organization along time* describes the consecutive stages the development process will pass through. During each of these phases, effort is allocated in a specific manner to the workflows, which are depicted in the *organization along content*. Along the development process, in each phase multiple iterations are done along the workflows, which eventually result in the final product. The reason why the RUP emphasizes iteration during software development is that it resolves particular problems that were found when using the classical waterfall process. In [Kroll2003] it is argued that iteration facilitates the change of requirements, addressing risks, management insights and decisions and design adjustment, amongst others.

The design steps in the RUP are taken by identifying analysis packages, classes and objects based on the design heuristics of the process. For example, analysis packages are identified by answering questions such as *"which system concepts are needed for interfacing?"* or *"which system concepts represent information?"*. These questions and heuristics are then resolved and used from the perspective defined by the use-case descriptions of the system. As a result, the development activities become very sensitive to these use-case specifications. Since typically use-case descriptions only offer a very abstract and partial view of the system, it becomes difficult to use the heuristics in a uniform and straightforward manner. In addition, the heuristics-based identification of analysis classes and packages remains vague. The imperfection that is present in both the use-case descriptions as well as the heuristic design advice hampers the development process, and therefore can lead to systems that do not fulfil the requirements.

The iterative nature of the RUP partially addresses these problems, since at the beginning of a new iteration cycle the available information can be reassessed. By reconsidering and adjusting earlier design decisions the functionality and quality can be adjusted. However, typically it is very complex to perform systematic iteration in the software development process in the way it is proposed by the RUP. Iterative cycles can only be properly managed after previous iterations have been completed, and it is very difficult to perform design and reiteration in parallel. Naturally, the iterative approach of the RUP can only be fully successful when it fulfils

the requirements that have been defined in section 1.2. While these requirements in general can not be fulfilled, the RUP is in particular vulnerable with respect to the *All-Always-Requirement*. Since the iterative cycle in the RUP takes a relatively long time, there is a high probability that the requirements will have changed upon completion of the iteration. As a result, the corrective step is invalidated and new iterative corrections must be performed.

2.2.4 Agile Software Development

In response to the very complex and detailed software development processes, agile processes have gained increasing momentum over the last decade. Contrary to complex and detailed development processes, agile development processes focus on short iteration cycles, which typically last for two to four weeks. In each of these iterations, a mini-increment of the desired functionality is achieved while including the feedback from the previous iterations. Agile development processes are typified in [Poppendieck2003] by the following *lean principles*:

Eliminate waste. Anything that is present in the software development process, that is not directly needed to fulfil the demands of the customer is considered waste. This can range from unused requirement specifications to superfluous components in the software architecture.

Amplify learning. Design should be seen as a learning process rather than a production process. Insights on the right decisions are attained over multiple iterations as a result. Therefore the development process should support learning facilities.

Decide as late as possible. To ensure that decisions are taken based on the best information, postpone decision making to the latest possible moment. Especially in an uncertain environment this enables the arrival of new relevant information, which can reduce errors.

Deliver as fast as possible. Where detailed processes typically try to avoid making errors, and deliver the software "when it is good", prototypes are delivered as fast as possible, since agile processes see this as the only way to gather meaningful feedback. The shortened iteration cycle is seen as one of the most important approaches in agile processes to reduce waste.

Empower the team. Due to the very short cycles, the responsibility of the project management shifts towards the individual workers. The success of the approach depends on the agreement that small increments of the software system are delivered at regular intervals, and is ensured by daily meetings, integration and testing.

Build integrity in. Integrity of the software system (how well it corresponds to the customer desires) is monitored by making sure the central concepts cooperate as a cohesive whole. In addition, it is aimed to keep the software meaningful over time.

See the whole. Finally, it is advocated to keep the necessity of the entire system in mind while designing the software. A maximization of individual parts is not necessarily the best overall architecture, which makes it important not to have too narrow a focus.

Iterations in agile processes take considerably less time than in detailed processes, which means that they are more capable of adapting to changes in requirements. This implies that agile processes are less vulnerable to the *All-Always-Requirement*, as defined in section 1.2. Since the iterative cycle takes less time, it is less likely that requirements change during the corrective steps. However, the shortened iteration cycle also makes it less likely for all the imperfection to be removed at the same time, a restriction on successful iteration stipulated by the *All-At-Once-Requirement*. As a result, the corrective steps are still based on imperfect information, and therefore likely candidates for future iterations. In addition, one of the main points of criticism on agile processes focuses on the lack of structure, the need for experienced developers and added difficulty in contractual negotiations. It is plausible that agile processes therefore are faced with imperfect information more frequently than detailed development pro-

cesses. This means that the structure of agile processes in part is both the solution and the cause of imperfect information.

2.2.5 Analysis-Synthesis

Analysis-synthesis is a relatively new approach to software design although in traditional engineering it has been applied for a longer time. As a basic notion, analysis-synthesis sees software development as a problem solving activity, much like it is done in traditional engineering disciplines, such as the Synbad approach that is described in [Tekinerdogan2000]. In such a view, the problems are presented by the requirement specification and the solution is formed by the completed system. In the synthesis based approaches typically the problems are decomposed into smaller, more manageable subproblems. From here the problems are transformed into the resulting software system in a finite number of steps. Based on the problem decomposition, the relevant domains of expertise are identified (commonly named solution domains). From these domains the solution concepts are extracted that make up the system design. Schematically an analysis-synthesis process can be characterized as follows:

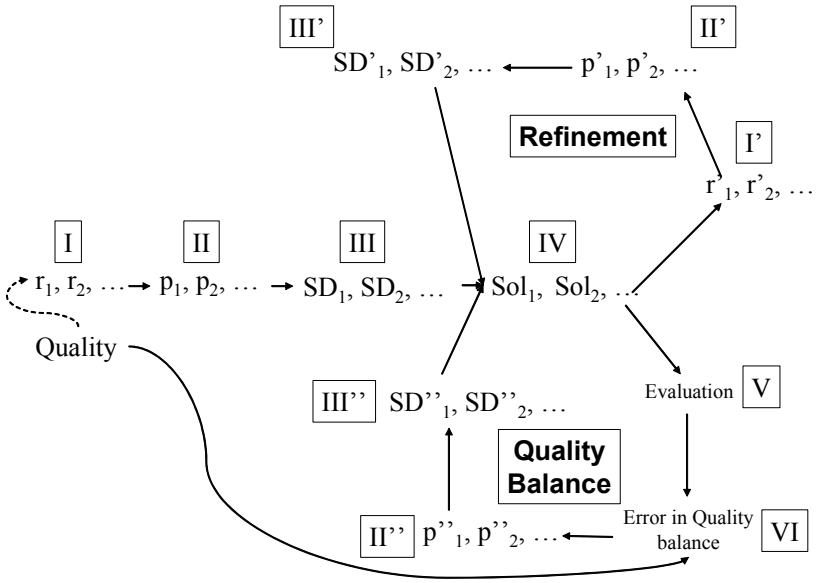


Figure 2.4 The Synbad Design Process

In Figure 2.4, the analysis-synthesis design process is depicted as a sequence of phases, numbered I, II, etcetera. These phases represent intermediate design models during the software design process. The first phase leads to the definition of the requirements, which are denoted by r_1, r_2,… in the figure. In the second phase, a decomposition of the requirements is made into the relevant problems in order to fulfil the requirements that were found in phase I. These problems are denoted p_1, p_2,… in the figure. The problems are mapped to solution domains in the third phase, denoted by SD_1, SD_2,…. In the fourth phase, the solution concepts, denoted Sol_1, Sol_2,… are selected from the solution domains to make up the final system.

After the completion of phase IV, two different iterations can be performed. The *refinement iteration* is the refinement of each solution concept into smaller sub-concepts, that provide the implementation at a lower level of abstraction. This is done by first defining requirements for each solution concept, which are denoted by r'_1, r'_2,… in the figure at phase I'. These requirements are in turn mapped to problems at stage II', denoted by p'_1, p'_2,…. In phase III', the problems are mapped to solution concepts, indicated by Sol'_1, Sol'_2,… in the figure. From here, a new phase IV is reached, where new, lower-level solutions are identified. This iterative

cycle can be repeated until an implementable solution is found. The second iteration is the *quality balance iteration* cycle. In this iteration, the expected quality of the current design is assessed. In phase V, the current set of solution concepts is graded by making an evaluation of the specific quality attributes such as stability, performance, etcetera. These characteristics are compared to the quality requirements in phase VI. In the case that the design does not fulfil the quality requirements, the problem areas are identified as errors in the quality balance. To resolve these errors, a number of problems are identified in phase II'', indicated by p''_1, p''_2,… in the figure. These problems are mapped to solution domains in phase III'', in the picture denoted by SD''_1, SD''_2,…. From the solution domains, solution concepts of higher quality are selected in phase IV, in order to increase the overall quality of the design. This iterative cycle can be repeated until a solution of acceptable quality has been identified. After this, the refinement cycle can be used to come to an implementation.

Analysis-synthesis-based development approaches have two mechanisms to address imperfection information, *iteration* and *decomposition*. Like other design processes, iteration can be used to adjust software designs, but an explicit distinction is made between quality-based iteration and refinement-based iteration. However, the difficulties originating from imperfect information with respect to iteration equally apply to synthesis-based approaches. For the iterative correction of the software design to be successful, the four requirements identified in section 1.2 must be fulfilled. However, just as with other development processes, these requirements are unrealistic for analysis-synthesis-based development methods. The second mechanism is the explicit decomposition of problems in the early phases of the software design process. By distinguishing multiple independent problem parts, the influence of imperfection can be isolated. As a result, the impact can be minimized during the software development. Nonetheless, decomposition suffers from imperfect information, since the decomposition itself is based on the information available at this point in the design process. When this information is subject to imperfection, the chosen decomposition can be invalidated at the later stages of the design process, which has a severe impact on the design effort up to that point.

2.2.6 Architecture Trade-off Analysis Method

In the Architecture Trade-off Analysis Method (ATAM) [Barbacci1998], a method is proposed for the systematic analysis of the expected quality attributes of software architectures. This is done by defining a model for each relevant attribute and performing a scenario-based analysis using these models. The results are compared to the requirements and server as a means to understand the trade-off that exists between attributes. After the software architecture has been described, ATAM consists of four steps:

1. Identify requirements, constraints and structural view of the architecture.
2. Define models for the relevant quality attributes.
3. Perform a scenario-based analysis using the attribute models.
4. Compare the results against the requirements and model the trade-off and sensitivity.

The activities that are performed in the ATAM are aimed at gaining insight into the quality of software during the architectural design phases. The usage of explicit models for quality attributes greatly increases this insight and can assist in the identification of, for example, ambiguous requirements.

The Architecture Trade-Off Analysis Method is a logical continuation of the Software Architecture Analysis Method (SAAM), which is aimed at analyzing and understanding the capabilities of a software architecture using usage scenarios. However, in SAAM the quality attributes were not explicitly modeled, which makes the assessment qualitative rather than quantitative. Over the years, many extensions have been proposed that enhance the evaluation capabilities of the SAAM. For example, in [Molter1999] ESAAMI (Extending SAAM by Integration in

the Domain) is presented, in which the SAAM is embedded in a domain-specific and reuse-based development process. This facilitates the reuse of domain knowledge. As a result, the evaluations become more accurate.

Scenario-based architecture analysis methods, such as the ATAM, assist the software engineer in gaining an early insight in the quality attributes of the current system design. Nevertheless, the validity of these insights heavily depends on how well the set of scenarios captures the context in which the final system will be used. Most of these approaches are aware of the fact that these scenarios form an imperfect description of this context, there is no support to express this imperfection by, for instance, probability annotations for each scenario. While it is possible to compare architectural alternatives in a uniform manner, these approaches do not describe the degree to which the evaluation corresponds to the real world, a point which is addressed in ATAM by explicitly modeling quality attributes. Nonetheless, the degree to which these models capture the actual attribute directly influences the evaluation. In addition, the evaluations of the software architecture are compared to the quality requirements to determine the applicability of the design for the stakeholders. As we have established in chapter 1, the necessity of defining quality requirements early in the development process makes it difficult to specify accurate and precise restrictions on the desired quality. The likeliness of mis-assessing the expected quality of a software architecture therefore only increases, since the scenario-based evaluation as well as the quality requirements are subject to imperfect information.

2.3 Types of Imperfect Information

In this thesis, our focus is on addressing imperfect information that occurs during software development activities. In this specific field, imperfect information can come from two distinct sources. The first source is formed by the stakeholder, or group of stakeholders. The functional requirements and the quality requirements are specified in accordance with these stakeholders, but this is a process that is notoriously difficult. As a result, requirement specifications are a natural point at which imperfection information can enter the software development process. The second source of imperfect information is formed by the software engineers that design the software system. As with most non-trivial design activities, in software architecture design it is difficult, if not impossible, to precisely indicate the quality of the resulting system during the initial design decisions. While the software architects will have a “pretty good idea” on the quality of the system that results from a particular decision, it can not be determined with certainty.

The models proposed in this thesis are intended to address imperfect information in a uniform and well-defined manner, by specifically considering and supporting the nature and characteristics of software development processes. To ensure a consistent understanding of the problems and solutions that are described, we introduce the following basic terms:

Perfect Information: Information that contains all the attributes and values with sufficient precision and certainty for the purpose for which it is used.

Imperfect Information: Information that is not perfect.

Uncertain Information: Information that is imperfect, but will become certain at some point in the future.

Imprecise Information: Information that is imperfect, and that will remain imperfect to a certain degree.

For our models we distinguish between two types of imperfect information, *impreciseness* and *uncertainty*. Although there are slight differences in the terminologies used in other fields, in the large our definitions for these terms conform to the standard definitions in the literature, such as [Bonissone1985] [Bosc1993] and [Parson1996]. Here, the term uncertainty refers to a

transient case, where imperfect information becomes eventually perfect (well known) in due time. In contrast, imprecise information will always remain imperfect to some degree.

The distinction we make between types of imperfection closely corresponds to the types of imperfection that can be encountered in the software development process. In the analysis of the development processes in section 2.2, we have seen that imperfection can originate from many sources and can be used to describe different aspects of what is not known. For example, when an imperfect description is given of the maximum budget for a software project, this in most cases indicates a certain tolerance with respect to the allowed costs. This type of imperfection closely corresponds to impreciseness. For impreciseness, it is important to have an indication of the acceptable boundaries, but it is seldom necessary to precisely define the amount of tolerance at any given point in time.

We have also seen that during software design frequently estimations are made of the expected quality attributes of the resulting system. Due to an incomplete view of what the finished system will look like, these estimations will also contain imperfection. However, this imperfection falls into the category of uncertainty, since the actual quality attribute values can be determined upon completion of the software system. The same is also true for market expectations. While it can be unclear how the market will behave four weeks from now, this will become clear when these four weeks have passed. In our view, it is important to include the notion that uncertain information in due time can be falsified, since uncertain information introduces risks into design processes. With these terms, we can classify the problems and properties of our approach accurately. In the remainder of this thesis, we will use these terms consistently.

It is important to note that there is a difference between information and the way in which it is seen by the people that use it. Suppose a software engineer has to design a user interface based on the assumption that all user are telepathic. Later, it turns out that only some of the users are actually telepathic. The assumption of all users being telepathic clearly was imperfect. However, at the moment the user interface was designed, the designer had no reason to doubt the information, and therefore *assumed* it to be perfect. In order to consider the imperfection that exists in available information, the information needs to be identified as being imperfect. After this, the type of imperfection must be identified and modeled. The telepathic users assumption was clearly imperfect, and in this case uncertain since after a while it turned out that not all users are telepathic.

2.4 Models for Imperfect Information

2.4.1 Introduction

In this thesis, we focus on mathematical representations of imperfect information by means of probability theory, fuzzy set theory and fuzzy probability theory. These theories offer a solid mathematical foundation for the specification and manipulation of imperfect information and their diversity ensures sufficient flexibility to express the various types of imperfection. Fuzzy set theory and probability are two models that can be used to describe imperfect information. The actual imperfection is different for both models, which means that the character of fuzzy sets and probability is orthogonal. The difference between probability theory and fuzzy set theory can be illustrated with the following example.

> *The coach of a basketball team is looking for a new player. This player must be at least 2 metres tall. He is pointed to a player that might fit this requirement. What is now the difference between a player that is probably 2 metres tall and a player that is approximately 2 metres tall? In the first case, the coach gets a player that is 2 metres tall with a high probability, say 0.95. This means, that the coach will receive a random player from a group of 20 players, from which 19 are at least 2 metres tall and one is definitely not. Therefore*

there is 5% chance the coach will get a player of unknown height but certainly smaller than 2 metres. In the second case, the 0.95 represents the membership value in the set of people that are at least 2 metres tall. This number is attained by asking 20 experts whether a player is 2 metres tall, without the possibility of actually measuring. From these 20 experts, only one though the player is not 2 metres. Based on this number, the coach will receive a player of which most experts think he is 2 metres tall, which means he will not be substantially smaller. - (Example by Bart Kosko)

In [Klir1995], among others, the orthogonal character of probability theory and fuzzy set theory is explained in more detail. In this section we explain the basic concepts of probability theory, fuzzy set theory and fuzzy probability theory, which are used as the starting point for imperfection models in the remainder of this thesis.

2.4.2 Probability Theory

Of all mathematical models that represent imperfect information, probability theory is the most well-known. Probability theory is used to describe situations where the results of experiments, when performed under identical circumstances yield different results, such as flipping a coin. In this thesis, we use random variables with associated probability distributions to describe imperfection that falls into the category of uncertainty. This means that probability distributions are used to describe the value for random variables in software development processes in future points in time. This reflects the situation where the value of a variable in the decision process is not known at the current point in time, but will be known when some point in the future is reached. We use this model of imperfection in two models described in this thesis. First, in chapter 4 quality requirements and estimations are described using continuous probability distributions as one of the possible manners to model imperfect information. The distributions are used in particular when behavior needs to be described that is itself subject to a continuous random variable, such as response time, that depends on a waiting queue. Second, in chapter 5 we use discrete probability distributions to model changes in market demands for products to be produced, which are used to optimize the allocation of resources with respect to the expected profit. Since the allocation is done in a finite amount of steps, the demand state from the market only needs to be considered in these steps, which makes a discrete probability distribution a sufficiently accurate model. As a starting point for the reader, we shortly summarize the basic concepts of continuous probability distributions that are used in this thesis.

Given a continuous probability distribution f for some event, the probability $P(a,b)$ of the occurrence of the event between a and b is given by:

$$P(a, b) = \int_a^b f(x)\, dx$$

and the expectation value E of the probability distribution is given by:

$$E = \int_U xf(x)\, dx$$, where U is the universe of discourse.

In chapter 4 this theory is applied to exponential density functions, which are given by $f_\lambda(x) = \lambda e^{-\lambda x}$ where $\lambda > 0$. For this choice the probability $P_\lambda(a, b)$ of the occurrence of an event between a and b is given by

$$P_\lambda(a,b) = \int_a^b \lambda e^{-\lambda x} dx = e^{-\lambda a} - e^{-\lambda b}$$

and the expectation value E_λ is given by:

$$E_\lambda = \int_0^\infty x\lambda e^{-\lambda x} dx = 1/\lambda$$

2.4.3 Fuzzy Set Theory

The second imperfection model that is used for our approach is fuzzy set theory [Klir1995], which is an extension of classical set theory. In classical set theory, membership of elements in a set is defined in a binary manner; an element is either a member of the set or it is not. Fuzzy set theory allows elements to be a *partial member* of a set, which is used to describe particular types of imperfect information. The partial membership of an element *x* in a fuzzy set is given by the membership value $\mu(x)$, where μ is a function that maps the universe of discourse to the interval [0, 1]. This value is the degree to which x is an element of the fuzzy set, where 1 means "completely a member" and 0 means "completely not a member". Like in probability theory, a distinction is made between discrete fuzzy sets and continuous fuzzy sets. By considering the degree of membership during manipulations of the imperfect information modeled by fuzzy sets, the resulting conclusions are arguable more justifiable. Fuzzy set theory is used in this thesis to describe imperfect information that falls into the categories of impreciseness and uncertainty. The flexibility of fuzzy set theory enables us to describe imperfection in functional requirements as well as quality requirements and estimations, which is achieved by using continuous fuzzy sets and in particular *fuzzy numbers* and *fuzzy intervals*.

Fuzzy numbers are a fuzzy set representation of imperfect descriptions of numbers. In such a fuzzy set description, exactly one element has a membership degree equal to one. In this thesis, we assume all fuzzy numbers to have a membership function with a triangular shape, which will be referred to as *triangular fuzzy numbers*. A triangular fuzzy number is a fuzzy set on the domain of real numbers whose membership function μ is given by:

$$\mu(x) = 0 \quad , \text{if } x \le a$$
$$\mu(x) = (x-a)(b-a) \quad , \text{if } a \le x \le b$$
$$\mu(x) = (c-x)(c-b) \quad , \text{if } b \le x \le c$$
$$\mu(x) = 0 \quad , \text{if } x \ge c$$

for some real numbers a, b, c with $a \le b \le c$, and is denoted by (a, b, c). In Figure 2.5 a triangular fuzzy number is depicted.

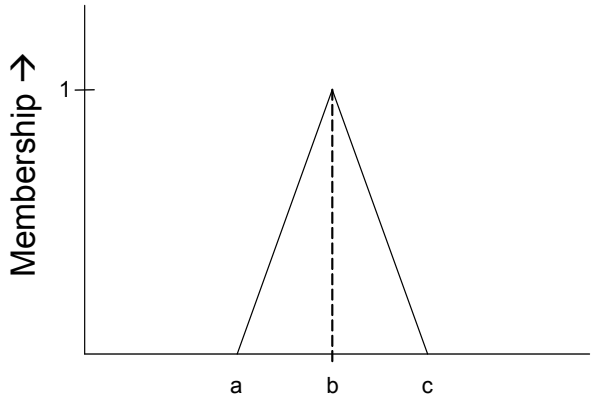


Figure 2.5 Triangular fuzzy number

It can be seen that the value b is seen as the most appropriate value, but values close to this value also have high membership values. Values smaller than a and larger than c are irrelevant, so they have membership value zero.

A second type of fuzzy set used in this thesis is the *fuzzy interval*, which is a representation for imperfect specifications of normal intervals. This type of fuzzy set in particular models the fuzzy boundaries of the interval. We assume all bounded fuzzy intervals to be *trapezoidal*, and the membership function μ of a trapezoidal fuzzy interval is given by:

$\mu(x) = 0$, *if* $x \leq a$

$\mu(x) = (x-a)(b-a)$, *if* $a \leq x \leq b$

$\mu(x) = 1$, *if* $b \leq x \leq c$

$\mu(x) = (d-x)(d-c)$, *if* $c \leq x \leq d$

$\mu(x) = 0$, *if* $x \geq c$

for some real numbers a, b, c, d with $a \leq b \leq c \leq d$, and is denoted by (a, b, c, d). In Figure 2.6 a trapezoidal fuzzy interval is used to represent the imperfect statement *approximately between b and c*.

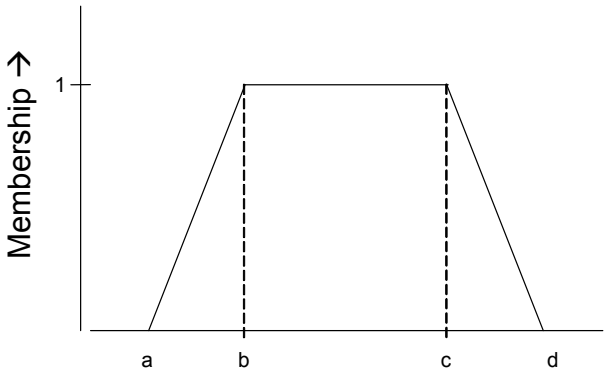


Figure 2.6 Trapezoidal Fuzzy Interval

Similar to the fuzzy number representation in Figure 2.5, the trapezoidal interval extends the boundaries of traditional interval specifications by including surrounding values to a certain membership degree. In chapter 4 we also use a special case of fuzzy intervals, the semi-infinite fuzzy interval. We assume such type of interval to have a *semi-trapezoidal* shape, and to have the following membership function:

$\mu(x) = 1$, *if* $x \leq a$

$\mu(x) = (x-a)(b-a)$, *if* $a \leq x \leq b$

$\mu(x) = 0$, *if* $b \leq x$

for some real numbers a, b with $a \leq b$, and is denoted by (a, b). In Figure 2.7 a semi-trapezoidal interval is depicted that could represent the imperfect statement *smaller than approximately a*.

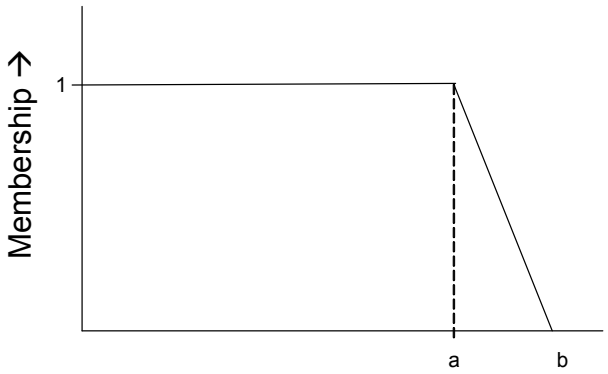


Figure 2.7 Semi-trapezoidal Fuzzy Interval

From the specification it can already be seen that semi-trapezoidal intervals are in particular useful for specifying constraints on acceptable values, which corresponds to, for example, quality requirements. In this picture, the interval describes a fuzzy upperbound, but naturally a fuzzy lowerbound can be described in an identical manner.

It can be desirable at certain points to remove the fuzzy information and replace it with crisp numbers. This is done be determining the crisp number that best represents the fuzzy set. This process is called *defuzzification*. For the defuzzification of fuzzy sets, a number of methods have been proposed in the literature, of which three have become predominant, according to [Klir1995]. These three methods are the center of area method, the center of maxima method and the mean of maxima method. The center of area method, which is also referred to as the center of gravity method or centroid method, defines the defuzzified value of a fuzzy set as the value for which the graph under its membership function is divided into two equal subareas. The center of maxima method defines the defuzzified value to be the average of the values that have the smallest and largest value that have the highest membership value. The mean of maxima method is usually applied for discrete fuzzy sets, and defines the defuzzified value to be the average of all values in the set with the maximal membership value. A more elaborate description of defuzzification operators can be found in the aforementioned reference.

An important concept in fuzzy set theory, is the concept of α-*cuts*, which is also used in this thesis. For $0<\alpha<=1$, the α-cut of a fuzzy set *F* with membership function μ is defined by $\{x \mid \mu(x) \geq \alpha \}$ and denoted by $F[\alpha]$. If F is a fuzzy number or a fuzzy interval, then $F[\alpha]$ is an interval.

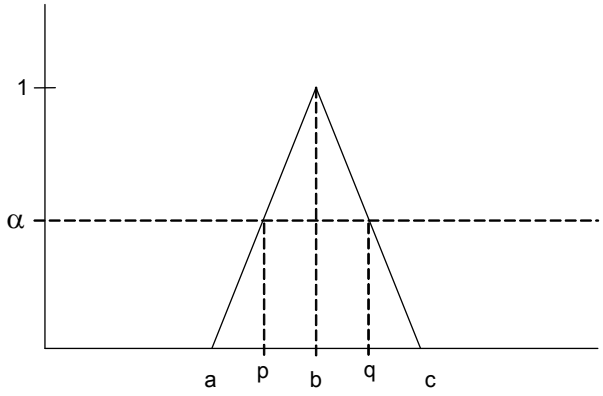


Figure 2.8 α-cut of a Triangular Fuzzy Number

In Figure 2.8, a schematic depiction is given of an α-cut of a triangular fuzzy number. In this case, F[α] = [p, q]. In general, an α-cut of a triangular fuzzy number (a_1, a_2, a_3) is $[a_1 + \alpha(a_2-a_1), a_3 - \alpha(a_3-a_2)]$, for a semi-trapezoidal fuzzy interval (b_1, b_2) the α-cut is given by $]-\infty, b_2 - \alpha(b_2-b_1)]$ and for a trapezoidal fuzzy interval (c1, c2, c3, c4) the α-cut is given by $[c_1 + \alpha(c_2-c_1), c_3 - \alpha(c_4-c_3)]$. The α-cut concept is used in this thesis for the derivation of generalized comparison operators for fuzzy numbers and fuzzy intervals.

2.4.4 Fuzzy Probability Theory

As a result of their orthogonal character, with fuzzy set theory and probability theory a wide range of imperfect information can be described and analyzed. However, even when using these imperfection models, finding or providing accurate information for these models can be hard. For example, when probability density functions are used to model stochastic behavior, identifying an accurate set of function parameters that best describe the situation can be a difficult, if not impossible, task. To facilitate this imperfection, which occurs within the definition of probability models, the concept of fuzzy probabilities has been introduced. Fuzzy probability theory extends probability theory with the possibility of expressing imperfection in the parameters of the probability density function. Recently there has been an increased interest in the fuzzy logic community in the area of fuzzy probabilities. In [Buckley2003], fuzzy probability distributions are defined by replacing parameters in families of crisp probability distributions (such as exponential, standard normal, etc.) with fuzzy numbers. This way it becomes possible to consider a range of density functions of the same family to a certain degree, indicated by the degree of membership of each parameter in the fuzzy set representation of the respective fuzzy parameter. Fuzzy probability distributions are in particular useful when probability distributions can globally describe imperfection of information, but it remains difficult to define the precise parameters of the chosen function. We use fuzzy probability distributions in chapter 4 to model performance estimations.

Consider, like before, the family of exponential probability density functions, given by $f_\lambda(x) = \lambda e^{-\lambda x}$ where $\lambda > 0$. For a fuzzy number Λ on the domain of non-negative real numbers. The probability $P_\Lambda(a,b)$ of the occurrence of an event between *a* and *b* is given by its α-cuts:

$$P_\Lambda(a,b)[\alpha] = \{ \int_a^b f_\lambda(x) \,|\, \lambda \in \Lambda[\alpha] \}$$

and the expectation value E_Λ is given by:

$$E_\Lambda[\alpha] = \{ \int_0^\infty x f_\lambda(x) \,|\, \lambda \in \Lambda[\alpha] \}$$

Note that probabilities resulting from fuzzy probability distributions, as well as expectation values, are fuzzy numbers, typically of a non-triangular shape. This can be intuitively explained by the fact that when it is only possible to provide imperfect inputs, it is generally not possible to provide perfect outputs. This also does not restrict the application of our approach, since the provided inputs for fuzzy probability distributions can be restricted to triangular fuzzy numbers, and the evaluation can be facilitated with defuzzification functions.

2.5 Decision Support during Software Development

2.5.1 Introduction

Decision processes have been studied in various areas, and in particular in the field of operations research and decision theory. In particular, in areas where complex decisions need to be taken and many parameters need to be considered simultaneously, decision support systems based on these approaches have proven to be very useful. Decision support systems are used in a very broad variety of applications and as such are difficult to capture in a single description. However, a typical component of decision support systems is the ability for automated reasoning with specific types of input and coming to a unique best decision based on this information. An important part of decision support systems is therefore formed by the decision making model, which finds its origin in decision theory. With decision support systems it is possible to utilize the existing knowledge in these fields, and ensure the proper application of the underlying theories. Decision support methods come in many different forms and shapes, but generally they offer an automatable approach for analyzing particular design states and advice on how to proceed from the current point. Based on the type of assistance that is offered, [Power2002] classifies decision support systems into five categories:

- **Model Driven Decision Support** is based on the usage of statistical or optimization models. The data and parameters provided by the users are used to assist decision makers in analyzing the situation at hand.
- **Communication Driven Decision Support** supports the coordination of multiple people working an a single task, and are used in particular in project settings.
- **Data Driven Decision Support** focusses on offering decision advice based the analysis of a sequence of relevant data items, such as trend analysis of market demands.
- **Document Driven Decision Support** is aimed at managing and working with unstructured information that can be found in a body of (electronic) documents.
- **Knowledge Driven Decision Support** aims to capture existing problem solving expertise by means of storing facts, reasoning rules and heuristics.

An ongoing trend in software design is that software systems have to fulfil requirements that are increasingly more complex, and as a result the software systems themselves also become more intricate. In response to this increased complexity, there has been extensive research in the area of decision support approaches for software development processes [Ruhe2004]. Since the inclusion of imperfection on development processes only adds to this complexity, automated analysis and support becomes vital for manageable software development.

In this thesis, we propose two decision support approaches that belong to the category of model driven decision support. Both approaches define an optimization model for providing decision support to the software engineer, and require inputs from both stakeholders and software engineers to determine the current state. The resource scheduling approach presented in chapter 5 can also be considered a decision support approach that belong to the category of model driven decision support. In particular, the optimization model and the resulting conditional production plan are typical properties of this type of decision support approach.

2.5.2 Optimization-based Decision Support

In this thesis, we are particularly interested in extending the expressive capabilities of software development processes with models that can describe imperfect information, such as probability theory and fuzzy set theory. However, the application of the mathematical operations that underlie these models can be difficult and cumbersome, in particular since these models are not always well known or understood by the intended users of the approach. We therefore

define our imperfection models as *model driven decision support models*, where the software engineers and stakeholders are expected to only provide the relevant optimization inputs. The optimization approach ensures the proper application of the mathematical operations and offers decision support based on the results of the optimization. In this paragraph, we explore the basic elements of optimization models, which are used as the starting point for our approach in the following chapters. Optimization theory is the area of mathematics that studies the extremal values of a function: its minima and maxima. Optimization theory has proven to be useful in particular in areas of complex decision making, such as design optimization and operations research.

Optimization problems in the mathematical sense are problems, for which a value defined by an expression should be optimized (maximized or minimized) by choosing the value of real or integer variables from an allowed set. Typically, a set of restrictions apply on the values these variables are allowed to take, which can be equality or inequality constraints. A mathematical optimization problem can be characterized as follows: Given: an objective function $f\colon D \to R$, where the domain D is a set. Then the optimization problem in case of minimization, is defined as: find $x_0 \in D$ such that $\forall x \in D \ f(x_0) \leq f(x)$ holds. The *search space* D is typically a subset of the Euclidian space R^n and its elements are called *feasible* or *candidate solutions*. The feasible solution that maximizes (or minimizes) the objective function is called the *optimal solution*. Depending on the attributes of interest of the optimization problem, different types of optimization approaches have been described, each with its own specific merits. For example, *integer programming* is an optimization approach that searches for optimal solutions that exist only of integer numbers. *Stochastic Programming* allows stochastic behavior in the parameters and *Dynamic Programming* focuses in particular on optimization problems that can be decomposed into smaller, reusable optimization problems. Other optimization approaches include combinatorial optimization and linear programming.

With the application of optimization theory within decision support systems, it becomes very important that the computations that are needed for the optimization, can be performed within a reasonable amount of time and memory usage. This aspect of optimization theory is studied in the field of *algorithmic complexity theory*. In this field, a distinction is made between between time complexity and space complexity. The time complexity of a problem is defined by the number of steps needed in the worst case as a function of the size of the inputs. The space complexity of a problem similarly is defined by the amount of memory needed in the worst case as a function of the size of the inputs. It is very important for decision support systems to be aware of the complexity of the used optimization models, since it directly impacts the usability of the approach in an industrial context. When the complexity of an industrial application becomes too large to handle, the optimization approach is effectively invalidated. The models in this thesis are assessed with respect to their complexity, and where possible and necessary we define heuristic approaches that reduce the complexity at the expense of finding only approximations of the optimal solution.

2.6 An Overview of Research on Treating Imperfect Information

2.6.1 General Approaches

The existence of imperfect information has been investigated in many different areas and for many different purposes. In the area of handling information that is subject to imperfection, two separated fields of research have emerged. The research is divided into a field that addresses imperfect information by means of symbolic representations and a field that addresses imperfection by means of numeric representations.

In the first field of research, imperfect information is represented by means of symbolic elements. The underlying formalism provides the facilities to reason with these symbolic ele-

ments, and with that provides a mechanism to handle imperfect information. The main results in this field have been achieved by means of non-monotonic formalisms. These formalisms enable reasoning with incomplete information, and in particular facilitate the revision of conclusions based on the arrival of new information. As a result, it is possible to falsify conclusions that were reached based on imperfect information that is falsified at a later point in time. This behavior is not achievable in classical logic, since the monotonicity prevents conclusions to be withdrawn. Examples of non-monotonic logics are Reiter's default logic, circumscription [McCarthy1986] [McCarthy1980] and autoepistemic logic [Moore1985].

The research in the second field has aimed to describe imperfect information by means of numerical representations. The representations that are most used in this area predominantly find their origin in probability theory and fuzzy set theory. By facilitating the use of these numeric representations, it becomes possible to include them in reasoning processes. A well-known approach in this area is fuzzy logic, where partial applicability of reasoning rules is facilitated. Lofti Zadeh pioneered the work on the combination of fuzzy sets and logic to enable the representation of and inference with imperfect information [Zadeh1965] [Zadeh1983] [Zadeh1983a]. The research of Zadeh enabled the mathematical representation and manipulation of imperfect statements and inputs for reasoning mechanisms, which enhanced the possibilities for dealing with imperfect information tremendously. Fuzzy logic has found many applications, especially in the domain of control theory [Mamdami1981]. In possibility theory [Zadeh1978] the possible number represents the possibility that the element is the actual value. Other approaches in this area are based on probabilistic logics [Nilsson1986] and purely mathematical approaches such as Bayesian belief networks [Besnard1989].

The models and extensions we have described in the previous paragraph, have lead to the definition of approaches that can deal with imperfect information within a specific context. For instance, in game theory [Fudenberg1991] imperfection is identified in the decision process that underlies a game-like problem. On the other hand, in artificial intelligence [Russel1995] the focus has been predominantly on the ability to represent and reason with imperfect knowledge within logical inference mechanisms. In all these approaches, attempts are made at classifying the types of imperfect information, depending on the character and nature of the imperfection. In these classifications, the distinctions that are made correspond to the ones we have made in section 2.3. For example, in an early classification in [Bonissone1985] imperfection is divided into three categories. These are uncertainty, incompleteness and imprecision. According to this classification, uncertainty reflects a subjective opinion on the truth of a fact, which at this point is not verifiable. Imprecision corresponds to a value which can not be measured with the desired precision and therefore remains imprecise. Finally, incompleteness refers to the complete absence of a particular value that is required for the information to be useful.

For other approaches, the classification of imperfection is refined into additional categories. For example, in [Bosc1993] in addition to slight alterations to the previous definitions, two new categories of imperfection are identified, called vagueness and inconsistency. Vagueness is defined as a fuzzy description of impreciseness, which implies an extension based on fuzzy set definitions. In [Parson1996], this new category is aimed at representing vague predicates such as cheap and efficient. Inconsistency is defined to be a type of imperfect information, where two values are directly or indirectly contradicting each other. When this is the case, the imperfect information can only become perfect by resolving the conflict, for example by removing the least reliable value. From this overview it can be seen that many classifications exist of imperfect information, all of which are aimed at establishing a common understanding of the nature of imperfection. It is important to understand the type of imperfection in the available information, since the impact of, for instance, impreciseness differs from the impact of uncertainty on decision making processes.

2.6.2 Explicit Support for Imperfection in Development Methods

We have already established that imperfect information is part of our everyday life. The difficulty of managing this imperfection increases with the complexity of the activity that we have to perform. In particular design activities where many different aspects need to be considered simultaneously, such as software development, suffer from imperfect information in the design inputs. In the literature, a number of methods have been proposed, which provide support for specific design activities that are hampered by imperfect information. Generally, these methods provide an approach that extends the expressive capabilities of the development process, in order to increase the accuracy of assessments. This is achieved by adding models to design activities, which describe the important properties of imperfect information, for instance by means of probability theory or fuzzy logic. By including these models in the design process and considering them according to their underlying theories, the influence of imperfect information can be more accurately considered. While this type of approach has received little attention in the field of software development, in other disciplines, such as mechanical engineering, approaches have been defined that successfully reduced the impact of imperfect information. In the following, we briefly explore the work that has been done in this area.

While modeling attributes of imperfection in the inputs of design processes is not new, it is seldom applied in the field of software design. In [Aksit2001a] and [Aksit2001], fuzzy logic is applied to support the partial applicability of design heuristics in the Object Modeling Technique development process. By applying fuzzy reasoning techniques, the inconsistency can be controlled and maintained to a point, where it can be resolved by new design input. In [Yen1993] and [Lee2003], a framework is defined based on fuzzy logic, which can be used to model imperfect functional requirements. The requirements are specified as pre- and post-conditions for design steps. After each design step, the proposed solution can be compared with the requirements, in a manner similar to proving an invariant over a piece of source code. The resulting value then indicates to which degree the requirement holds. In [Shaw1996], credentials are introduced as a means for incremental and evolving specifications. They are defined as property lists, where for each property, such as reliability or robustness, a value is given for its specification. In addition to this, the credibility of the value is given, for example asserted or verified, which indicates to which degree the value can be trusted. The credentials are maintained and updated along the design process, so that the software designer has a good understanding of the information that is used. In this thesis, we extend on this notion by defining mathematical models to express the credibility of the information. These models can be used to systematically analyze decision alternatives and maintain the credibility information over multiple subsequent design steps.

Also in the field of computational intelligence, efforts have been made to address the problem of imperfection in software design activities. Based on specific areas support models have been proposed. For example, the work in [Mayrhauser1998] analyzes the possibilities of using neural networks for supporting software testing. The premise is that the learning capabilities of neural networks can facilitate a more effective testing approach. In [Gray1998]], a number of techniques is proposed for development effort of software products based on fuzzy logic. In [Pedrycz1998], an extension of the traditional object-oriented datamodel is proposed. This is achieved by for instance supporting imperfect input in class attributes. Also typical operations within this model is treated, such as inheritance and aggregation. In [Pedrycz1999], a granular model is proposed the capture the imperfection in the estimations of cost of software development processes. This model is augmented on the COCOMO cost estimation models and can capture the inherent fuzziness of the input variables. The assessment of software quality using techniques from computational intelligence is proposed in [Reformat2002]. In particular, genetic decision trees are used to classify software objects with respect to their quality. With this approach to describe estimated results and use it for preliminary evaluation. Finally, [Pedrycz2002] identifies the suitability of computational intelligence to support the activity of software engineering. The three main technologies of computational intelligence, neural net-

works, granular computing and evolutionary optimization, are linked to various activities in software engineering, such as cost estimation, evaluation of domain knowledge and data visualisation.

Other approaches are more generic, and focus on imperfection support during development processes in general. For instance, in [Liu2005], the decision making process is captured in a tree structure, based on decision trees. To this structure, an extension to decision trees is proposed, which describes the imprecise attitude of the decision maker with respect to risks. This is modeled using techniques from fuzzy logic, and combined with the decision optimization algorithms of probabilistic decision trees. In the field of engineering design, an approach has been proposed in [Law1995], to model imprecision in design inputs. This imperfection is captured using fuzzy set theory, and is then used to explore the possible design alternatives based on this model. In addition, the method defines means to evaluate design alternatives based on these models.

2.7 Conclusions

In this chapter, we have taken a first look at the problem of imperfect information in software design processes. Imperfect information is a phenomenon that is well studied, and in the literature a multitude of approaches can be found. However, these results have found little application in the field of software development. In this chapter, we have made a distinction between imprecise information and uncertain information, a classification that is based on the nature of the imperfection. In imprecise information, imperfection exists and is not necessarily resolved in the future. Uncertain information, on the other hand, will be resolved with certainty within a limited time. Additionally, we have made a distinction between logical imperfection and temporal imperfection, where the first is imperfection in the phenomenon that needs to be described, and the second is imperfection in the time of occurrence of a well-known phenomenon.

To assess the imperfection support in state-of-the-art software development processes, we have examined the waterfall model, the Rational Unified Process, Agile Processes and Analysis-Synthesis processes. We conclude that most modern development processes are aware of the difficulty of defining precise and accurate requirement specifications. In response to the sensitivity of the waterfall-model for imperfection in requirements and other sources within the development processes, many design processes have introduced iterative design as a countermeasure. Nonetheless, differences can be identified between the implementation of incremental design in software development processes. Where a detailed process like the RUP uses relatively long iterations, agile processes have very short iterations and synthesis analysis is somewhere in between. In addition to iterative design, analysis-synthesis approaches also introduce problem decomposition to isolate the influence of imperfection information on the software design process.

While iterative design has achieved some success in addressing changes in requirements, the influence of imperfect information on software development processes is still considerable. As we have identified in chapter 1, for iterative design to be successful, four requirements must be fulfilled. However, in a realistic design setting these requirements can not be fulfilled, which means it is dangerous to rely solely on iteration to resolve problems caused by imperfect information. Rather, the imperfection that is part of the information being used must be described and considered in the design process, in addition to using an incremental design approach.

For the description of imperfect information, we have introduced the basic concepts of three imperfection models, probability theory, fuzzy set theory and fuzzy probability theory. Each of these models describes specific types of imperfection and therefore is useful for particular types of imperfection. To support the application of these imperfection models within a specific context, such as software design, we have introduced the core concepts of optimization

based decision support models. The mathematical definition of these models ensures the proper application and reasoning with the models used for imperfection. In the following chapter we introduce our approach for dealing with imperfect information in software design processes.

CHAPTER

3

"My Lord Baron, if you wish to make the best use of my services, you must give me adequate information. Wasn't this conversation recorded?"

- From Frank Herbert's Dune [Herbert2005]

DECISION SUPPORT FOR IMPERFECT FUNCTIONAL REQUIREMENTS

3.1 Introduction

Software systems have to fulfil requirements that become increasingly more complex, and as a result the software systems themselves also become more intricate. One of the key issues for managing this increased complexity, is to have a concise and unambiguous requirement specification at the start of the software development process. However, as is identified by most modern development processes, it is typically very difficult to come to such a specification. Software engineers and stakeholders can only form a partial view of the complete system at the early stages of software development and as a result requirement specifications contain ambiguities, conflicts, vagueness, etcetera. Therefore the definition of requirement specifications introduces imperfect information into the software development process.

In this chapter we describe an approach for the support of imperfect functional requirements during software design. To facilitate the use of the proposed imperfection model, we first define the *Artifact Trace Model*, which is aimed at offering decision support when trade-offs must be made between system functionality and cost of implementation. In the second part of this chapter, we extend the Artifact Trace Model with the concept of *Fuzzy Requirements*, which is used to describe the aforementioned imperfection in functional requirement specifications. The approach is illustrated by an industrial example based on a traffic management system.

3.2 Imperfect Information in Functional Requirements

During the last decades, a considerable amount of software design methods have been introduced, such as Structural design [Yourdon1979] and the Rational Unified Process [Jacobson1999]. Although there are differences among the methods, the general structure of the methods is quite similar. They all require a well-defined requirement specification, which is transformed into a system design in a number of design steps. As has been identified in [Marcelloni1999], one major problem with software design methods is the occurrence of incomplete information during the design process. While modern software design methods acknowledge the difficulty of defining "perfect" requirements, they depend on their perfection to ensure that the resulting software system precisely reflects the requirements. When at later stages the requirements change or have been misinterpreted, additional iterations and redesign is needed. The task of defining requirement specifications that are "perfect enough" is the responsibility of the stakeholders and software engineers, and to support this activity various approaches have been proposed and applied. In particular, in the field of formal specification the aim is to define requirement specifications in such a manner, that it becomes possible to formally verify the correctness of the designed system with respect to these requirements. Other approaches try to improve requirement specifications by exhaustive descriptions and abstractions to represent the concepts. While these approaches have been successful in isolated parts of software design, software development still suffers from imperfect and changing requirements. Existing approaches aim to come to a perfect set of requirements, which requires considerable effort and is only rarely achieved. We conclude that imperfect information is inherently present in all requirement specifications. By application of requirements analysis the imperfection can be resolved in parts of the requirements, but not completely removed from the requirements specification. If imperfection in requirement specifications is recognized and taken into account during the design process, it is possible to minimize the amount of incremental design steps that are needed to stabilize the software design.

3.3 Relationship Tracing for Intermediate Design Artifacts

3.3.1 Introduction

One of the big challenges in software engineering is to balance the design and implementation of the software system with budgetary restrictions and time constraints. Software engineers select the system design from several design alternatives, and try to reuse existing system parts to minimize costs and development time. Even in the case of a crisp and concise requirements specification this is a very challenging task. This is caused by the fact that costs and development time largely depend on the components that need to be implemented, while it is at the same time unclear which requirements are being implemented by the respective components. The lack of a formal trace from the requirements to the components that implement them, makes it impossible to systematically explore the alternative component sets that can be used to implement the system. This hampers in particular the possibility to analyze the trade-offs between the functionality that is provided to the customer and the components that will be implemented. In response to the increased complexity in software development there has been much research in the area of decision support approaches for software design processes [Ruhe2004]. The accuracy of the decision support approach heavily depends on how well the model represents the relevant information in the software design process, and whether the reasoning model is capable of interpreting the information accordingly. However, since decision support approaches are typically not prepared for imperfect information, the usefulness of the decision support mechanism is drastically reduced.

To resolve the problems, an approach is needed that explicitly captures the relationship between the requirement and the components that implement this particular requirement. This

relationship should enable the software engineer to trace components of the designed architecture back to the requirements from which the components originate. Additionally, the relationships should capture the reuse of components, or intermediate design artifacts, during the design of the system architecture, since these elements become vital when considering trade-offs between functionality and implementation effort. Finally, this approach should provide techniques to model imperfection in functional requirement specifications, and support reasoning with these techniques to provide better decision support. To resolve these problems, we propose the Artifact Trace Model. In the second part of this section we extend the Artifact Trace Model with fuzzy requirements, as a means to describe and reason with imperfect information. With this approach, the possibility to model and analyze imperfect information is introduced in the definition of functional requirement specifications. The Artifact Trace Model offers optimization support, which can assess the imperfect information in the design process and minimize its risk and impact.

3.3.2 The Artifact Trace Model

Our approach for dealing with imperfect information in functional requirement specifications is divided into two parts. The first part is a tracing model that addresses the need for a technique, which makes the relationship amongst intermediate design artifacts explicit. The second part extends the tracing model with a technique to describe imperfect functional requirements. In this section we present the *Artifact Trace Model* (ATM), which captures the relationships between intermediate design artifacts of subsequent design steps. This tracing model is based on design processes that follow the analysis and synthesis approach, as for instance exemplified in the Synthesis-based Software Architecture Design method [Tekinerdogan2000], known as Synbad. In Figure 2.4 in chapter 2, the design phases of Synbad are depicted schematically. In an analysis and synthesis based approach, such as Synbad, the user requirements in phase I lead to the definition of a relevant set of interrelated problems that should be solved in phase II. Based on this problem decomposition, the relevant domains of expertise are identified (commonly named solution domains) in phase III. In step IV from these domains the solution concepts are extracted that make up the system design.

From phase IV two different iterations can be done. The *refinement iteration* is the refinement of each solution concept into smaller concepts that implement the current one, which essentially repeats the design process at a lower level of abstraction. This is done by defining (lower-level) requirements for the solution concept, which are in turn mapped to problems. These problems are mapped to solution concepts. This iterative cycle can be repeated until an implementable solution is obtained. The second possible iteration is the *quality balance iteration* cycle. The current set of solution concepts is graded by making an evaluation of the specific quality attributes such as stability, performance, etcetera. These characteristics are compared to the respective quality requirements. This leads to the identification of errors in the quality balance. These errors are mapped to problems, which are in turn mapped to solution domains. From the solution domains eventually the solution concepts of higher quality are selected. This iterative cycle can be repeated until a solution of acceptable quality has been obtained, and refinement iterations can be performed.

In each step in Synbad, an intermediate design artifact (such as a requirement), is refined into new intermediate design artifacts (like for instance a set of problems that should be solved to implement a particular requirement). The ATM is a directed graph, in which the nodes are the *intermediate design artifacts* that result from the software development process. In order to describe complete traces of analysis and synthesis based design methods, the Artifact Trace Model identifies the following intermediate design artifact types:

- *Requirement*
- *Problem*

- *Solution Domain*
- *Solution*
- *Component/Class*

The *refinement* is a design step, which takes a single design artifact as input and has a set of artifacts as output. The activity of design without tracing, therefore, means that the input is replaced by the output. In the Artifact Trace Model the relation between inputs and outputs is recorded; the output nodes of a refinement become children of the input node. In an analysis-synthesis development process, the refinement steps follow the order as it is indicated in Figure 2.4, in accordance with Synbad. In this order requirements are requirements are refined to problems, problems to solution domains, solution domains to solutions and solutions to classes and.or components. Frequently in analysis and synthesis based design, the solution domain step is not made explicitly, but rather incorporated into the step from problem to solution. In the ATM the artifact relations will follow the same order. As a result of having a set of outputs, refinement in the ATM can be schematically depicted as in Figure 3.1.

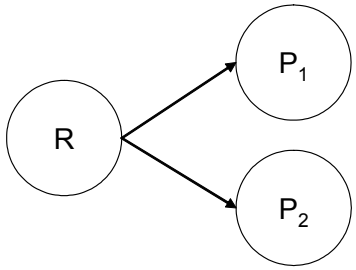


Figure 3.1 Artifact Refinement

In this picture, the requirement is refined to multiple new artifacts. This describes the situation where *all* the new artifacts must be solved in order to fulfil the originating requirement. Note that this situation does not describe alternative refinements for the input. The evaluation and selection of design alternatives is treated in chapter 4. Obviously, when artifacts from previous refinements can be reused for the refinement of other inputs, the respective nodes in the ATM are reused as much as possible. Due to the tracing of design artifacts, the Artifact Trace Model is extended as a result of every refinement step. The inputs of the refinements are not discarded, but remain part of the artifact trace. The starting point of the ATM is a set of requirements, which form the inputs of the first refinement steps, in accordance with analysis-synthesis approaches. A typical Artifact Trace with multiple refinement relations is depicted in Figure 3.2.

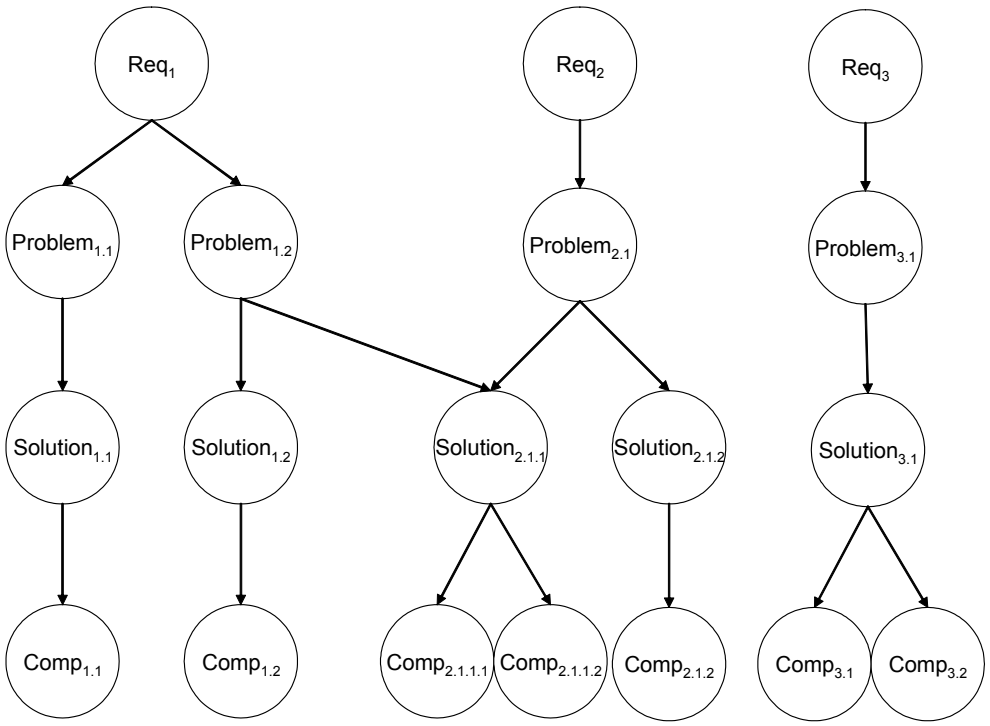


Figure 3.2 An example of an Artifact Trace

This Artifact Trace Model starts with three requirements at the top of the picture. Req_1 is refined to two problems that should be solved to fulfil this requirement, Req_2 and Req_3 are refined to a single problem. In the figure, we also see two overlapping relations: $Solution_{2.1.1}$ provides part of the solution for both $Problem_{1.2}$ and $Problem_{2.1}$, which makes this a reuse refinement. Using the traces that have been defined we now determine the components that are needed for both requirements. For Req_1 the following set of components is needed: { $Comp_{1.1}$, $Comp_{1.2}$, $Comp_{2.1.1.1}$, $Comp_{2.1.1.2}$ }. For Req_2 the following set is needed: { $Comp_{2.1.1.1}$, $Comp_{2.1.1.2}$, $Comp_{2.1.2}$ } and for Req_3 the following set: { $Comp_{3.1}$, $Comp_{3.2}$ }. Alternatively, a set of components can be traced back to the set of requirements that are implemented by it. For example, the component set { $Comp_{1.1}$, $Comp_{1.2}$, $Comp_{2.1.1.1}$, $Comp_{2.1.1.2}$ } implements Req_1. Req_2 is not implemented since $Solution_{2.1.2}$ is not implemented by this set of components.

3.3.3 Trade-offs between Stakeholder Desires and Implementation Effort

The Artifact Trace Model allows the software engineer to determine the components that are needed for the implementation of any set of requirements. This is achieved by tracing down all the components that can be reached from a set of requirements in the Artifact Trace Model. We can use this tracing capability of the Artifact Trace Model to perform a trade-off analysis between the requirements that will be implemented and the estimated effort for the implementation of the needed components. To facilitate the trade-off analysis, we define two additional properties, *cost* for components and *stakeholder interests* for requirements. The cost property of components corresponds to the estimated effort that is needed to implement the component. Stakeholder interests are numeric values of requirements that represent attributes of interest for the stakeholders. Examples of stakeholder interests are relevance, urgency, desirability and applicability. Stakeholder interest values can be defined for multiple interests of the stakeholders, which means that all requirements are tagged with a numeric value for each stakeholder interest.

To illustrate the use of stakeholder interests, suppose that the requirements in Figure 3.2 are tagged with a number that indicates their relevance and a number that indicates their desirability: Req_1 has a relevance 1 and desirability 1, Req_2 has a relevance 2 and desirability 1 and Req_3 has a relevance 1 and desirability 2. This means, for instance, something like: Req_2 is twice as relevant as Req_1 and Req_3 is twice as desirable as Req_1. The stakeholder interest values enable the trade-off between implementing a set of requirements and the costs or stakeholder interests that are attributed to the implementation effort. The trade-off implies that only a subset of the identified requirements will be implemented. The problem for a trade-off therefore can be states as: which subset *S* of the set of requirements should be implemented in order to attain the optimal result.

We define two functions for the evaluation of a subset S of the initial set of requirements *R*. Firstly, *Cost(S)* corresponds to the sum of the costs for implementation of the components that are needed for the requirements of S. Secondly, *StakeholderInterest(x, S)* corresponds to an aggregated value of the stakeholder interest values of the requirements of S for stakeholder interest x. The aggregation of this combined value can differ per design process, but typical aggregation operators for this are summation or multiplication, which can be combined with a weight per requirement. Here, we define this function to be:

$$StakeholderInterest(x, S) = \sum_{r \in S} r_x = \sum_{r \in R} p(r) r_x$$

In this definition, r_x is the stakeholder interest value for stakeholder interest x for requirement r and *p(r)* corresponds to the membership of r in S. When we consider the requirements of figure Figure 3.2 with the values for relevance and desirability as defined above we can, for instance, make the following computations:

StakeholderInterest(Relevance, { Req_1, Req_2 }) = 1+2 = 3

StakeholderInterest(Desirability, { Req_2, Req_3 }) = 1+2 = 3

StakeholderInterest(Relevance, { Req_1, Req_2, Req_3 }) = 1+2+1 = 4

By using the combined stakeholder interest values, it is possible to optimize the subset of requirements that will be provided with respect to a certain *optimization goal*. This goal can for example, be to maximize relevance or to minimize costs. The definition of the goal function *G* can be configured to best represent the optimization problem. For example, G might be a weighted average of stakeholder values:

$$G(S) = \sum_{x} \mu(x) StakeholderInterest(x, S)$$

where $\mu(x)$ is the weight of the stakeholder interest x. Another possible goal would be to minimize the cost of the system:

$G(S) = Cost(S)$

We define the trade-off analysis as an optimization problem as follows: For a system design with requirements set *R*, the set of requirements *S* is optimal with respect to the goal *G* under a set of restrictions, when:

1. S satisfies every restriction in the restriction set
2. If a set S' exists that satisfies the restrictions in the restriction set, then $G(S) \geq G(S')$ (maximization) or $G(S) \leq G(S')$ (minimization)

In this definition *G(S)* corresponds to the goal value of the set *S*. The restrictions that are imposed on the optimization can be of two types:

$$Cost(S) \leq c$$
$$StakeholderInterest(x, S) \geq c(x)$$

which respectively express that the cost of the system should not exceed c, and that the aggregate stakeholder interest value for interest x should be at least the threshold value c(x). The variables of the optimization problem are p(r), which takes values zero (requirement not implemented) and 1 (requirement implemented). The goal function and the restrictions are linear expressions on these variables. This means, that the optimization problem can be solved by 0-1 programming [Foulds1984]. While in section 3.4 we apply the Artifact Trace Model approach to the Traffic Management System example with only one stakeholder interest (see below), in section 3.5.2 we elaborate on the concept of multiple stakeholder interests and extend it with *fuzzy requirements*.

3.4 Case Study: The Traffic Management System

To demonstrate the application of the Artifact Trace Model and the trade-off analysis, we apply it to an example called the Traffic Management System. We revisit the results of this section in section 3.6 to demonstrate the expansion of the Artifact Trace Model with fuzzy requirements.

3.4.1 Example Case: The Traffic Management System

The Traffic Management System (TMS) is a regulation system, designed to monitor and regulate the traffic flow on a national scale. To utilize the infrastructure fully and to plan the future of the traffic systems, a new TMS is being developed. The system is supposed to provide the necessary technical support for monitoring, controlling, managing, securing and optimizing the traffic flow effectively. Since this scale and scope of the TMS is too large to consider completely here, we will focus on the section which handles task allocations based on scenarios and available traffic information. The description that is provided by the stakeholders for this particular part is the following:

> *"The TMS should provide assistance when the traffic flow is limited. It is the job of the TMS to support operators to coordinate the activities that should reset the traffic flow to its "normal" state. To achieve this, the TMS should support the action coordination for traffic flow normalization. This is done by allocating tasks and scenarios to system operators. The Task Allocation part should gather and store information about traffic in its direct and indirect geographical vicinity. To communicate the tasks and actions, the TMS should be able to access its connected roadside systems. In addition, the TMS should support systems operators in identifying tasks and actions that will normalize traffic flow as fast as possible."*

We summarize the functional requirements for the TMS from this specification as follows:

1 *The TMS must support displaying relevant information to the users of the TMS*

2 *There should be an explicit, convenient model of tasks and scenarios*

3 *The system must support action coordination for optimal normalization of traffic flow*

4 *The system should support task allocation*

5 *Contextual Information should be accessible*

6 *The TMS should be able to communicate with the roadside system*

These requirements describe the section of the system which is particularly aimed at task allocation and the communication with to the outside world. Obviously, for a system that is responsible for regulating traffic flow, it is very important that the system adheres to the described requirements to ensure traffic safety. Assume that after a first evaluation of the requirement specification, the software architects have come up with an abstract architectural design of the Traffic Management System, which is depicted in Figure 3.3.

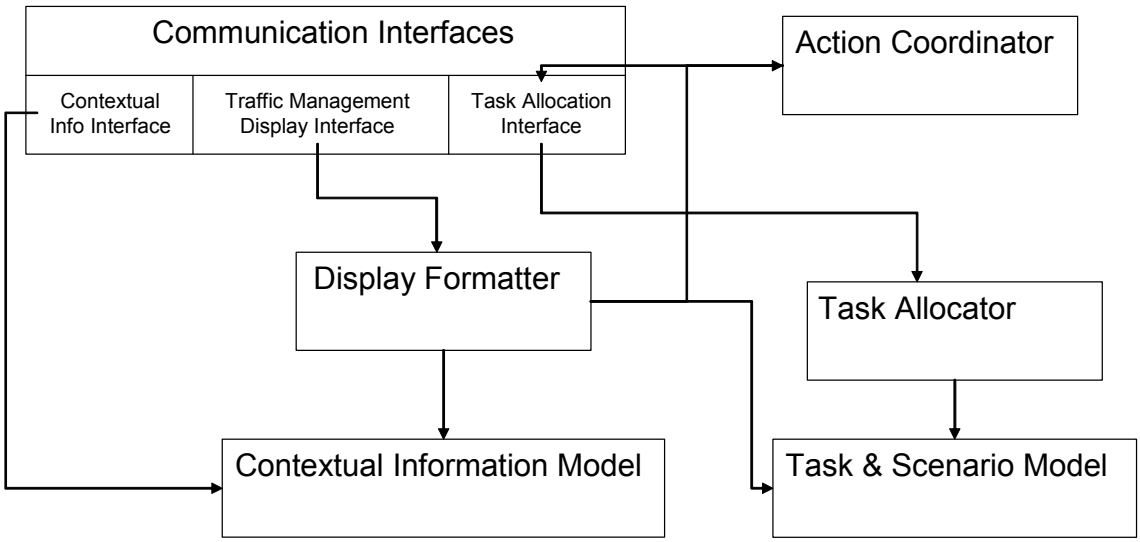


Figure 3.3 Abstract Architecture for the Traffic Management System

The conceptual model in this figure describes an abstract architecture of the Traffic Management System. In this architecture, the functionality is divided into several abstract entities, which will contain the most important functionality in the complete system. These entities are represented by boxes in the figure. For example, the box *Task & Scenario Model* is the abstract entity that represents all the classes and components that will be defined to model tasks and scenarios. The arrows between the boxes indicate the existence of a "usage" relationship between these entities. For example, the *Action Coordinator* uses the *Task & Scenario Model* to describe actions in terms of tasks and scenarios. The sub-boxes in the *Communication Interfaces* indicate individual interfaces, but they have been grouped for the sake of readability.

In the abstract architecture, six elements are identified for the Traffic Management System, which relate to the concepts described in the requirement specification. The tasks and scenarios for the normalization of the traffic flow are modeled with the *Task & Scenario Model*. Contextual information regarding the current state of the traffic is modeled with the *Contextual Information Model*. Task allocations are handled by the *Task Allocator*, which retrieves the information from the Task & Scenario Model. The *Display Formatter* transforms the information that needs to be displayed to the users of the TMS to a displayable form for the *Traffic Display System*. Finally, the *Action Coordinator* determines and coordinates the actions that

should be taken to normalize traffic flow. The *Communication Interfaces* enable the interaction with the system from both inside and outside the TMS.

In the case that traffic flow is hindered, the TMS will initiate a traffic flow normalization. Based on this conceptual model this is achieved by defining a number of tasks and scenarios using the Task & Scenario Model. In the next step, the Action Coordinator allocates these tasks and scenarios to system operators using the Task Allocator. After this, the information from the roadside system is collected and stored in the Contextual Information Model. Finally, the roadside receives the information and is normalized by sending the actions to the display using the Display Formatter. This process is repeated until the traffic flow has been normalized.

3.4.2 The Architectural Design of the Traffic Management System

The next step in the design of the Traffic Management System is to identify the components that are needed to realize the proposed structure of the abstract architecture in Figure 3.3. To design the refined architecture of the Traffic Management System, we apply a synthesis-analysis approach, which means that first the requirements are transformed into a set of problems that need to be solved. For each of these problems a solution domain and a solution is identified. Finally, from these solutions the refined architecture is defined. In Table 3.1 the first step is described, where for each requirement the relevant problems are identified. Note that the table essentially defines the relationships between the requirements and problems.

Table 3.1 From Requirements to Problems

Requirement	Problems to be solved
1	P1 How do we display information? P6.1
2	P2.1 How do we express Tasks and Scenarios in an extensible manner? P2.2 How do we capture Task and Scenarios in a portable and exportable manner?
3	P3.1 How do we normalize traffic flow with actions? P3.2 How do we rate normalizations with respect to each other?
4	P2.1 P4.1 How do we support a generic Task Allocation Support Model? P4.2 How do we offer this information?
5	P5.1 How do we support interaction with the system? P5.2 How do we define a generic model that captures contextual information for external usage?
6	P6.1 How do we make the internal data available? P6.2 How do we realize a constant and stable communication stream?

In Table 3.1, for each requirement a number of problems are identified that need to be resolved, in order to fulfil the respective requirement. These problems focus on particular aspects of the requirement, for example, for requirement 5 the first problem is to decide on the interaction mechanism, and the second problem is how this should be supported by the model. Note that a number of problems are reused for multiple requirements. For example, P2.1 is a problem that should be solved for both requirement 2 and requirement 4. This means that when P2.1 is solved, this resolves a part of requirement 2 as well as requirement 4. Also note that a specific interpretation is given to vague descriptions in the initial requirements. For example, requirement 2 requires a “convenient model“, which is interpreted as “extensible“ during the problem definition phase.

After the problem identification the design process is continued with the identification of solutions for the problems that have been found. In order to solve the problems that have been identified in Table 3.1, the software engineers use available knowledge sources on the specific areas, which are part of the applicable solution domains. By choosing solutions that can resolve multiple problems at the same time, the amount of effort needed to complete the system can be reduced. The solutions that were selected by the software architect for the problems in Table 3.1 can be found in Appendix A, Table A.2. For example, a common problem for the Traffic Management System is the communication of data. Typically, a *uniform communication interface* is a solution from which the base functionality can be reused multiple times. As a result the uniform communication interface is used to solve P6.1 and P6.2. In for example problem P2.1 there is emphasis on the extensibility of the task and scenario model, and for P5.2 there is an emphasis on genericity of the model. By capturing the models in XML and reusing the communication facilities, these considerations can be addressed while minimizing implementation effort. The complete set of solutions that are chosen can be found in Appendix A.

As the final step, the software architects refine the selected solutions to a set of components and place them in the abstract architecture, which localizes the functionality that is needed to implement the system. Since the TMS is decomposed into solution parts, the structure of the refined architecture is largely known. However, since a number of solutions are too large to fit into one component, and other functionality can provided by commercial components, the components form a more refined model of the TMS system. The refinement step from the selected solutions to the components that implement them is described in Appendix A, Table A.3. In this table for each component also the implementation effort is estimated in person-months. The software engineers now have completed the architectural design of this part of the Traffic Management System. When we depict the relationships defined in the various tables in an Artifact Trace Model we get the picture of Figure 3.4.

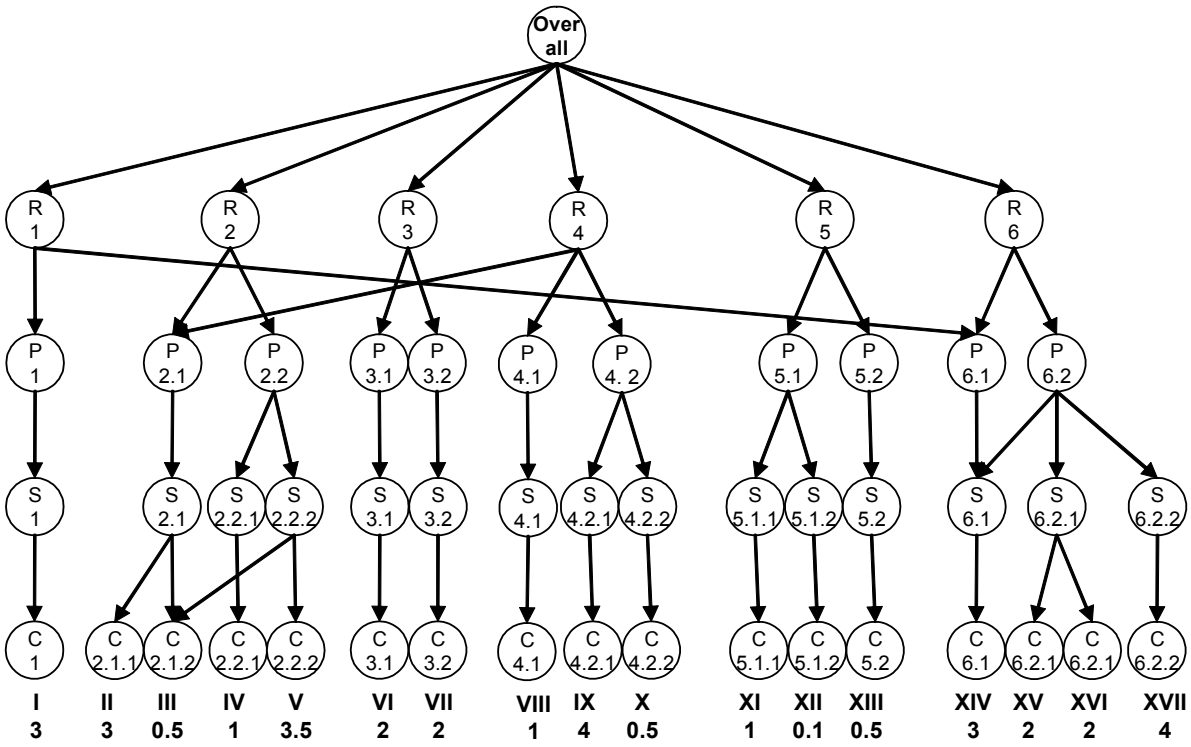


Figure 3.4 Artifact Trace Model for the Traffic Management System Architecture

In this figure all the relationships between the intermediate design artifacts are depicted. In case of shared relationships the node representing the shared artifact is also shared by its parents. In the figure, all the information of the tables is included, as well as the expected implementation effort of the components. For example, for problem R1 two subproblems were identified: P1.1 and a shared problem P6.1. We can also see that P1.1 is resolved by the component C1, which takes 3 person-months to implement. When we place the components in the abstract architecture according to its structure, this results in the following picture:

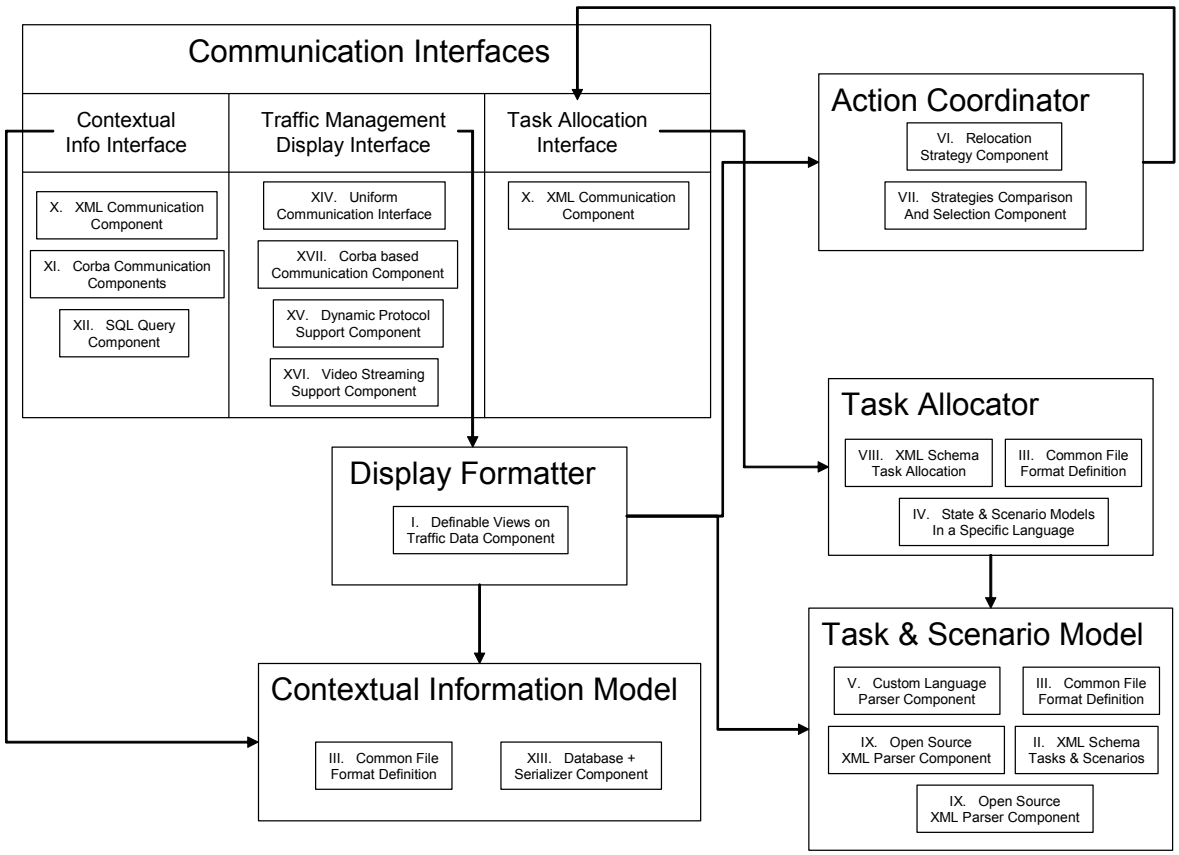


Figure 3.5 Refined Architecture for the Traffic Management System

In Figure 3.5, the identified components are placed in the abstract architecture. In this figure, the components represent implementation units, which are placed into the respective parts of the abstract architecture. The resulting refined architecture is an implementation of the requirements specification at the beginning of the paragraph. Obviously, all the components are needed to implement the requirements for the Traffic Management System, but for illustrational purposes we analyze a number of requirement sets in Table 3.2. First we assume that we have identified the stakeholder attribute "relevance" and since the stakeholders consider all requirements to be equally relevant, the relevance value for each requirement is set to 1. Additionally we define the overall relevance for a system to be equal to the number of implemented requirements.

Table 3.2 System Evaluations of the TMS

Requirements	Components	Relevance	Cost
1, 2, 3, 4, 5	I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, XIII, XIV	5	22.1
1, 3, 4, 5, 6	I, II, III, VI, VII, VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, XVII	5	28.1
2, 3, 5, 6	II, III, IV, V, VI, VII, XI, XII, XIII, XIV, XV, XVI, XVII	4	24.1
1, 2, 3, 4	I, II, III, IV, V, VI, VII, VIII, IX, X, XIV	4	23.5
1, 3, 5, 6	I, VI, VII, XI, XII, XIII, XIV, XV, XVI, XVII	4	18.6

Table 3.2 describes five partial systems by summing up the requirements that are fulfilled by implementing the respective components. In the first column of Table 3.2, the requirements are displayed that are implemented for the system alternative. The second column contains the components needed for the implementation of the requirement set. The third column indicates the relevance of the system alternative, which is in this case equal to the amount of requirements that are implemented since all requirements are equally relevant. Finally, in the fourth column the cost of implementing the set of components is displayed, which is equal to the sum of the implementation efforts. Assume that these 5 systems satisfy the set of restrictions. In section 3.3.3 we defined two configurations, cost-based optimization and relevance-based optimization. Based on cost minimization the fifth option should be selected. However, based on relevance, either the first or the second option should be selected. In a typical industrial application the relevance evaluation can be more complex by including, for instance, weights on requirements. In section 3.6, we extend the Artifact Trace Model with fuzzy requirements and revisit the optimization analysis for the Traffic Management System.

3.5 A Model for Imperfect Requirements based on Fuzzy Sets

3.5.1 Imperfect Information in Functional Requirement Specifications

While modern software design methods acknowledge the difficulty of defining “perfect“ requirements, they depend on the perfection of the requirements to ensure that the resulting software system precisely reflects the requirements. When at later stages the requirements change or have been misinterpreted, additional iterations and redesign are needed. The task of defining requirement specifications that are “perfect enough“ is the responsibility of the stakeholders and software engineers, and to support this activity various approaches have been proposed and applied. In particular, in the field of formal specification the aim is to define requirement specifications in such a manner, that it becomes possible to verify the correctness of the designed system with respect to these requirements. Other approaches try to improve requirement specifications by exhaustive descriptions and abstractions to represent the concepts. While these approaches have been successful in isolated parts of software design, software development still suffers from imperfect and changing requirements. Existing approaches aim to come to a perfect set of requirements, rather than acknowledge the fact that imperfect information is inherently part of requirement specifications. As a result, it is impossible to define a complete set of requirements without any imperfection, even when using these approaches.

The cause of the imperfection in requirement specifications is two-fold. Firstly, the initial requirements are defined in an early phase of the design process. At this point it is very difficult for both the stakeholders and the software engineers to precisely visualize the system upon completion. This is exemplified by changes that are made to the requirements along the design process, and the occurrence of new requirements. Secondly, requirements are normally described in natural language, which typically suffers from imperfection. Many terms in natural language have multiple meanings, are ambiguous or vague. The consequence is that the system designers should either clarify the requirements with the stakeholders, or interpret the imperfect requirement(s). However, neither approach guarantees a satisfactory result, since stakeholders might be unable to clarify the requirements, and designers can interpret imperfect requirements differently from stakeholders. Using formal languages does not resolve the situation, since defining a formal specification requires a very clear understanding of the statements that should be formalized. Formal methods can only be used if the information we are using is perfect, which makes it impossible to resolve imperfect information in this manner.

We conclude that imperfect information is inherently present in all requirement specifications. By application of requirements analysis the imperfection can be resolved in parts of the requirements, but not completely removed from the requirements specification. However, since software development methods depend on the perfection of the requirements, all design decisions that are taken, based on imperfect information that is assumed to be perfect, become particularly vulnerable to redesign. In this section we extend the Artifact Trace Model with *fuzzy requirements*, a technique that allows software engineers to capture imperfect requirements by means of fuzzy set based descriptions. In addition, the trade-off capabilities of the Artifact Trace Model are extended such that it can work with these fuzzy requirements.

3.5.2 The Fuzzy Requirement Concept

We have identified that modern software design processes generally require perfect requirements for the design process to commence. However, as opposed to assuming and/or requiring that the initial requirement specification only contains perfect information, a more fitting solution is to recognize imperfection where it occurs and capture its properties. By considering the imperfect information in the design process and taking its potential consequences into account, the software design is less vulnerable for the alternative interpretations of ambiguous statements. An ambiguous statement in a requirement has multiple possible interpretations, most of which do not correspond to the intentions of the stakeholder. Considering only one of the possible interpretations, when we are not sure it reflects the stakeholder's interests, can then introduce a considerable risk. Therefore, instead of intuitively assuming one interpretation that *should* correspond to the stakeholder's intentions, we propose to include a range of possible interpretations in which the correct interpretations are *most likely* included. By including the most likely interpretations in the design process, the resulting software system is more likely to fulfil the desires of the stakeholder. To achieve this, we define the concept of a fuzzy requirement.

We assume that a crisp (i.e. perfect) requirement is an element of a universe U, where U is the set of all possible requirements. For instance, specification of the set {A, B, C} corresponds to the requirement specification: *"I need requirements A, B and C to be fulfilled and no other from the universe U"*. In the case that one or more requirements in this set are imperfect, they can be replaced by a *fuzzy requirement*. We define a fuzzy requirement to consists of the specification of a fuzzy set FS on U. The degree of membership of some element in the fuzzy set describes the degree to which this particular element is considered as the correct interpretation of the imperfect requirement at the current point in time. Therefore, this number is indicated by the stakeholder, for instance during a meeting in which the alternatives are evaluated.

For example, suppose a stakeholder asks for *"I. a convenient model"* in the requirement specification. The requirement set representing this specification then is $\{ I \}$. This requirement con-

tains an ambiguous statement since it is not clear what "convenient" exactly means, and as a result this requirement is classified as imperfect. We can interpret this requirement in a number of ways, such as:

1 An easily understandable model (0.4)

2 An easily modifiable model (0.6)

3 An easily portable model (0.8)

Each of these interpretations are evaluated by the stakeholders, with respect to how well they think the respective interpretation reflects the imperfect requirement. Between parentheses we have indicated the degree of membership in the fuzzy set, which represents this feedback from the stakeholder. From this point in the requirement specification the imperfect requirement is replaced with a fuzzy requirement containing the interpretations and their individual evaluation. The requirement specification *I* is thus replaced by *{1/0.4, 2/0.6, 3/0.8}*.

3.5.3 Fuzzy Requirements in the Artifact Trace Model

In the fuzzy requirement concept an imperfect requirement is replaced by a number of possible interpretations, each of which is tagged with value between zero and one corresponding to a particular stakeholder interest. By treating these interpretations as normal "perfect" requirements, software engineers are able to continue the development process as usual. However, since these requirements are interpretations of a single fuzzy requirement, they are modeled different from normal perfect requirements in the Artifact Trace Model. Traditional perfect requirements are modeled by singular nodes in the Artifact Trace Model. A fuzzy requirement essentially fulfils the same role in an Artifact Trace Model, since it represents a (vague) request from the stakeholders. As such a fuzzy requirement is also included as a singular node. However, instead of a traditional perfect requirement a fuzzy requirement node is related to child artifacts that represent the alternative interpretations that have been identified. In addition to each interpretation node the stakeholder interest values are attached. A typical picture for a fuzzy requirement and its alternative interpretations is displayed in Figure 3.6.

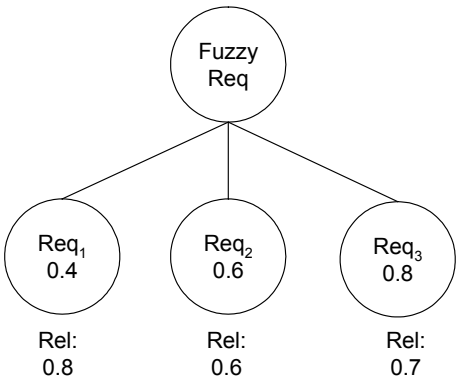


Figure 3.6 Alternative Interpretations

In this figure, the three requirements, Req_1, Req_2 and Req_3, as defined in the previous paragraph, are depicted as interpretation child nodes of the fuzzy requirement. In addition, a value for the stakeholder interest relevance is given as they have been indicated by the stakeholders. From this point the interpretation nodes are intermediate design artifacts for which we can now trace the relationship between requirements and the components that implement them in the same manner as with perfect requirements. The four nodes in this picture represent the imperfect requirement.

For the continuation of the development process the imperfect requirement is replaced by the fuzzy requirement and its interpretations, which are now considered in the same manner as perfect requirements. All the interpretations in the Artifact Trace Model can now be refined during the development process like normal requirements, and the relations can be used to trace an interpretation to the components that implement it. The resulting system design eventually will support all the perfect requirements as well as all the interpretations that have been defined for the individual fuzzy requirements. However, for the system to fulfil the requirements not every interpretation needs to be included until the completion of the design process and typically this is also not desired by the stakeholders and the software engineers. During the course of the development process both stakeholders and software engineers can come to a better insight on the system, which enables them to clarify previously imperfect requirements. Since the Arti-

fact Trace Model indicates which intermediate design artifacts are related to a particular interpretation, it is possible to remove all "faulty" interpretations from the model without removing vital artifacts for other requirements. Note that according to this model, a crisp (non-fuzzy) requirement is its own interpretation, which is indicated like a requirement with one interpretation that has a membership value of 1.

However, it can be desirable to remove interpretations from the Artifact Trace Model even when there is no new insight on how to interpret imperfect requirements. This is especially the case it is no longer feasible to maintain the broad design including fuzzy requirements from, for example, a cost or workload perspective. The choice of which interpretations should be maintained in the design is non-trivial since it is possible to derive a multitude of possible systems, each with their own cost and stakeholder interest values. For example, in Figure 3.6 the fuzzy requirement has three interpretations Req_1, Req_2 and Req_3. This fuzzy requirement can now be (partially) implemented by implementing one or more of these interpretations. This means that seven possible implementations can be identified: *{Req_1}, {Req_2}, {Req_3}, {Req_1, Req_2}, {Req_1, Req_3}, {Req_2, Req_3}* and *{Req_1, Req_2, Req_3}* when we discard the possibility of implementing none of the interpretations. Obviously, maintaining all interpretations adds to the development effort and therefore will be more expensive, while implementing a limited set of interpretations will result in a system that likely corresponds less to the desires of the stakeholders. Note that this problem differs from the problem described in section 3.3.3, where the problem was to determine the best subset of all requirements that needed to be implemented. Here, the best subset of all interpretations must be found, which at least has one interpretation for each fuzzy requirement.

To compare the possible designs that can be derived from the interpretations of a fuzzy requirement, we use the membership values that are given to its interpretations. The goal function remains the same:

$$G(S) = \sum_{x} \mu(x) StakeholderInterest(x, S)$$

In section 3.3.3 we have introduced the concept of stakeholder interest, which can, for instance, be relevance or urgency. A stakeholder interest was represented by a number that was attached to a requirement. For a requirement specification that contains fuzzy requirements we define all stakeholder interest values to be between zero and one. For fuzzy requirements for which multiple interpretations are implemented, we define the stakeholder interest value to be the algebraic sum of the resulting values of all its interpretations. The algebraic sum of two numbers A and B is defined as *AS(A, B)=A+B-AB*. Note that this the same as the standard definition for the probabilistic sum. Since membership values of fuzzy sets are always numbers between zero and one, the algebraic sum ensures that fuzzy requirements do not have a stakeholder interest value larger than one. The StakeholderInterest function becomes:

$$StakeholderInterest(x, S) = \prod_{fr \in FR} StakeholderInterest'(x, fr)$$

where

$$StakeholderInterest'(x, fr) = AS \{ \mu(x) r(x) \mid r \in fr \wedge r \in S \}$$

Here, FR is the set of all requirements, both fuzzy and crisp. For example, when we implement the fuzzy requirement in Figure 3.6 by implementing the components for Req_1 (0.4) and Req_2

(0.6), the stakeholder interest value of Req_1 is 0.32 (0.4*0.8) and Req2 is 0.36 (0.6*0.6). The overall value is AS(0.4*0.8, 0.6*0.6) = 0.565. If this value represents relevance, this means that the implementation of the fuzzy requirement with these interpretations according to the stakeholder has a relevance of 0.565. Obviously, other choices of the t-conorm for calculating stakeholder interest values for fuzzy requirements can be used, such as the maximum, the bounded sum (min(1, A+B)) or the drastic union [Klir1995]. Each of these operators can be used in the situation that demands the particular behavior. Generally, the maximum operator is used, however for our example this would mean that we can not distinguish between systems that include only the most relevant interpretation and systems that include additional interpretations. Therefore the algebraic sum was chosen.

Since the defuzzification of the fuzzy requirements is done with respect to the entire system, we define the value of a stakeholder interest for a complete system to be the product of all values of the individual requirements that are implemented. By this definition it becomes imperative that all perfect requirements are implemented and that at least one interpretation for each fuzzy requirement is implemented, since otherwise the multiplication will contain a zero and as a result the overall value will become zero. To demonstrate the comparison of system design that can be derived from fuzzy requirements, suppose we have the requirement specification that is depicted in Figure 3.7.

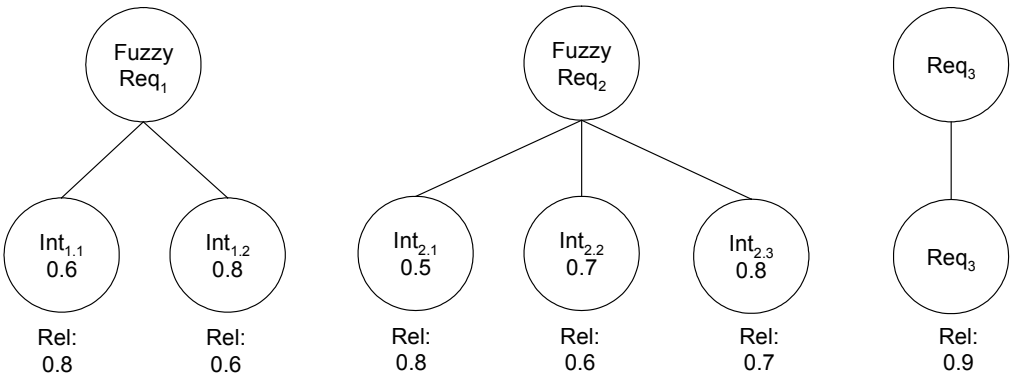


Figure 3.7 A fuzzy requirement specification

In this picture there are two fuzzy requirements, *Fuzzy Req1* and *Fuzzy Req2*. Fuzzy Req_1 has two interpretations, $Int_{1.1}$ and $Int_{1.2}$, and Fuzzy Req_2 has three interpretations, $Int_{2.1}$, $Int_{2.2}$ and $Int_{2.3}$. Their membership degrees and their relevance are included in the picture. Finally, Req_3 is a crisp requirement with relevance 0.9. Suppose we want to compare the system design that includes $\{ Int_{1.2}, Int_{2.1}, Int_{2.3}, Req_3 \}$ and the system design that includes $\{ Int_{1.1}, Int_{1.2}, Int_{2.3}, Req_3 \}$. The overall relevance value for the first system is computed as follows: the relevance of Fuzzy Req_1 is 0.48. For Fuzzy Req_2 the relevancy is AS(0.4, 0.56), which equals 0.736. Finally, for Req_3 the relevancy is equal to 0.9. The relevancy for the entire system now is the result of the multiplication of these three values: 0.48*0.736*0.9 = 0.318.

Computed in analogue fashion the overall relevance for the second system is 0.730*0.56*0.9 = 0.368. Based on relevance therefore the second system would be a better choice. Typically in this type of trade-off other considerations will be included such as the cost of the required components. In section 3.5.4 we elaborate on the inclusion of fuzzy requirements in the trade-off analysis of the Artifact Trace Model.

In the example above we have focused on a single stakeholder interest evaluation that each interpretation has received. Obviously, it is possible to attribute multiple membership degrees to an interpretation, such as one for relevance, one for urgency, etcetera. Using the approach defined above this means the evaluation of a system will result in an overall value for each

stakeholder interest. The combination of these values into an overall evaluation can be achieved by, for example, a weighted average that reflects the preferences of the stakeholder with respect to the interests.

3.5.4 Trade-off Analysis with Fuzzy Requirements

In the previous section we have identified that at times it can be desirable to remove interpretations from the Artifact Trace Model. This can be caused by either resolving imperfection or a necessity to minimize the remaining design effort. In the first case essentially an iterative design step is initiated, where a change in the available information facilitates a new design. In the case that imperfection is resolved to an interpretation that is present in the design, the iterative step exists of removing the superfluous interpretations of the fuzzy requirement. The traces in the Artifact Trace Model can be used to remove intermediate design artifacts that are no longer needed. When the correct interpretation is not part of the system, the trade-off analysis of the Artifact Trace Model can be used to minimize to a system with the most desirable properties. Therefore, the assessment of system designs can be used to coordinate the incremental steps of the software development process. For the purpose of assessing the system designs that can be derived from interpretations of a single fuzzy requirement, we have defined how the stakeholder interest is computed for fuzzy requirements. These definitions enable us to reuse the optimization definitions we have made in section 3.3.3, without the need for additional definitions. As indicated earlier, typically the assessment of the system is a trade-off between stakeholder interests and the expected cost for the implementation of the system, which means the optimization configurations of cost-minization and stakeholder interest-maximization remain applicable.

Both configurations search for a particular optimal system among all possible systems that can be derived from the set of crisp and fuzzy requirements. Since the perfect requirements all need to be implemented, the amount of interpretations for the fuzzy requirements present in the Artifact Trace Model determine the complexity of the optimization. The amount of possible designs that can be derived from a fuzzy requirement is equal to the number of subsets that can be taken from the set of interpretations minus one (since every fuzzy requirement must be implemented by at least one interpretation the empty set is not allowed). In addition, each of these possible subsets needs to be combined with each possible subset of the other fuzzy requirements, which means the amount of system designs grows exponentially with the amount of interpretations for the fuzzy requirements. The amount of system designs that can be derived from an Artifact Trace Model with *n* fuzzy requirements equals:

$$\prod_{i=1}^{n} (2^{m_i} - 1), \text{ where } m_i \text{ is the amount of interpretations for fuzzy requirement } i$$

For the application of the optimization in industrial settings, an exponential complexity becomes unmanageable. To reduce the complexity of the optimization, we propose the use of a heuristic approach when evaluating the system designs. This approach is based on the systematic removal of alternative interpretations of fuzzy requirements until a particular stopping criterion is fulfilled. We distinguish two cases:

Cost minimization

In the case of cost minimization, we take the system that contains all interpretations as the starting point. This system satisfies all restrictions on the stakeholder interest values, since these now have their highest possible value. Now, all systems that result from removing one of the interpretations are evaluated. From all the systems that satisfy the restrictions on the stakeholder interest values, the system is selected that has the lowest costs. The removal of interpretations for this system is repeated. The removal of interpretations and the selection of systems

is done until no interpretation can be removed without the resulting system violating one or more restrictions. The current system then is the heuristically cost-optimal system.

Stakeholder Interest Value Maximization

The second case, an aggregated stakeholder interest value is maximized while the cost does not exceed a particular threshold. In this case, we also start with the system that contains all interpretations. At this point the cost will exceed the threshold and all the restrictions on stakeholder interest values are satisfied. All systems that result from removing one of the interpretations are evaluated. The system that satisfies all restrictions and has the highest value G(S) for the stakeholder value of interest is selected. This is repeated, until a system is found for which the costs lie below the threshold. This system then is the heuristically optimal system for the stakeholder interest value.

The complexity of the heuristic optimization is reduced since at every step one interpretation is removed, which reduces the amount of systems in the next step. In a worst case scenario this heuristic approach is faced with a quadratic complexity.

3.6 The Traffic Management System Revisited

3.6.1 Architecture Design with Fuzzy Requirements

The design process in section 3.4.2, at first glance, does a fairly good job at designing the system architecture from the initial requirements. However, the initial requirements are imperfect in several definitions. For example, the second requirement prescribes that there must be an explicit and convenient model of tasks and scenarios. However, the term “convenient” can imply completely different solutions from the user point-of-view and the software designer point-of-view. Also, in the fourth requirement, for instance, it is described that the system must be able to communicate with the roadside system. However, it is not described what kind of communication is needed, and whether this is single- or bidirectional. Whether the resulting architecture reflects the stakeholder desires therefore very much depends on whether the software engineers have chosen the correct interpretations for the imperfection.

For our example, suppose the software architects have identified imperfection in the functional requirement specification of the Traffic Management System. In particular, the requirements 2, 4, 5 and 6 contain much ambiguity in their description, therefore the software architects decide to replace them with fuzzy requirements as described in the previous section. In addition, in accordance with the stakeholders, the interpretations are tagged with a number between 0 and 1 (i.e. its membership value). Furthermore, the keep the example understandable, all interpretations receive a relevance of one. In Table 3.3, the requirement specification including fuzzy requirements is described.

Table 3.3 Interpretations of Imperfect Requirements

Requirement		Membership
1	The TMS must support displaying relevant information to the users of the TMS	1
2	There should be an explicit, convenient model of tasks and scenarios	
2.1	There should be an easily extensible model of tasks and scenarios	0.8
2.2	There should be an easily understandable model of tasks and scenarios	0.9
2.3	There should be an easily exportable and portable model of tasks and scenarios	0.6
3	The system must support action coordination for optimal normalization of traffic flow	1
4	The system should support task allocation	
4.1	The system should support user extensible task allocation profiles	0.6
4.2	The system should support task allocation as individual task blocks	0.2
4.3	The system should support task allocation with automated decision support	0.9
5	Contextual Information should be accessible	
5.1	Contextual Information should be accessible internally in a generic format	0.7
5.2	Contextual Information should be accessible externally at an interface in a generic format	0.5
5.3	Contextual Information should be accessible both internally and externally at an inter-face in a generic format	0.3
6	The TMS should be able to communicate with the roadside system	
6.1	The Traffic Management System should be able to communicate with the roadside system unidirectionally	0.3
6.2	The Traffic Management System should be able to communicate with the roadside system with flexible support for separate data formats	0.6
6.3	The Traffic Management System should be able to communicate with the roadside system for realtime video	0.8

In Table 3.3 the initial requirement specification for the Traffic Management System is indicated by the grey rows. Additionally, for the imperfection identified in requirements 2, 4, 5 and 6 the three interpretations are described together with the membership value in the second column. It can be seen that the interpretations of the fuzzy requirements are taken from multiple perspectives, and generally the stakeholder evaluation reflects this. For example, for requirement 2 the term “convenient” is interpreted as either “extensible“, “understandable“ or “portable“. From the stakeholder perspective the two first interpretations make more sense than the third interpretation, which is more applicable for the software design team. This is reflected in the stakeholder evaluation, since the membership value is higher for the first two interpretations. The actual values in this table could have been defined in many different ways. For example, if the number results from a questionnaire posed to all stakeholders, a value of 0.8 can correspond to the statement that 8 out of 10 stakeholders identified the interpretation to be applicable. Note that this does not necessarily mean that other interpretations are not applicable for the same stakeholder, which makes this evaluation distinctly different from a probabilistic model.

As in the previous design of the Traffic Management System the software architects start by identifying the problems that should be solved for each individual requirement. At this stage the fuzzy requirements are replaced by the interpretations, and these interpretations are treated as “perfect“ requirements. At a later stage the fuzzy requirements are used for the optimization of the fuzzy design. In Table 3.4 the identified problems for each requirement are described.

Table 3.4 From Requirements to Problems

Req.	Problems to be solved
1	P1.1 How do we display information? P6.1.2
2.1	P2.1.1 How do we support a generic model that captures tasks and scenarios? P2.1.2 How do we express Tasks and Scenarios in an extensible manner?
2.2	P2.2.1 How do we capture tasks and scenarios in an easily understandable manner? P2.2.2 How do we support Tasks and Scenarios while maintaining system performance?
2.3	P2.3.1 How do we capture Tasks and Scenarios in a portable and exportable manner? P2.1.2
3	P3.1 How do we normalize traffic flow with actions? P3.2 How do we rate normalizations with respect to each other?
4.1	P4.1.1 How do we support a generic Task Allocation Support Model? P4.1.2 How do we offer this information? P2.1.2
4.2	P4.2.1 How do we offer a highly composable Task Allocation Support Model? P4.2.2 How do we extract the information from the model? P4.1.2
4.3	P4.3.1 How do we provide reasoning support for Task Allocation? P4.3.2 How do we extract this information from the Reasoning System? P4.1.2
5.1	P5.1.1 How do we define a generic model that captures contextual information for internal usage? P5.1.2 How do we make this generic model available inside the system?
5.2	P5.2.1 How do we support interaction with the system? P5.2.2 How do we define a generic model that captures contextual information for external usage?
5.3	P5.1.2, P5.2.1 P5.3.1 How do we define a generic model that captures contextual information for internal and external usage?
6.1	P6.1.1 How do we realize the unidirectional communication? P6.1.2 How do we make the internal data available?
6.2	P6.2.1 How do we achieve dynamic switching of communication protocols? P6.1.2
6.3	P6.3.1 How do we realize a constant and stable communication stream? P6.1.2

In Table 3.4, the problems are defined, which should be resolved to implement the requirements. Note that the interpretations replace the actual fuzzy requirements in this design step.

At this point, also the membership degrees are not considered during the design step. These will be used during the optimization of the Artifact Trace Model. The subsequent steps corresponding to the refinements for the crisp requirements, where problems are refined to solutions, and solutions to components can be found in Appendix A, in Table A.6 and Table A.7 respectively. When the relations between the intermediate design artifacts are traced using the Artifact Trace model this results in the model depicted in Figure 3.8. In this figure, the nodes labeled “OR“ depict the imperfect requirements, and the fact that at least one of their interpretations must be implemented. In addition to the artifact relations also the estimated implementation effort for each component is indicated in person-months. Compared to the model in Figure 3.4 this model is more complex, which means that the extra interpretations have introduced additional effort for the design of the Traffic Management System. However, since it is possible to derive multiple systems from the Artifact Trace Model based on the added interpretations, the chance for a complete redesign is reduced and therefore also the added costs for these iterations.

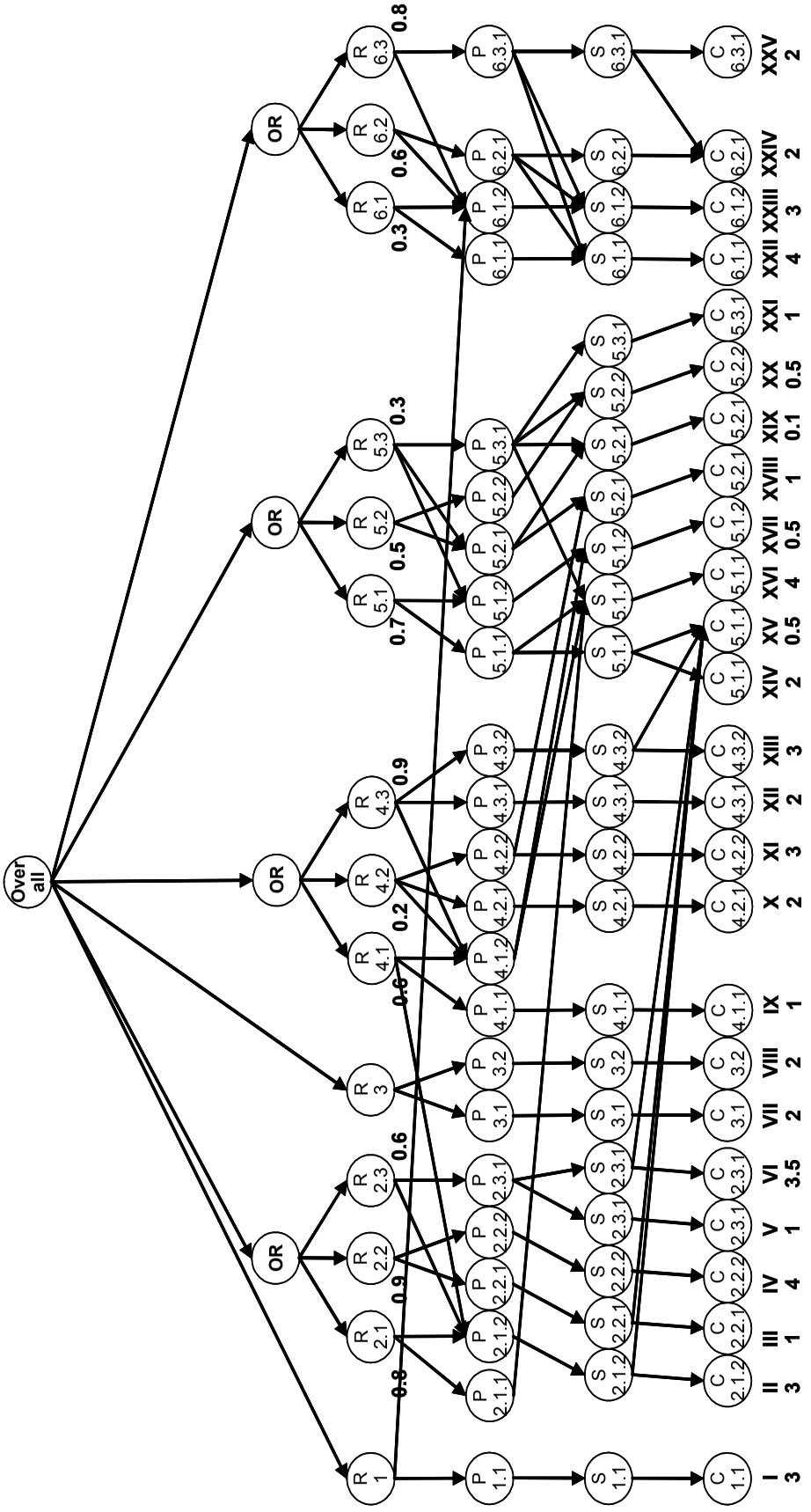


Figure 3.8 Artifact Trace Model of the TMS with fuzzy requirements

3.6.2 Optimization of Traffic Management System Architecture

As indicated in Section 3.5.3, not every interpretation needs to be implemented for each fuzzy requirement, which means that multiple systems can be derived from the Artifact Trace Model. To analyze how the refined architecture in Figure 3.5 compares to the possible systems that can be derived from this Artifact Trace Model, we will optimize the system design both for cost and relevance in the next section. When we take as a reference point the system from section 3.4, we see that the requirements that are implemented by these components are *{ R1.1, R2.3, R3.1, R4.3, R5, R6 }*. When we determine the overall relevance according to the stakeholder evaluation of these interpretations, this results in a relevance of 0.114. In addition, the cost for implementing all the components for this system is 33.1 man-months. In this paragraph we examine whether it is possible to derive systems from the fuzzy requirement design that either offer lower costs or higher relevance.

A Cost-Minimized Architecture for the Traffic Management System

First, using the optimization capabilities of the Artifact Trace Model, we identify the system with minimal costs. However, since our reference system has a relevance of 0.114, we require our optimized system to have a relevance of at least 0.114. This results in the following optimization configuration:

$$Cost(S) = Min\ \{\ Cost(S')\ |\ S' \in P(R)\,,\ Relevance(S') \geq 0.114\ \}$$

After optimization, the design that results from the Artifact Trace Model implements the following requirements: *{ R1, R2.1, R3, R4.1, R5.1, R5.2, R6.1 }*. To implement this system the following components are needed: *{ I, II, VII, VIII, IX, XIV, XV, XVI, XVII, XVIII, XIX, XX, XXII, XXIII }*. When we place these components into the abstract architecture of the ATM, we get the picture of Figure 3.9.

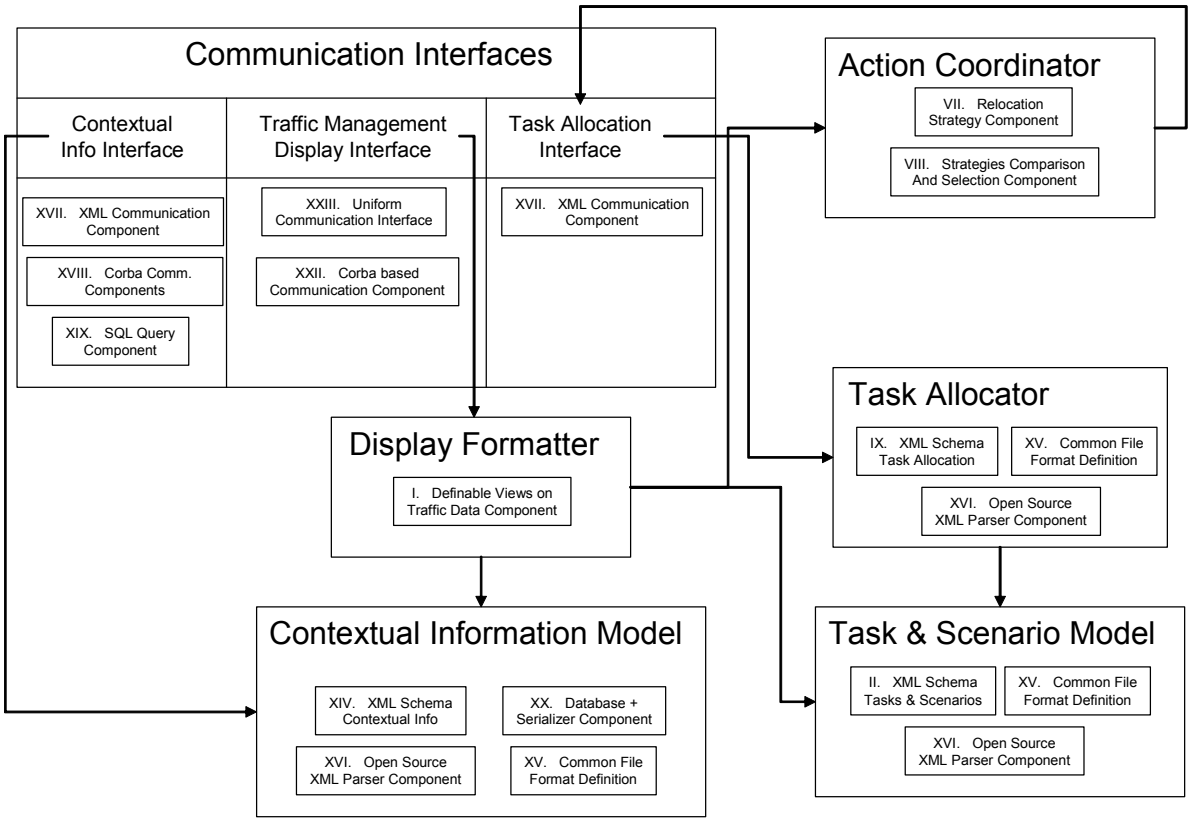


Figure 3.9 Cost minimized TMS architecture

In this figure, the identified components from the optimization are placed in the abstract architecture. In this figure, the components represent implementation units, which are placed into the respective parts of the abstract architecture. From this refined architecture, it can be seen that the cost minimization in comparison to Figure 3.5 is achieved by choosing different interpretations for both fuzzy requirement 2 and 4. Additionally, the relevance is raised by including two additional interpretations for requirement 1. As a result, this system has a relevance of 0.122, which adheres to our constraint of minimally 0.114. Our optimization criterion, cost, for this system is equal to 26.6, which is considerably lower than the 33.1 for the "crisp" system in section 3.4. We can conclude that the cost optimal system that is found by using the Artifact Trace Model not only exhibits lower costs than the crisp system, but also has a better relevance.

A Relevance-Maximized Architecture for the Traffic Management System

Second, we aim to maximize relevance, but like in the previous optimization the system should not exceed the cost of our reference system, which is 33.1 person-months. This results in the following optimization configuration:

$$Relevance(S) = Max\ \{\ Relevance(S')\ |\ S' \in P(R)\ ,\ Cost(S') \leq 33.1\ \}$$

The Artifact Trace Model optimization proposes a system that implements the following requirements: *{ R1, R2.1, R3, R4.3, R5.1, R6.1, R6.2, R6.3 }*. For the implementation of these requirements the following components are needed: *{ I, II, VII, VIII, XII, XIII, XIV, XV, XVI,*

XVII, XXII, XXIII, XXIV, XXV }. The refined architecture of the resulting system is depicted in Figure 3.10:

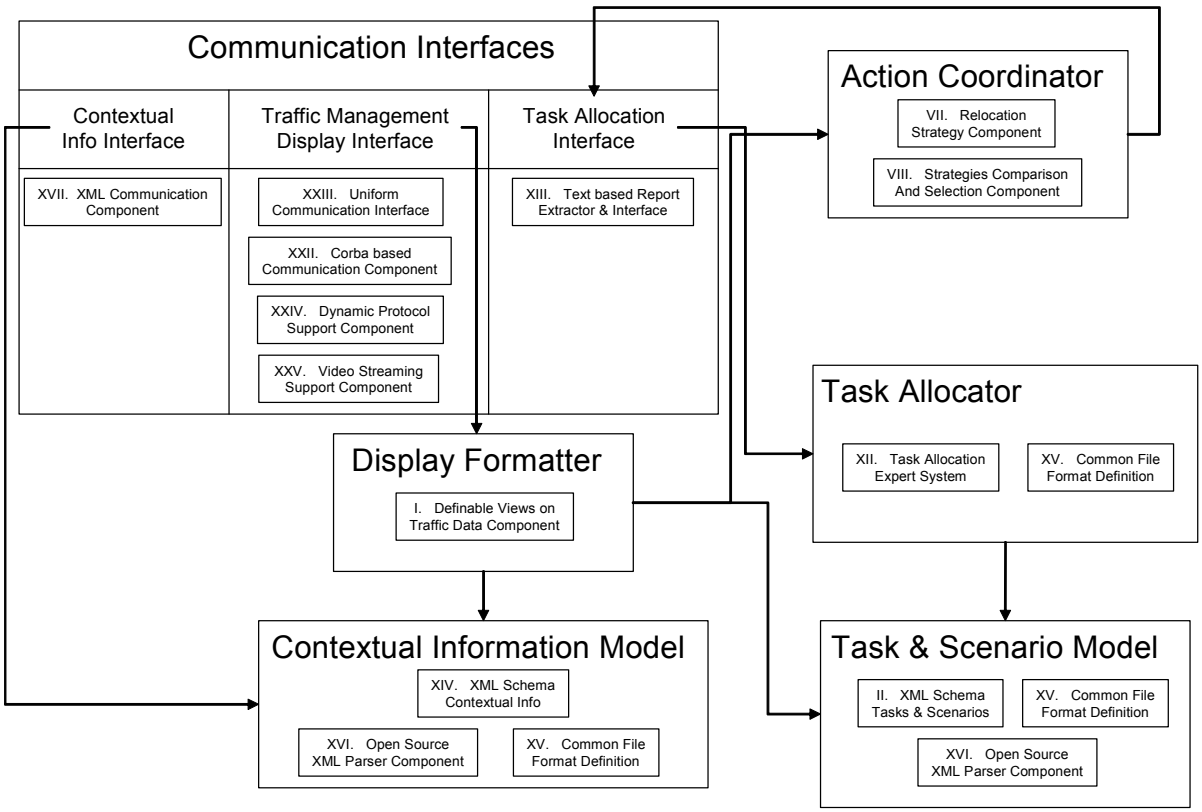


Figure 3.10 Relevance maximized TMS architecture

In this figure, the identified components from the optimization are placed in the abstract architecture. The refined architecture in this picture differs considerably from the refined architecture in Figure 3.5. Especially for requirement 4, multiple interpretations have been included, which increases the relevance of this system. The relevance of the system is 0.476 with the cost of implementing the components being 33.0 person-months. This means that by modeling alternative interpretations for the fuzzy requirements, it is possible to find a design for the Traffic Management System that exhibits a considerably higher relevance, while the costs for this design are actually lower than for the crisp system in section 3.4.

3.7 Related Work

3.7.1 Traceability of Intermediate Design Artifacts in Software Engineering

In our approach we define a tracing model specifically aimed at capturing relationships between intermediate design artifacts. Requirements tracing is a well-defined area and has resulted in numerous techniques for tracing software design processes. Each of these approaches is aimed at different uses, and is specifically suited to achieve its purpose. For instance, a tracing approach based on hypertext [Kaindl1993] is primarily aimed at easily browsing through documentation by use of hyperlinks. Other approaches are aimed at specifically linking elements together to determine coverage and balance of intermediate design steps, such as trace matrices [Davis1990] and matrix sequences [Brown1991]. Another use of trace models is to analyze the fulfilment of requirements based on the structure of the require-

ment trace. This is, for instance, done by assumption-based truth maintenance networks [Smithers1991] and constraint networks [Bowen1990]. While all these approaches have specific uses, it is not possible to apply these approaches to work with imperfect inputs and to optimize system designs. This is due to the specific attributes that are needed in the trace model, which are mostly only in part captured by the tracing models.

3.7.2 Decision Models of Software Processes

To address the complexity of developing software systems, many design methods have been proposed and expanded upon, such as Structural design [Yourdon1979] and Rational Unified Process [Jacobson1999]. Each of these methods emphasize different aspects and generally differ from each other with respect to the adopted models, such as functional models, data oriented models, etcetera. In addition to structuring the consecutive phases of software development, the methods propose a large set of explicit and implicit heuristics rules, which can differ per method. In [Tekinerdogan2002], based on their heuristics, architecture design methods are classified as artifact-driven, use-case-driven and domain-driven. In the artifact-driven approaches, software is designed from the perspective of the available software artifacts. For example, in the OMT method, a class is identified using the rule: "If an entity in the requirement specification is relevant then select it as a tentative class". In the use-case driven approaches, use cases are applied as the primary artifacts in designing software systems. For example, in the RUP, analysis packages, which are the primary means to decompose software, are identified with the rule: "Identify the analysis packages if use cases are required to support a specific business process". In the domain-driven approaches, the fundamental software components are extracted from the concepts of the domain model. An extensive number of software engineering environments have been proposed to support software engineering methods. Most environments provide model editing, consistency checking, version management and code generation facilities. There is a considerable amount of research on process modeling [Kaiser1994] [Finkelstein1994], as well as research in the field of assisting software designers with automated reasoning mechanisms. However, formalizing design heuristics and providing some sort of expert system support during the design process is not exploited well. This is because most approaches can not deal with imperfect information in the design process. In [Aksit2001], a design heuristics support approach based on fuzzy logic is proposed. However, this work does not address the same problem of imperfect information as defined in this chapter.

3.7.3 Imperfect Information in Design Processes

Over the years, there have been a number of approaches to support imperfection in design processes, but only a relatively small amount has focused on the area of software design. However, the existing approaches are mostly based on fuzzy logic and fuzzy set theory. In [Aksit2001a], partial applicability of design heuristics in the OMT development process is supported by a fuzzy logic based representation. The inconsistency is controlled and maintained by fuzzy reasoning techniques, until it can be resolved by new design inputs. In [Lee2003], imprecise functional requirements are captured used a fuzzy logic framework. Each intermediate design step, the applicability of the design can be assessed with respect to the requirements in a manner similar to proving an invariant over a piece of code. The resulting value then indicates to which degree the requirement holds.

Also, more generic models have been proposed for the support of design processes. An approach to model imprecision in design inputs is proposed in [Law1995]. This imprecision is captured using fuzzy set theory, and the imprecision is then used to explore the possible design alternatives and the evaluation of design alternatives. To describe the sequence of decisions, in [Liu2005] an extension to decision trees is proposed. In this decision tree the imprecise attitude

of the decision maker with respect to risks is considered using techniques from fuzzy logic, and combined with the decision optimization algorithms of probabilistic decision trees.

3.8 Discussion

Can imperfection in functional requirement specifications be avoided with more complex specification and analysis methods?

The approach proposed in this chapter addresses imperfection in functional requirement specifications by means of fuzzy requirements. An alternative approach could lie in the prevention of imperfection in functional requirement specifications, so that the imperfection will not manifest itself in the design inputs. In the related work already a number of approaches were identified, which facilitate the definition of requirement specifications in order to achieve this goal. However, imperfection in functional requirement specifications can not always be resolved in this way, for instance when the required information is not available. In this case the imperfection can not be prevented by additional deliberation. The failure of the waterfall-model, and the general acceptance of iterative and agile approaches is a clear illustration of this fact. Typically, only in specific or isolated cases, where the problem domain is very well understood, it is possible to come to such precise specifications. Since imperfection is an intrinsic property of software development, it is not sufficient to facilitate the requirement definition process.

Can imperfection in functional requirement specifications be avoided by more experienced designers?

Complementary to the discussion point in the previous section, increased insight and experience in the definition of functional requirements can be very useful, By training the architect in recognizing imperfection in design input, the imperfection can be removed and/or resolved before the design process is continued. However, as was already mentioned above, imperfection is an intrinsic part of the software development process. While using more experienced designers will increase the accuracy of these estimations, they will still be prone to imperfection due to the lack of information. Nonetheless it will be very useful for architects to identify imperfect information when it occurs, since this will make it possible to treat the imperfection more appropriately.

How do we define stakeholder interest values that accurately represent the current situation?

The fuzzy requirement concept, which has been presented in this chapter, is based on the definition of stakeholder interest values that describe the imperfection. To ensure that the results of the optimization with the Artifact Trace Model are relevant, the stakeholder interest values should accurately represent the imperfection in the requirement specification. However, there is no standard manner to determine these stakeholder interest values. Nonetheless, in our early experiments the definitions were quite natural for the users, since the notion fits the state of mind of the software designer fairly well. By capturing domain specific information and user experience with the approach, and using these definitions as a guideline, the use of fuzzy functional requirements can be facilitated. Also the support of tools can help greatly, since it facilitates experimentation with specific fuzzy sets and probability distributions.

Does the added effort of fuzzy functional requirements and the Artifact Trace Model scale up to industrial application?

With the inclusion of alternative interpretations using fuzzy requirements, the effort for designing software systems is increased. It is necessary to include more requirements in the design, and additionally the relationships between intermediate design artifacts must be administrated. Furthermore, the optimization based on the Artifact Trace Model in an industrial setting can become computationally cumbersome. By supporting the capturing of the design decisions in an automated tool, the additional effort is minimized. In this tool also the computational support for comparing different types of imperfect information is included, which minimizes the computational overhead for the software engineer. On a larger scale, the approach is intended to minimize the impact of imperfect information, which means that the added effort in the early stages of the development process is compensated by a lower need for complex adjustments in the software design.

3.9 Conclusions

In this chapter, imperfect information in functional requirements is identified as an important problem in the design of software systems. This can lead to the development of software systems that do not reflect the stakeholder's intentions, since requirements that contain imperfection can be misinterpreted by software engineers.

We have shown that imperfect functional requirements can be managed by capturing the ambiguous nature of the imperfection. To accomplish this, we capture the possible interpretations of the imperfect information in fuzzy sets, and treat these fuzzy requirements in the same way as traditional requirements. The membership degrees of the interpretations in the fuzzy requirements can be used to model particular interests of stakeholders, such as desirability or applicability. In addition, we have shown that the fuzzy requirements can be incorporated and traced in the Artifact Trace Model, and thereby benefit from the optimization capabilities of this model. The interpretations of fuzzy requirements can be traced in the same manner as perfect requirements, and the optimization can be used to determine, for instance, cost-minimized or relevance-maximized system architectures. This makes the design process more flexible and resilient to the occurrence of new insights alongside the development process. When these new insights correspond to interpretations included in the design, the system can be adjusted easily. In addition, the assessment and optimization capabilities of the Artifact Trace Model can be used to adjust the design of the software system and steer the incremental design steps of the development process. With this model, a new approach to specifying functional requirements has been defined. Instead of trying to specify the requirements in minute detail, the approach offers the possibility to capture the uncertainty that exists with the stakeholders. With the Artifact Trace Model the inclusion and consideration of such imperfection in the development process is facilitated and supported.

Our approach was demonstrated by applying the approach to the Traffic Management System example, where four of the initial requirements contain imperfect information. In the traditional evaluation method, one interpretation for each requirement was used. When it was evaluated using our approach, not considering imperfection resulted in a system design that was considerably more expensive and less adequate than the optimized systems that resulted from our approach. In chapter 6, we present tooling support for the Artifact Trace Model approach, which can be used to trace artifact refinement and to perform trade-off analysis between stakeholder interests and for instance implementation effort. In chapter 7 we explore the applicability and usability of the approach by applying the tooling in a pilot study. The results of this pilot study are used to evaluate the approach proposed in this chapter.

CHAPTER

4

"Any area is open to my speculation if it does what you've hired me to do," Hawat said. "I am a Mentat. You do not withhold information or computation lines from a Mentat."

- From Frank Herbert's Dune [Herbert2005]

SPECIFICATION AND EVALUATION OF IMPERFECT QUALITY REQUIREMENTS AND ESTIMATIONS

4.1 Introduction

In the previous chapter, we have defined an evaluation model that facilitates the modeling of and reasoning with imperfect information in functional requirement specifications. However, the occurrence of imperfect information is not limited to functional requirements. The specification of precise quality requirements is equally difficult, since these requirements are also defined in the early stages of the software development process. Since both stakeholders and software engineers have a partial view of the completed system, the correct assessment of the desired quality attributes is hampered. In a similar manner, this partial view complicates the quality based assessment of alternative system designs for software engineers. As a result, design decisions need to be reconsidered at later stages of the development process, when new insights are attained. Consequently, delivering software systems that fulfil all quality requirements of the stakeholders is very difficult, if not at all impossible. To address these problems, we define a method for tracing design decisions with support for imperfect quality requirements and estimations. To model and compare imperfect requirements with imperfect estimations, we present fuzzy and stochastic techniques. The approach is illustrated by an industrial example based on a storm surge barrier system.

4.2 Quality-based Design Alternative Selection

4.2.1 Problem Statement

In many practical software projects, a number of design decisions are taken in a sequential manner. Typically, for each *design issue* several candidate solutions or *design alternatives* are considered. These design alternatives are evaluated based on the quality expectations, in order to establish an ordering among them. The design alternative that offers the best quality is then selected to fulfil the design issue in the design of the system.

This activity of evaluating and selecting design alternatives based on their expected quality attributes is hindered by the presence of imperfect information. The evaluation of design alternatives is typically performed in the early stages of the design process. At this point, the software engineer only has a partial, abstract view of what the resulting system will be. As a result of this abstract view, the quality of a system that results from selecting a candidate solution is largely unclear. The quality estimations that are used in the evaluations can be considered to be imperfect. Nonetheless, these estimations are expressed using traditional numerical expressions, which do not capture the imperfection appropriately. Further, these numerical expressions are used to establish the ordering among the design alternatives, which makes the ordering vulnerable to the unmodeled imperfection in the quality estimations. The selection of design alternatives is hindered further, since the system is supposed to fulfil quality requirements. As with functional requirements, the definition of these quality requirements can contain imperfect information. Since quality requirements are defined early in the development process, it is difficult to provide the restrictions with sufficient precision. As a result, the quality requirements are likely to be changed or refined over time.

The presence of imperfect information in quality estimations as well as quality requirements directly influences the uniform and systematic evaluation of candidate solutions based on their quality attributes. However, since this imperfection is not captured and considered during the evaluation activity, it is likely that the established ordering of the candidate solutions does not reflect the ordering based on the actual resulting quality. Nonetheless, the software design process is continued after each decision, assuming that all these orderings were correct. Therefore, imperfect information can cascade through a sequence of design decisions, before the imperfection can be resolved by new insights. The arrival of new insights during the software development process can invalidate (parts of) the design of the software system. In most software design processes, the correction of designs is facilitated by means of iterative design cycles. At the moment the current design is no longer satisfactory, the design process will perform an iterative correction on the design. However, in chapter 1 we have established that for successful iterative design four requirements must be fulfilled: the All-Always-Requirement, the All-at-Once-Requirement, the All-Isolation-Requirement and the All-Infinite-Requirement. As a result of these requirements, iterative design is not capable of resolving the consequences of imperfect information effectively. In addition, since the nature of imperfect information as well as the impact of using this information in the evaluation process is not well understood, adjusting the design with iterative steps is not done in a systematic manner, which makes incremental design even less effective.

In this chapter, we propose a tracing model, with which it is possible to describe the design issues that have been resolved, the order in which they have been addressed and the design alternatives that have been considered. Our model is used to systematically explore the designs that are available based on the evaluations of individual design solutions. Additionally, our approach includes a configurable algorithm, to reflect different managerial interest for the correction in the design process, such as minimization of costs, or design for the highest possible quality. This design algorithm guarantees that the space of alternatives is explored in a systematic manner. To address the influence of imperfect information on the exploration algorithm, we extend quality requirements and quality estimations with fuzzy and probabilistic tech-

niques. These techniques enable the method to evaluate design alternatives while considering the imperfect nature of both requirements and estimations. The novelty of the approach lies in the possibility to model and analyze information in quality requirements an expectations that does not contain the level of detail that is desirable. The manner of specification is extended with models that can describe this imperfection accurately. The design tree model ensures that the decision making process based on this information considers the imperfection and its consequential risks accordingly.

4.2.2 The Design Tree Model

In the previous section we have identified that during software development several design issues are identified and resolved, often in a sequential manner. From this perspective, the design of software can be seen as a process of steps, in which customer requirements are refined into a software system that incorporates these requirements. In each step of the process one of the remaining design issues is resolved by identifying and selecting the most suitable solution for this issue. Based on this view of the software development process, iteration is achieved by rolling back to one of the previous steps and reconsidering the solutions that were selected from this step forward. In our approach we describe a design step as a transition between two design states. We define a *design state* to consist of the following elements:

- *Unresolved Issues Set*
- *Principle Design Issue*
- *Resolved Issues Set*
- *Quality Expectation*

The *Unresolved Issues Set* contains the remaining work for the software engineer by means of a set of unresolved design issues. The *principle design issue* is the design issue that will be resolved first before other design issues can be addressed. The principle design issue is selected from the unresolved issues set by the software engineer at every design state. The *Resolved Issues Set* contains the issues that have been resolved, and the solution that has been selected for each resolved issue. The *Quality Expectation* is a numeric expression that describes the maximum quality that the software engineers expects to be achievable from this design state. We elaborate on quality based evaluation of design states in section 4.2.3. Based on this definition for design states, a design step schematically can be described as follows:

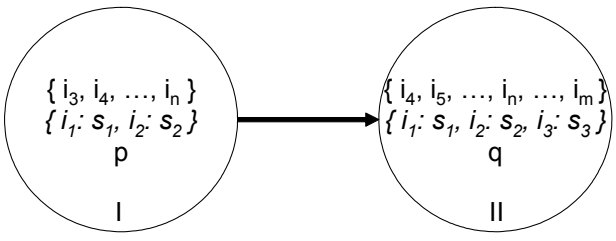


Figure 4.1 Design Step

In Figure 4.1, a design step from design state I to design state II is described. In state I, two design issues i_1 and i_2 have been resolved with solutions s_1 and s_2 respectively. The software engineer has identified i_3 as the principle design issue for state I. Furthermore, the expected quality that can be achieved from this state is indicated with p. During the step from state I to step II the principle design issue is resolved with the solution s_3. Therefore, in state II the set

with resolved issues contains i_3 with a solution s_3. The quality expectation is updated to a value q, since the inclusion of s_3 can lead to new insights on the quality of the resulting system. The new issues that result from the selection of s_3 are added to the unresolved issues list.

Obviously, it is, in general, possible to resolve a design issue in more than one way. For most design issues solutions exist that are functionally equivalent, but exhibit different quality behavior. The choice between such design alternatives is described by adding multiple resulting design states to the current design state. An example design decision with design alternatives is depicted in Figure 4.2.

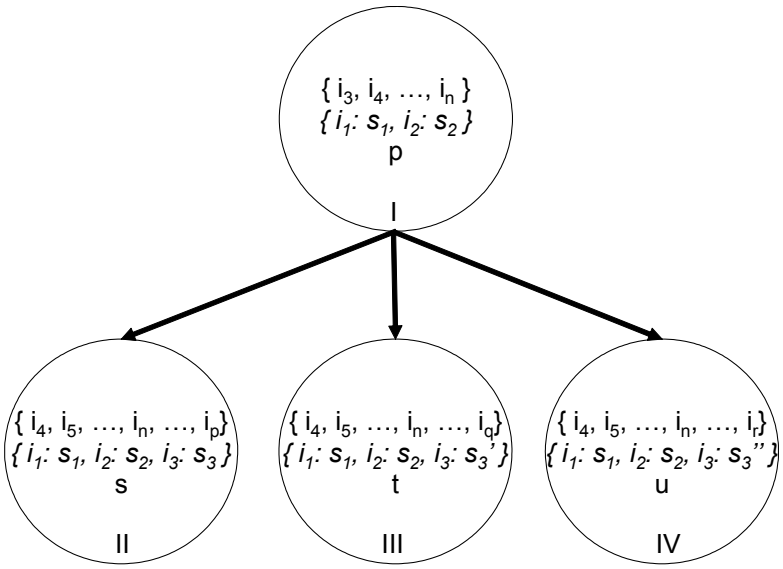


Figure 4.2 Design Step with Alternative Solutions

In this picture issue i_3 can be resolved with three solutions, s_3, s_3' and s_3''. This means that the design states II, III and IV describe alternative design states from which the design process can be continued. In each of these states i_3 has been resolved and i_4 is the new principle design issue. The value for the expected quality can differ per state, since each state leads to a different system design. The selection of a design alternative can be based on these values, for instance by choosing the value that indicates the highest expected value. When the entire sequence of design decisions, that are taken in the course of a software development process, are captured using design states as described above, a *design tree* is created. A design tree is a tree structure that contains all the design decisions that have been made and the alternatives that have been considered. In the design tree each node represents a design state in the software development process. A sample design tree is depicted in Figure 4.3.

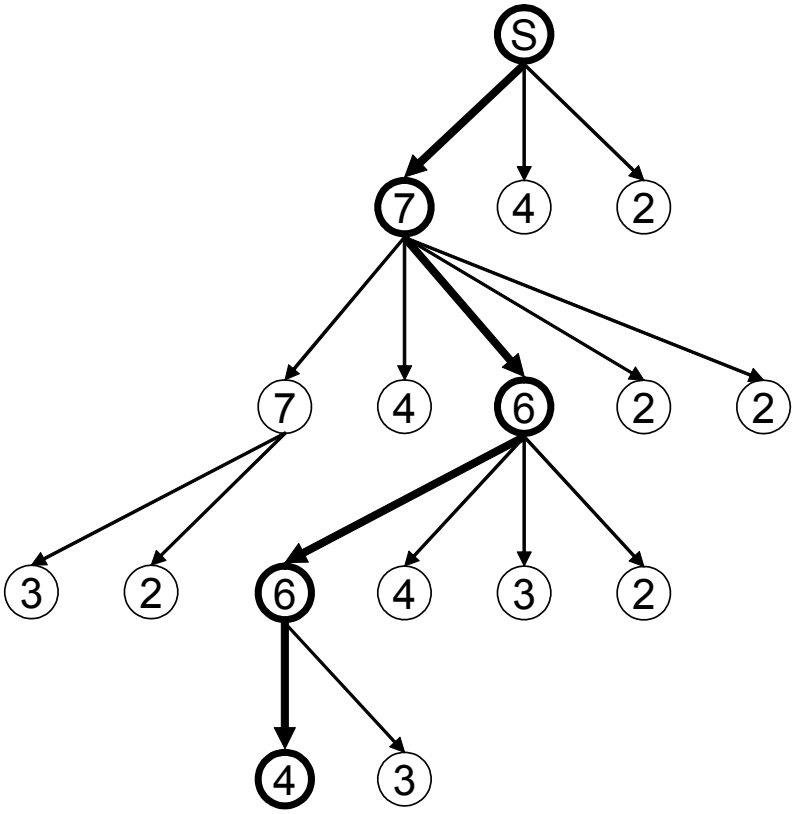


Figure 4.3 Design Tree

This design tree represents a sequence of four design decisions. In the first design state *S* four design issues are identified. For the principle design issue of this state, three different design alternatives have been identified. Each of these alternatives has received an evaluation with respect to the expected quality of the resulting system, indicated by the numbers attached to the node (a higher number means higher expected quality). The design process is continued by choosing the alternative with the highest quality expectation. This is repeated until all design issues have been resolved. Note that in each state the alternative was chosen that offers the highest expected quality. From the picture, it can be seen that a design tree describes the decisions, the order in which they have been resolved and the contemplated and selected alternatives. In the case that a design for a software system needs to be adjusted, the design states in the design tree can be revisited as potential points to roll back to. In the picture it can be seen that the deepest node with quality 7 has two child nodes, one with quality 3 and one with 2. This describes the situation where the estimation was considerably larger than the quality of the new design states. At this point, it is better to continue with the node with quality 6 and expand from there. In section 4.2.4, we elaborate on the strategies for selecting design states with respect to particular design interests.

4.2.3 Quality-based Evaluation of Design Alternatives

In Figure 4.3 each design state has a number that represents the expected quality of the resulting system. This number is a compound description on how well the final system is expected to perform with respect to the quality requirements. To enable a comparison mechanism for the selection of design alternatives, the quality evaluation of design alternatives must be performed in a uniform manner. To facilitate the uniform evaluation of design alternatives, we define a standard evaluation method in this section. Since design alternatives generally are

functionally equivalent, the selection of one alternative over the other can only be based on differences in their quality attributes. Quality attributes are non-functional aspects of software systems, such as performance or reliability. This definition assumes that the quality attributes of interest are expressible in numbers. We define quality requirements in a fashion very similar to a traditional non-functional requirement, however for the uniform evaluation of the expected system quality a numeric description is required. Therefore, we define quality requirement to be a numeric boundary on a quality attribute. Quality estimations can now also be defined in terms of quality attributes. A quality estimation is defined to a numeric description of the expected value of a quality attribute. The quality estimations that are made for a design alternative should describe the expected quality of the *overall system* if this system would include this design alternative. Additionally, for each quality attribute *x* we define $Quality_x$ to be the boolean evaluation of the estimation $Estimation_x$ with respect to its quality requirement $Requirement_x$.

$$Q_x = Requirement_x \geq Estimation_x$$

Obviously, this definition is used for quality requirements that define an upperbound restriction. For lowerbound requirements the comparison operator is reversed. For the evaluation of *j* design alternatives for a particular design issue with respect to *n* quality requirements, we introduce the following evaluation table:

Table 4.1 Design Alternatives Evaluation Table

Design Decision	$Estimation_1$	$Estimation_2$	...	$Estimation_n$	$Quality_1$	$Quality_2$	...	$Quality_n$	Overall Quality
Option 1									
Option 2									
...									
Option j									

Table 4.1 describes a standard table for the evaluation of the design alternatives for a particular design issue. Each row describes the evaluation of a single design alternative, or design state, based on its respective quality estimations. In the first columns of the table $Estimation_x$ contains the estimated value for the quality attribute that is constrained by $Requirement_x$. For example, if $Requirement_1$ defines a restriction on the response time of the system, the column indicated by $Estimation_1$ contains the estimated response times for the design alternatives. The second set of columns contains the boolean evaluation of comparing the estimations to the requirements, indicated by $Quality_1$, $Quality_2$, ..., $Quality_n$. In the example above, $Quality_1$ would contain a "true" for option 1, if the response time of this alternative is estimated to be faster than the required value described in $Requirement_1$.

The final column in Table 4.1, labelled *Overall Quality* contains the numbers that describe the overall evaluation of the respective design alternatives. These numbers are statements based on the evaluation results of individual quality attributes ($Quality_1$, $Quality_2$, ..., $Quality_n$). There are many different ways in which the attribute evaluations can be combined, depending on the particular interest of the stakeholder. Straightforward examples for the overall quality computation include the multiplication of the attribute values, which corresponds to the logical AND-operator, or the sum of the attribute values. It is also possible to define more complex functions that reflect stakeholder preference for quality attributes by means of weighted averages or penalty functions. In current software engineering practices, it is quite usual to make estimations

and evaluations as it is carried out in this section [Clements2004] [Kazman1998], although this is not always defined explicitly. The evaluation approach presented in this section is extended in section 4.3 by defining imperfection support for quality requirements, quality estimations and the comparison operators.

4.2.4 Configurable Design Strategies for Design Trees

With the design tree model, software design can be defined as a search problem within a search space, which is comprised of all design states that can be considered if all possible design alternatives are known. This search space is the tree structure that is comprised of all possible alternative system designs for the identified design issues, and is called the *principle design tree*. The activity of resolving design issues as described in the previous section then becomes a partial exploration of the principle design tree as depicted in Figure 4.4.

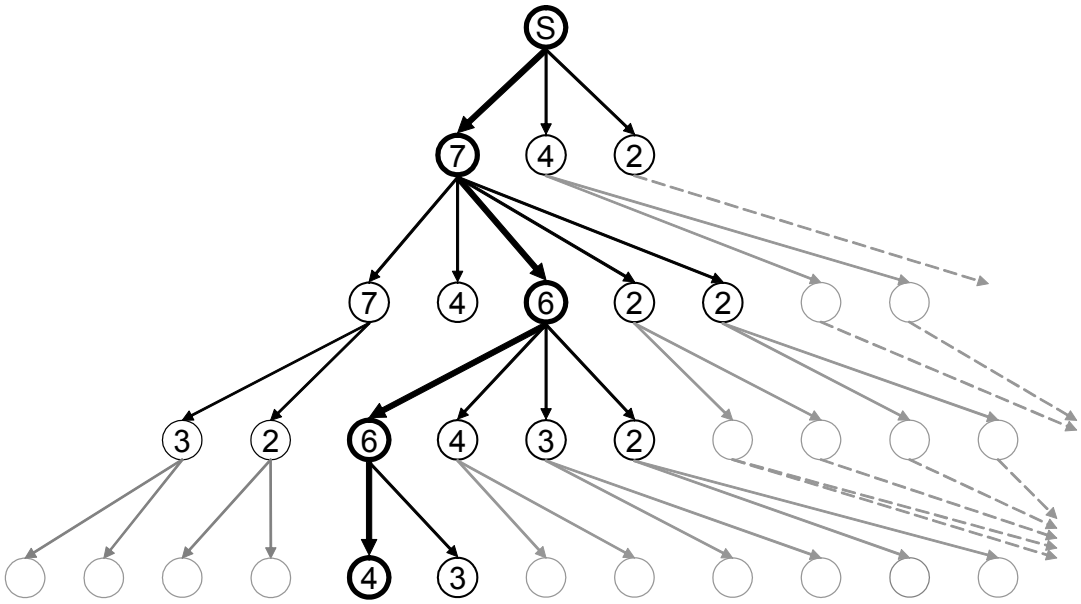


Figure 4.4 A Partial Exploration of the Principle Design Tree

In this picture, the design tree of Figure 4.3 is depicted as partial exploration of the principle design tree. The design states of the principle design tree that have not been identified as alternative solutions, have been greyed out. If all the design states of the principle design tree could be identified, it would be possible to find the best possible system design. Typically, the principle design tree is too big to explore completely, therefore the development process must aim at expanding the design tree only until a design is found of acceptable quality.

The decision support capabilities of the design tree model lie in a search algorithm that determines in a systematic manner from which design state the software engineer should continue. This ordering of the potential design states is achieved by sorting all the leaf nodes of the current design tree. The selection of the design state from which to continue is therefore determined by the way the nodes are ordered, or the so called *design strategy*. Note that it is possible to sort nodes based on multiple criteria, such as the quality expectation or the depth in the tree. These ordering criteria represent different design strategies, such as quality or cost optimization. The preference of one strategy over the others is based on managerial motives such as minimization of costs, or time to market. We define the following design algorithm, which enables the construction of a design tree based on configurable design strategies

```
Design
{
  List L = { Root Node };
  Node N = First element of L;
  While (N is not a completed design)
  {
    Remove N from L;
    Add Children of N to L;
    Sort(L);
    N = First element of L;
  }
  Return N;
}
```

In the first step of this algorithm, the root node of the design tree (which is the only node) is stored in a list *L*. Additionally, the current design state *N* is set to be the root node of the design tree. The while-loop of the search algorithm is entered when the unresolved issues set of N is not empty. Since N is the design state for which a design issue is resolved, it is removed from L in the first step of the loop. In the next statement all children of N are added to L, which corresponds to the identification of design alternatives for the principle design issue in N. After this, all the design states in L are ranked according to the design strategy by sorting the list. Finally, the first element of L is selected as the new current design state N. In case N is a design state without remaining design issues the design process is completed, otherwise the while-loop is repeated. Note that L contains all leaf nodes of the design tree, which means that design states that were not selected will be reconsidered in subsequent design steps.

In this algorithm the function *Sort* rearranges the list L such that it becomes an ordered list. This means that design strategies are implemented in the Sort-function and only differ in the comparison criterion for two nodes. By choosing a different sorting function we can configure the design strategy we want to use. Below we describe three different design strategies for decision support in the design process. Note that these design strategies are variants of the branch-and-bound searching algorithm, and are in particular variants of the well-known *A*-search algorithm*, which is for instance described in [Russel1995]. As a result, it is vital for the quality estimations to be *optimistic estimations*. This means that each design state should always expect a better quality value than the final system will have. When this restriction is not adhered to, the A*-search algorithm, and with that the optimization approach, can not guarantee the optimality of the final result.

Quality-based Design Strategy

The first strategy, aimed at finding the optimal design, uses a comparison based on only the quality evaluations of the individual design states. The nodes are ordered based on their individual quality estimations, with the node with the highest estimation ordered in front. This strategy guarantees to find the best possible design of all designs that can be found in the design tree. However, due to the need to explore the entire principle design tree, this strategy will take a very long time to result in the final system and as a result the costs of this strategy can become too high. When we examine the final decision in Figure 4.3, we are at the deepest node with quality evaluation 6, and two alternatives for the current principle design issue. The first alternative is evaluated with quality 3 and the second with quality 2. The application of this strategy means we continue with the node with the highest quality evaluation, which is the node with expected quality 5.

Time-based Design Strategy

The second strategy therefore is directed at minimizing the time needed to find a design, *any* design. The depth of a node in the design tree indicates how many design issues have been resolved, which means that a node that is deeper in the tree is closer to a completed design. Therefore the fast strategy always chooses the lowest leaf node in the tree, and in case two or more nodes are at the same depth, the node with the highest quality estimation is taken. To implement this strategy, in addition to the quality estimation for each node, also the depth of the node in the tree is needed. Therefore the value of a node in the design tree is represented by a 2-tuple of type:

(Depth, Quality Value)

The first element of the tuple is the depth in the tree and the second element is the quality evaluation of the node. The comparison of two nodes can now be performed by the standard comparison operator for tuples:

$$(n_1, m_1) > (n_2, m_2) \Leftrightarrow (n_1 > n_2) \vee ((n_1 = n_2) \wedge (m_1 > m_2))$$

Note that this strategy aims to find a design as soon as possible and disregards any quality constraints, which means there is no guarantee that the system will satisfy the quality requirements. When we use this strategy to determine the best node in the final decision in Figure 4.3, we choose the node the is deepest in the tree, which means one of the children of the current node. Since both nodes have an equal depth in the tree, we choose the node with the highest quality expectation, which is the node with expected quality 3.

Balanced Design Strategy

In order to offer a balanced advice on the selection of design alternatives, the third design strategy offers a trade-off between the expected quality of the system and the amount of work that is needed to come to a completed design. To achieve this, the balanced strategy selects the deepest node for which the quality requirements are fulfilled by the quality estimations. We extend the 2-tuple of the previous strategy with a Boolean that indicates whether the node satisfies the defined quality constraints. The value of a node now becomes a 3-tuple of type:

(Boolean, Depth, Quality Value)

The first element is a boolean that reflects the truth of the statement *"The quality estimations of the system satisfy the quality constraints"*. The second element is the depth of the node from the root of the tree. The third element is the actual quality estimation of the node. Again, using the standard comparison operator for booleans, the nodes can be sorted based on these values. The final design that is found by this strategy satisfies the quality constraint (if such a design exists), but it needs not to be the design with the highest possible quality. This strategy will find an acceptable system rather than the best system, but the amount of reiterations will be reduced compared to the quality-based design strategy. In addition the strategy assures that if a system exists that adheres to the defined constraints that it will be found. The selection of the best design state for the last design decision in Figure 4.3 now depends on the quality require-

ment for the system. If the quality should be at least 4, the deepest node in the tree with quality 4 would be selected. If the quality should be at least 5, this strategy would behave the same as the first strategy and select the node with quality 5.

Note that continuing the design from a previous node corresponds to taking an incremental design step, where the system is adjusted. Therefore, the adjustable design strategies in the design tree approach facilitate the systematic iterative design of the software system.

4.2.5 Example Case: Remote Water Sensor

We demonstrate the design tree approach by means of an example. Consider a storm surge barrier designed to protect a moderately populated urban area. The choice of this example is inspired from [Tretmans2001]. The barrier has to be closed only in case of absolute necessity; otherwise the cargo transport can be hampered unnecessarily. However, leaving the barrier open during storm situations can result in immediate danger for the population. Since the decision to close the barrier is a complicated task, it has been decided to incorporate a computer-controlled system for this purpose. The control system should make a decision every 10 minutes, based on numerous inputs such as weather forecasts, changes in the water level, tides, etc.

In order to work out this example case, we need to decide on a software design process. First, we will present the non-functional requirements. Second, we will describe the design process. Finally, we will further restrict our scope by making some initial design decisions. We would like to point out that the techniques proposed in this example are general and the suggested process is introduced for illustration purposes only. The functional requirements are summarized as follows: The Remote Water-level Sensor (RWS) system measures the water level of the river and reports it periodically to the host computer, which is placed at some other geographic location. The host computer, in turn, sends control requests to the RWS. The system architects are requested to analyze the RWS with respect to performance, reliability and cost. The following non-functional requirements are provided by the stakeholders:

PR1: Client must on average receive a water-level reading within 500 milliseconds after sending a control request.

PR2: Client must at the latest receive a water-level reading within 650 milliseconds after sending a control request.

PR3: When a failure occurs in the measuring part, the host system must be able to continue operating for 10 seconds.

PR4: The cost of this system must not exceed 225K euros.

The stakeholders have required that all these requirements must be fulfilled, otherwise the system is not acceptable. The software engineers have established the following two design issues for the main design of the software system:

a) The number of sensors and the scheduling of the server has to be determined.

b) The architecture style has to be selected. This step can be further specialized as the selection of the sensor, server and connection topology.

We assume that the system architects initially take the following design decisions. The RWS is embedded into a system architecture based on the client-server idiom. The RWS functionality is encapsulated in a server that serves some number of clients. The RWS server hardware

includes an analog-to-digital converter (ADC) that can read and convert a water-level for one sensor at a time. Requests for water-level readings are queued and fed, one at a time, to the ADC. The ADC measures the water-level of each sensor at the frequency specified by its most recently received control request.

4.2.6 Design Decisions for the Remote Water Sensor

Alternatives of the sensor server architecture

According to the process in the previous section, first the internal architecture of the server must be selected. Assume that the architecture contains three kinds of components: water-level tasks (independently scheduled units of execution), that are scheduled to run with some period; a shared communication facility task (Comm), that accepts messages from the water level tasks and sends them to a specified client; and the ADC task, which accepts requests from the water level tasks, interfaces with the physical sensors to determine their water heights, and passes the result back to the requesting water level task. The alternatives for the server architecture lie in the implementation of the ADC and the amount of sensors.

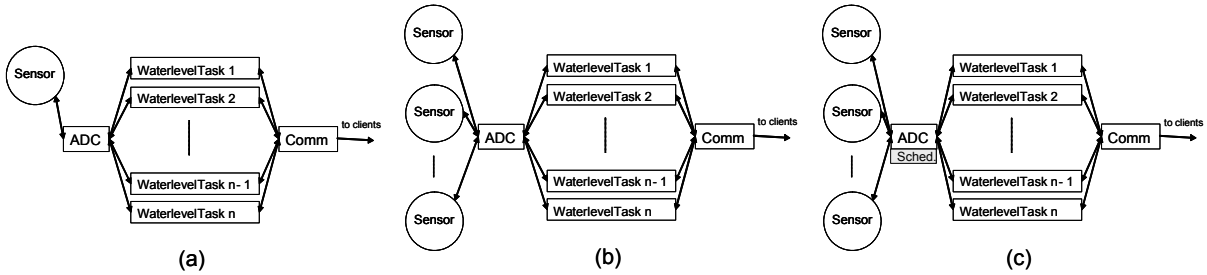


Figure 4.5 Server Architecture Alternatives: (a) Single Sensor, (b) Multiple Sensors, (c) Multiple Sensors with Scheduler

Figure 4.5, shows three alternatives for the server architecture. The alternative *a* is based on a single sensor. In this alternative, only one measurement can be performed at a given point in time. During measurement, all requests that arrive will have to wait according to a first come first served principle. We assume that in this option no priority mechanism or scheduling is implemented. In alternative *b* the server is connected to multiple sensors and the waterlevel tasks are stacked on to the sensors until all sensors are occupied. Once this is completed, each task will be added to the set of sensors on a first come first served principle. Again here, we assume that no priority mechanism or scheduling is implemented. In architectural option *c*, we assume that the server is connected to multiple sensors. In addition, this architecture also contains an intelligent scheduling mechanism based on priority levels of individual tasks, such that the most important measurement tasks can be performed as soon as possible. We will now examine these three design alternatives in more detail. The results of the quality estimations and evaluations are given in Table 4.2. Note that the estimations are made for the expected *overall* system quality and in an *optimistic* fashion, to ensure the optimal results of the design strategies.

Table 4.2 Design Decision 1

	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1									
Opt. 1.1	400	400	∞	180	1	1	1	1	1
Opt. 1.2	350	350	∞	190	1	1	1	1	1
Opt. 1.3	300	300	∞	230	1	1	1	0	0

In this table the design alternatives for the first design decision are evaluated according the approach defined in Section 4.2.3. For each of the design alternatives an estimation is made for the average performance, the maximum performance, the reliability and the cost. The values for $Quality_1$, $Quality_2$, $Quality_3$ and $Quality_4$ indicate whether or not the estimations satisfy their respective requirement given in Section 4.2.5. For example, $Quality_1$ for option 1.1 has the value "1", since the estimated average performance is 400 milliseconds, which satisfies the requirement of a response time within 500 milliseconds. Finally the column Overall Quality indicates the amount to which the option in its entirety satisfies the quality requirements. Since the stakeholders have demanded that all requirements are fulfilled, the overall value is computed by multiplying $Quality_1$, $Quality_2$, $Quality_3$ and $Quality_4$. With the multiplication, any requirement that is not fulfilled will cause the overall result to be 0 as well.

In the table, option 1 has the worst performance estimation, since this architecture only has one sensor, which is used for all measurements. Option 2 has a better performance figure, since the workload can be divided amongst multiple sensors. Option 3 can even optimize this with a dedicated scheduler. However, each of these options becomes increasingly more expensive due to these extra facilities. The reliability for each option is estimated to be optimal, since the communication architecture is not influenced by the choices that are made for the server architecture. As a result, for this quality attribute the estimation becomes an irrelevant value. From the table it can be seen that systems that include option 1.1 as well as option 1.2 are expected to fulfil the quality requirements. The intelligent scheduler is not a feasible choice, due to the cost of this alternative. Based on these quality evaluations, no distinctions can be made between the single server and multiple server architecture. At this point we choose option 1.2, but if this option turns out to be unacceptable at a later stage, it is still possible to backtrack to the single server architecture.

Alternatives of the Communication Architecture

The second design issue is to decide on the communication architecture of the sensor server and the client systems. This decision in particular focuses on the connection topology, which means that the quality evaluations will now explicitly consider the reliability of the system.

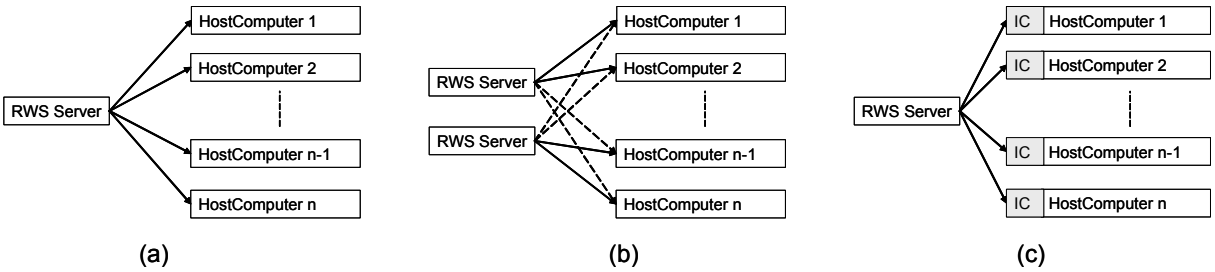


Figure 4.6 Communication Architectures: (a) Single Server, (b) Redundant Server, (c) Intelligent Caching

The first option for the communication architecture is indicated with *a* in Figure 4.6. This is a simple and inexpensive client-server architecture, with a single server (RWS Server) and multiple clients. Option *b* differs from the first option in that it adds a second server to the system architecture. These servers interact with clients as a "primary" server (indicated by the solid lines between servers and clients) or as a "backup" server (indicated by the dashed lines). Every client will automatically switch to their specified backup if they detect that the main server is down (because it has failed to send requests for a prescribed period of time). Option *c* extends option 1 by a "wrapper" that intercedes between the client and the server. This wrapper is an "intelligent cache", shown as *IC* in the figure. The cache intercepts periodic water level updates from the server to the client, builds a history of these updates, and then passes each update to the client. When the server is interrupted, the cache synthesizes updates for the client. The cache is considered to be intelligent because the updates it provides take advantage of historical water level trends to extrapolate plausible values into the future. This intelligence may be nothing more than linear extrapolation, it can be a sophisticated model that analyzes changes in temperature trends, or takes advantage of domain-specific knowledge on how water levels rise and fall. Obviously, the synthesized updates of the cache will become less meaningful over time. In Table 4.3 the evaluations of the design alternatives of the second design decision can be found, after choosing the second option at the first decision.

Table 4.3 Design Decision 2

	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 2									
Opt. 2.1	400	∞	0	190	1	0	0	1	0
Opt. 2.2	400	650	∞	200	1	1	1	1	1
Opt. 2.3	450	450	13	205	1	1	1	1	1

The average response time of the first two options in Table 4.3 is identical, since they both use the same server (but the second option has a redundant server that increases the reliability). The third option is obviously slower, since the intelligent caching needs to be updated. The maximum response time is indefinitely long for the first option since in case the server fails, there will be no reply. For the second option, the system will wait for a time-out of the first server before the second server sends the measurement. The third option has a maximum response time identical to the average, since the cache can provide "measurements" any time the server fails. The reliability for the first option is 0, since in case of a server failing, the sys-

tem is not able to continue running. For the second option the reliability is infinite, since in case a server fails, the system can continue operating normally using the backup server. For the third option, the reliability depends on the time the intelligent cache is able to provide sensible extrapolated values. Finally the cost for the multiple servers and intelligent caching is estimated to be higher than the cost for the single server solution. In this table it can be seen that the redundant server topology and the intelligent cache topology satisfy the quality requirements. Here we choose to continue with the intelligent cache alternative. As a result of this choice, the software engineers have to decide on which extrapolation algorithm for the intelligent cache to use. Therefore, the choice for this algorithm is added as a new design issue.

Alternatives of the Intelligent Cache

The final design decision to be made is with respect to the type of intelligent cache that will be used. This means that this issue will only arise when the intelligent cache option is selected in the second design decision. In this example three different cache implementations are considered: *Linear Extrapolation, Trend Extrapolation* and *Domain Analysis Extrapolation*. In linear extrapolation, we assume that only the values that have occurred recently from the sensors are considered. In this case, the cache does not need to keep track of a large number of measurement values. However, a linear extrapolation cannot be used over extended periods of time, since it does not keep track of the periodical behavior of rivers, for instance caused by rainfall or temperature changes. The trend extrapolation cache analyses the trends that have occurred in the available measurements, and tries to extrapolate multiple values according to this trend. For this type of extrapolation a larger set of values needs to be cached in order to make a reliable trend analysis (the actual amount of data depends on the kind of trend analysis). In addition to the amount of data required, the computational complexity also increases, since the trend analysis must be performed as well as the extrapolation. The domain analysis cache includes specific knowledge on how water levels change over time. This can for instance be knowledge on seasonal swings in water levels caused by precipitation or temperature levels. Together with a trend analysis based on recent data from the sensors, this domain knowledge can be used to perform an informed extrapolation. This should result in the possibility to provide credible extrapolations for a longer period of time. The three alternatives have been evaluated and the results of the quality estimations and evaluation, are given in Table 4.4.

Table 4.4 Design Decision 3

	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 3									
Opt. 3.1	510	510	9.5	205	0	1	1	1	0
Opt. 3.2	500	500	10	225	1	1	1	1	1
Opt. 3.3	850	850	12	300	0	0	1	0	0

The performance for the first option is estimated at 510, which is slightly higher than the second option. This is due to the fact that a linear extrapolation always needs to consider the newest value that has been measured to determine the linearity. The trend extrapolation does not need to do this. The algorithm of the second option always needs to consider a complex mathematical model of the environment variables, which makes the performance much slower. The reliability for the linear extrapolation is somewhat lower than the trend extrapolation, since it has a simpler means of extrapolation of sensor readings. The third option is obviously superior in this field. Finally the cost of each option increases as the complexity of the extrapolation

algorithm increases. In the Overall Quality column it can be seen that only one system fulfills the quality requirements.

4.2.7 Design Tree of the Design Decisions for the Remote Water Sensor

For the design of the Remote Water Sensor we now have addressed three design issues, for each of which three possible alternatives have been considered. The alternatives have been evaluated with respect to their expected quality, and based on this evaluation for each design issue a solution was selected. In Figure 4.7 a design tree is depicted, that represents the three design decisions that have been taken for the Storm Surge Barrier, as well as their overall quality evaluation.

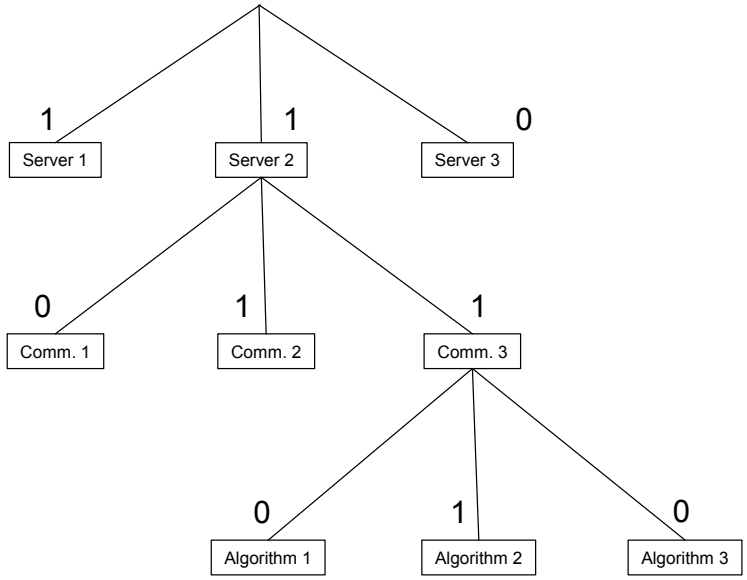


Figure 4.7 Design Tree of Remote Water Sensor

In this design tree, the overall quality evaluations of the individual design alternatives are included, as well as the choices that have been made. Note that the design strategies in this case can not sort a number of nodes distinctively, since the function we use to determine the overall quality can not sufficiently distinguish between alternatives that satisfy all quality requirements.

4.3 A Model for Imperfect Requirements and Estimations

4.3.1 Crisp Specifications of Quality Requirements and Estimations

In the RWS example, the quality requirements provide very precise boundaries for the acceptability of the quality attributes. For instance, the cost requirement expresses that the maximum cost must not exceed 225.000 euros. While the precision of this specification is very useful with respect to evaluation of design alternatives, as described in section 4.2.3, it generally is very difficult for stakeholders and software engineers to achieve. A stringent restriction does not allow system designs that exceed this limit, even when it is only marginally larger. For example, a system with a cost of 225.005 euros would be rejected based on this requirement specification. Typically, this is not the way in which the requirement specification was intended. The numeric restrictions in quality requirements generally are indications of desired behavior, and should not be treated as non-negotiable boundaries during the development pro-

cess. Rather, these restrictions should be seen as indicative boundaries, that can be exceeded within a certain tolerance range. However, while software design processes acknowledge the difficulty of defining accurate requirement specifications, they do not provide means to capture this imperfection and include it in the evaluation of design alternatives. Instead of modeling the imperfect information appropriately, a crisp specification is defined, even when the particular choice can not completely be justified.

In a similar manner quality estimations suffer from the use of crisp numeric expressions to indicate the expected quality. Quality estimations are needed during the development process in the case that it is not possible to measure or to determine the quality behavior of the completed system. In particular during the early stages of software design, software engineers need to make design decisions based on quality expectations, since at this time point only an abstract view of the system design is available. But while estimations are an imperfect description of the quality the system will have upon completion, they are described using crisp numeric expressions that do not capture the imperfect character of the estimations. For example, consider a cost estimation of *approximately 225.000 euros*. If this estimation is modeled by using only the numeric expression 225.000, the expected quality would fulfil the cost requirement of the Remote Water Sensor example. However, the term “approximately“ indicates that the actual cost can still differ from this estimation, which makes it possible that the costs of the completed system will exceed the restriction.

While the evaluation of the RWS example has led to two alternatives that satisfy the quality requirements, the manner in which decision is reached can invalidate the results. By defining and treating the estimations in the same manner as an actual measurement on the finished system, the inherent imperfection of estimations can invalidate many design decisions at later stages. Similarly, the boundaries set by the quality requirements can be deceiving. In order to address these problems, quality requirements and estimations need to be extended with models that can express imperfection such as tolerance. In addition, the evaluation mechanism for design alternatives needs to be extended, such that the evaluations can be performed with support for the imperfection models.

4.3.2 Imperfection Models for Quality Requirements

In this section, we present the first part of our approach, which consists of models that can capture the imperfect nature of quality requirements and estimations. As has been identified earlier, in both the quality requirements as well as the quality estimations it can be very difficult to determine the precise values required for a straightforward evaluation. In our approach we propose that the numeric values that are used in requirements and estimations are described with probability distributions and fuzzy sets, in addition to normal, “crisp” numbers. By using these models, it is possible to describe additional knowledge that exists about requirements and estimations, such as tolerance and variance. For the definition of these extensions, we first refine our definition of quality requirements to the following: a quality requirement is an interval of acceptable quality attribute values. In this definition, a quality requirement enforces its restriction on the acceptable behavior by means of an interval specification. For example, a maximum response time requirement of *rq* milliseconds corresponds to the acceptable interval [0, *rq*]. Any design alternative with a response time within this interval is acceptable, and a value outside the interval is not acceptable. Schematically, we can depict a quality requirement as follows:

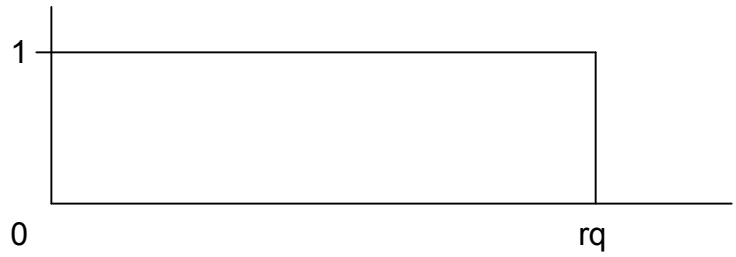


Figure 4.8 A Crisp Requirement

In Figure 4.8, the crisp requirement interval is described by a step function. Each possible value of the quality attribute is mapped to 0 (meaning not acceptable) or 1 (meaning acceptable). The sharp boundary of the requirement is represented by the discontinuous change from 1 to 0 at the value rq. This function is called the *membership function* of the interval.

Fuzzy Requirements

The first type of imperfection we address is tolerance in the specification of quality requirements. According to the definitions in chapter 2, this falls into the category of *impreciseness*, which means that a number of values beyond a specific boundary are still acceptable but less desirable. We model impreciseness in quality requirements by means of a *fuzzy interval*. In a fuzzy interval the membership function contains no discontinuous transition from 1 to 0, but rather has a gradual decline. A typical fuzzy requirement therefore looks as follows:

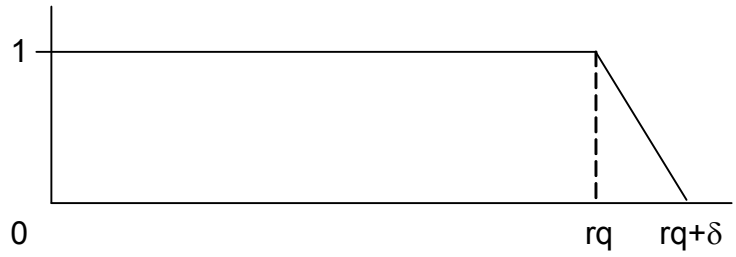


Figure 4.9 A Fuzzy Requirement

In Figure 4.9, the fuzzy interval is described by a piecewise linear membership function. The transition from completely acceptable (1) to completely unacceptable is a linear decline between *rq* and $rq+\delta$. The tolerance of the stakeholder with respect to the acceptable values is modeled by the linear decline. We refer to the shape of such membership functions as *semi-trapezoidal* and as a convenient shorthand notation for them we use $(x, x+\delta)$.

Probabilistic Requirements

The second type of imperfection that can occur in quality requirements relates to the definition of *uncertainty* in chapter 2. In this definition the actual value of an imperfect specification is not known at the current time point, but there will be a perfect specification in due time. Contrary to impreciseness, such as tolerance, uncertain specifications introduce the *risk* for each choice to be completely outside the boundaries, instead of being less desirable. Numeric expressions in an uncertain requirement specification are defined by a *probability density function*. Given a probability function f, the probability that the actual requirement value lies in the interval [a, b] is equal to:

$$\int_a^b f(x)dx$$

4.3.3 Imperfection Models for Quality Estimations

Imperfection in quality estimations belongs to the category of *uncertain* information, as defined in chapter 2. After completion of a software system, the quality attributes can be measured on a physical system, which means that estimations describe imperfection that will be resolved to a single value in due time. Depending on the nature of the imperfection, different types of imperfection models can be used for uncertain estimations. We define three types of models for uncertain estimations: *fuzzy estimations*, *probabilistic estimations* and *fuzzy probabilistic estimations*.

Fuzzy Estimations

The first imperfection model for quality estimations consists of *fuzzy estimations*. This imperfection model should be used in case the actual quality attribute value will fall within a particular interval. We model fuzzy estimations as extensions of crisp estimations by means of *triangular fuzzy numbers*. The membership function of a typical fuzzy estimation looks as follows:

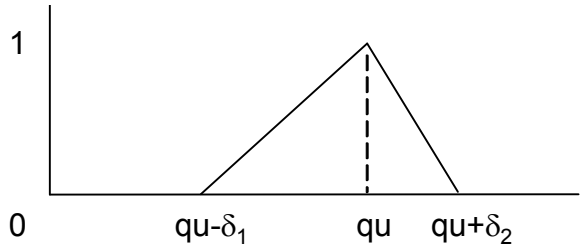


Figure 4.10 A Fuzzy Estimation

The fuzzy estimation in Figure 4.10 is described with a piecewise linear function. This estimation considers all values between $qu\text{-}\delta_1$ and $qu+\delta_2$ to be relevant to a certain degree, with the most prominent value being x. As a convenient shorthand notation for triangular fuzzy numbers we use ($qu\text{-}\delta_1$, qu, $qu+\delta_2$). Note that it is not necessary for the fuzzy estimation to be symmetrical, δ_1 and δ_2 can be different values. Additionally note, that a crisp estimation is the special case of fuzzy estimations with $\delta_1 = \delta_2 = 0$.

Probabilistic Estimations

The second imperfection model for quality estimations is *probabilistic estimations*. Probabilistic estimations should be used in case the actual behavior of a quality attribute depends on probabilistic events. For instance, in the RWS example the response time of the server depends on the amount of tasks that are waiting in the queue. The expected quality attribute values are described in a probabilistic estimation using a *probability density function f*. The average of estimation should correspond to the expectation value E of the distribution, given by:

$$E = \int_U xf(x)\,dx$$

Fuzzy Probabilistic Estimations

Finally, the third imperfection model for quality estimations is *fuzzy probabilistic estimations*. Probability distribution functions can be categorized in a number of families, such as normal distributions and exponential distributions. This categorization is based on the parameters that are needed to describe the desired behavior of these functions. For example, the exponential distribution is described by means of the parameter λ, as can be seen in section 2.4.2, and the normal distribution is described by means of its mean value and variance. In the case that estimations are made, where it is possible to determine the distribution family to be used but still very difficult to determine the precise parameters, a fuzzy probabilistic estimation can be used. Fuzzy probability distributions allow the parameters of density functions to be a fuzzy number, and therefore consider multiple parameter values to a degree. In this case, a fuzzy probabilistic estimation is made with a *fuzzy probability density function f* with fuzzy parameters $P_1, ..., P_n$. The average value of the quality estimation should correspond to the *defuzzification* of the fuzzy expectation value *E*, given by:

$$E[\alpha] = \{ \int_0^\infty x f_{p_1, p_n}(x) \,|\, p_1 \in P_1[\alpha], ..., p_n \in P_n[\alpha] \}$$

In this definition, *E* is defined in terms of α-cuts, which have been explained in section 2.4.3.

4.4 Comparison Operators for Requirements and Estimations

4.4.1 Introduction

As we have described in section 4.2.3, the assessment of design alternatives is commonly performed by comparing their expected quality attributes to the respective quality requirements. In the case where both the requirements and the estimations are expressed by crisp numbers, this can be achieved in a straightforward manner. However, in the previous section we have proposed extensions to the numeric expressions that can capture imperfect information. As a result, the comparison operators for the evaluation of imperfect requirements and estimations need to be defined for each combination of imperfection types. To achieve this, we define the *degree of acceptance* as a number in the range [0,1], which indicates to which degree an estimation falls within the acceptable interval of its respective requirement. A degree of acceptance of 1 indicates *completely acceptable* and a value of 0 indicates *completely unacceptable*. For crisp requirements and estimations, the degree of acceptance will always be either 0 or 1.We define the comparison operators for imperfection models such that they return a uniform degree of acceptance.

We define two functions for comparing requirements to estimations: $Comp_{avg}$ for comparing estimations to restrictions on the average, and $Comp_{max}$ for comparing estimations to restrictions on the maximum. The functions take a requirement and an estimation as parameters, and return the degree of acceptance of the estimation for this requirement. Since $Comp_{max}$ only differs from $Comp_{avg}$ in case of probabilistic or fuzzy probabilistic estimations, for crisp and/or fuzzy estimations both functions will be denoted by *Comp*. In these definitions, we indicate the requirement with an *a* and the *estimation* is with a *c*. The degree of acceptance of estimation c to requirement a therefore corresponds to *Comp(a, c)*.

In this section, we define the comparison operators for requirements and estimations. We have established that three types of requirements can be identified: crisp, imprecise and uncertain

requirements. Imprecise requirements are described by means of fuzzy requirements and uncertain requirements by means of probabilistic requirements. In addition, we have identified that uncertain estimations are described by probabilistic, fuzzy or fuzzy probabilistic estimations. In the following, we define the comparison operators for each type of requirement and estimation.

4.4.2 Comparison Operators for Crisp Requirements

The first type of requirements that can be provided for the design process are crisp requirements. In this section, we define the comparison operators for probabilistic, fuzzy and fuzzy probabilistic estimations with crisp requirements. Note that the comparison operators for crisp requirements and crisp estimations has been omitted, since this corresponds to the trivial comparison of crisp numbers.

Probabilistic Estimations

Let the probabilistic estimation *c* be a probability density function f. For a crisp average requirement *a* the degree of acceptance is:

$$Comp_{avg}(a, f) = 1 \quad , if \int_U xf(x)\,dx \le a$$

$$Comp_{avg}(a, f) = 0 \quad , otherwise$$

The degree of acceptance for a maximum requirement a is:

$$Comp_{max}(a, f) = \int_{x \le b} f(x)\,dx$$

Fuzzy Estimations

Let the fuzzy estimation be the triangular fuzzy number (c_1, c_2, c_3) and the crisp requirement be that the number is smaller than the number *a*. The degree of acceptance for evaluating fuzzy estimations with crisp requirements is:

$$Comp(a, (c_1, c_2, c_3)) = 0 \qquad , if\ a \le c_1$$

$$Comp(a, (c_1, c_2, c_3)) = \frac{a - c_1}{c_3 - c_1} - \frac{c_2 - a}{c_3 - c_1} \ln\left(\frac{a - c_1}{c_2 - a} + 1\right) \quad , if\ c_1 < a \le c_2$$

$$Comp(a, (c_1, c_2, c_3)) = \frac{a - c_1}{c_3 - c_1} \qquad , if\ c_2 = a\ \&\ a \le c_3$$

$$Comp(a, (c_1, c_2, c_3)) = \frac{a - c_1}{c_3 - c_1} - \frac{a - c_2}{c_3 - c_1} \ln\left(1 + \frac{c_3 - a}{a - c_2}\right) \quad , if\ c_2 < a \le c_3$$

$$Comp(a, (c_1, c_2, c_3)) = 1 \qquad , if\ a > c_3$$

In this definition we distinguished five cases, which correspond to the five positions the estimation value can take with respect to the requirement interval. The expressions are special cases of the more general expressions that compare fuzzy requirements with fuzzy estimations. We elaborate on the derivation of these general expressions in section 4.4.3 and Appendix B.

Fuzzy Probabilistic Estimations

Fuzzy probabilistic estimations are defined as a fuzzy probability distribution with density function f_{p_1, p_n}. The expectation value E for a fuzzy probability is a fuzzy number, which means that Comp_{avg} is equal to the function for comparing crisp requirements and fuzzy estimations. The degree of acceptance a fuzzy probabilistic estimation with density function f_{p_1, p_n} for a crisp average requirement a is:

$$Comp_{avg}(a, f_{p_1, p_n}) = Comp(a, E)$$

where

$$E[\alpha] = \{ \int_0^{\infty} x f_{p_1, p_n}(x) \mid p_1 \in P_1[\alpha], ..., p_n \in P_n[\alpha] \}$$

However, typically expectation values of fuzzy probability distributions are not triangular. Therefore, the definition in the previous section can not be reused, but Comp_{avg} can be derived according to the definitions in Appendix B. $\text{Comp}_{\text{max}}(a, f_P)$ is defined to be the defuzzification of the fuzzy number Q, which α-cuts are given by:

$$Comp_{max}(a, f_{p_1, p_n}) = Defuz(Q)$$

where

$$Q[\alpha] = \{ \int_{x \le a} f_{p_1, p_n}(x) \mid p_1 \in P_1[\alpha], ..., p_n \in P_n[\alpha] \}$$

This definition is similar to the definition for evaluating probabilistic estimations with crisp requirements, since the probability is used of the requirements interval. However, here the resulting probability is a fuzzy number. Therefore the function Defuz transforms the fuzzy probability Q to a crisp number. For the actual defuzzification, any of the standard defuzzification functions can be used as identified in section 2.4.4.

4.4.3 Comparison Operators for Imprecise Requirements

The first type of imperfection we have identified in quality requirements is *impreciseness*, which we have modeled with *fuzzy requirements*. In this section, we define the comparison operators for crisp, probabilistic, fuzzy and fuzzy probabilistic estimations with imprecise requirements.

Crisp Estimations

For the evaluation of crisp estimations with fuzzy requirements we define the degree of acceptance to be equal to the membership value of the crisp estimation *c* in the fuzzy interval (a_1, a_2):

$$Comp((a_1, a_2), c) = 1 \quad , if\ c \le a_1$$

$$Comp((a_1, a_2), c) = \frac{a_2 - c}{a_2 - a_1}, if\ a_1 < c \le a_2$$

$$Comp((a_1, a_2), c) = 0 \quad , if\ c > a_2$$

This function is a special case of the more general function that is used to compare fuzzy requirements with fuzzy estimations, given in Appendix B.

Probabilistic Estimations

The second type of estimation is a probabilistic estimation. Let the estimation *c* be given by density function *f* with expectation value *m*. The degree of acceptance for a fuzzy requirement (a_1, a_2):

$$Comp_{avg}((a_1, a_2), m) = 1, if\ m \le a_1$$

$$Comp_{avg}((a_1, a_2), m) = \frac{a_2 - m}{a_2 - a_1}, if\ a_1 < m \le a_2$$

$$Comp_{avg}((a_1, a_2), m) = 0, if\ m > a_2$$

For the definition of $Comp_{max}$ we have to consider the tolerance area of the fuzzy requirement in the evaluation. When comparing crisp requirements to probabilistic estimations the following expression is used:

$$\int_{x \le a} f(x)\, dx$$

This can be written as:

$$\int_0^{\infty} f(x)\mu(x)dx \text{ , with } \mu(x) = 1 \text{ for } 0 \le x \le a \text{ and } \mu(x) = 0 \text{ otherwise.}$$

This means that $\mu(x)$ is the membership function of the crisp requirement interval. When this is extended to fuzzy requirements, and we have a fuzzy maximum requirement with membership function A, we get:

$$Comp_{max}(A, f) = \int_0^{\infty} A(x)f(x)dx$$

Fuzzy Estimations

Let the fuzzy estimation be the triangular fuzzy number (c_1, c_2, c_3) and the fuzzy requirement be the fuzzy interval (a_1, a_2). One of the possible positions these fuzzy entities can take with respect to each other is depicted in Figure 4.11.

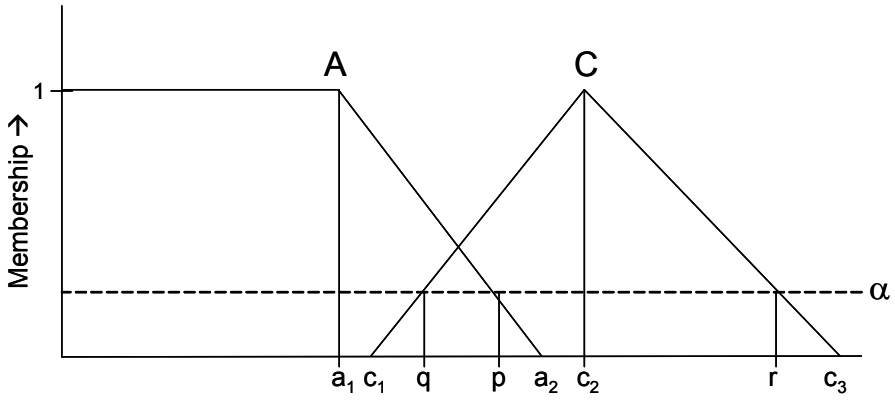


Figure 4.11 Comparing Fuzzy Requirements and Fuzzy Estimations

In this figure, the fuzzy estimation C is evaluated with respect to the fuzzy requirement A. This is done by determining the portion of the interval [q, r] at height α that lies inside the requirement interval]-∞, p]. The degree of acceptance is found by generalizing this for all α:

$$Comp((a_1, a_2), (c_1, c_2, c_3))$$

$$= 1 \qquad , if\ c_2 \le a_1\ \&\ c_3 \le a_2$$

$$= \frac{a_2 - c_1}{c_3 - c_1} + \frac{a_1 - c_2}{c_3 - c_1} \ln\left(1 + \frac{c_3 - a_2}{a_1 - c_2}\right) \qquad , if\ c_2 < a_1\ \&\ c_3 > a_2$$

$$= \frac{a_2 - c_1}{c_3 - c_1} \qquad , if\ c_2 = a_1\ \&\ c_3 > a_2$$

$$= \frac{a_2 - c_1}{c_3 - c_1} - \frac{c_2 - a_1}{c_3 - c_1} \ln\left(\frac{a_2 - c_1}{c_2 - a_1} + 1\right) \qquad , if\ c_2 > a_1\ \&\ c_1 < a_2\ \&\ c_3 \ge a_2$$

$$= 1 + \frac{c_2 - a_1}{c_3 - c_1} \ln\left(1 - \frac{c_3 - c_1}{c_2 - a_1 + a_2 - c_1}\right) \qquad , if\ c_2 > a_1\ \&\ c_3 < a_2$$

$$= 0 \qquad , if\ c_2 > a_1\ \&\ c_1 \ge a_2$$

The expressions above distinguish between six cases, which correspond to the six positions a fuzzy estimation can take with respect to the fuzzy requirement interval. A more elaborate explanation of these expressions and their derivation is given in Appendix B.

Fuzzy Probabilistic Estimations

Fuzzy probabilistic estimations are defined as a fuzzy probability distribution with density function f_{p_1, p_n}. The expectation value E for a fuzzy probability is a fuzzy number, which means that Comp_{avg} is equal to the function for comparing fuzzy requirements and fuzzy estimations. The degree of acceptance a fuzzy probabilistic estimation with density function f_{p_1, p_n} for a fuzzy average requirement a with membership function A is:

$$Comp_{avg}(a, f_{p_1, p_n}) = Comp(a, E)$$

where

$$E[\alpha] = \{ \int_0^\infty x f_{p_1, p_n}(x) \,|\, p_1 \in P_1[\alpha], ..., p_n \in P_n[\alpha] \}$$

Typically, expectation values of fuzzy probability distributions or not triangular. Therefore, the definition in the previous section can not be reused, but Comp_{avg} can be derived according to the definitions in Appendix B. The degree of acceptance of a fuzzy probabilistic estimation with density function f_{p_1, p_n} for a fuzzy average requirement a with membership function A is:

$$Comp_{max}(a, f_{p_1, p_n}) = Defuz(Q)$$

where

$$Q[\alpha] = \{ \int_0^\infty A(x) f_{p_1, p_n}(x) \,|\, p_1 \in P_1[\alpha], ..., p_n \in P_n[\alpha] \} \}$$

This definition is similar to the definition for evaluating probabilistic estimations with fuzzy requirements, since the requirements interval is multiplied with the density function. However, here the resulting probability is a fuzzy number. Therefore the function *Defuz* transforms the fuzzy number Q to a crisp number. For the actual defuzzification, any of the standard defuzzification functions can be used as identified in section 2.4.4

4.4.4 Comparison Operators for Uncertain Requirements

The second type of imperfect information we have identified for quality requirements is *uncertainty*, for which we have defined a model based on probability density functions. Like to other types of imperfect requirements, the comparison operators need to be defined for crisp, fuzzy and fuzzy probabilistic estimations. For the evaluation of crisp and probabilistic estimations, the standard operations from probability theory can be applied, much like we have done in the definitions above. For the evaluation of fuzzy estimations with probabilistic requirements we adopt the definition probabilistic requirements and fuzzy estimations. Let the requirement *a* be given by density function *f* with expectation value *m*. The degree of acceptance to fuzzy estimation c with membership function *C* is:

$$Comp_{avg}(m, (c_1, c_2)) = 0, \text{ if } m \leq c_1$$

$$Comp_{avg}(m, (c_1, c_2)) = \frac{c_2 - m}{c_2 - c_1}, \text{ if } c_1 < m \leq c_2$$

$$Comp_{avg}(m, (c_1, c_2)) = 1, \text{ if } m > c_2$$

The degree of acceptance for a restriction on the maximum is:

$$Comp_{max}(f, C) = \int_0^{\infty} f(x)C(x)dx$$

Above, we have defined models for the specification and evaluation of imperfect quality requirements and estimations. When we incorporate these imperfection models in the evaluation steps of the design tree model, we are now capable of including imperfect information in the software development process. The quality based evaluation of design alternatives, which has been defined in section 4.2.3, can compare imperfect requirements and non-crisp estimations based on the definitions in this section.

4.5 Case Study: Storm Surge Barrier

To demonstrate the use of imperfection models for requirements and estimations, we revisit the Remote Water Sensor example of section 4.2.5. In this example we have resolved three design issues by identifying three alternatives per issue and evaluating them with respect to their expected quality. The results of this evaluation and selection process are given in Table 4.5.

Table 4.5 Design Decisions Evaluation

	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1									
Opt. 1.1	400	400	∞	180	1	1	1	1	1
Opt. 1.2	350	350	∞	190	1	1	1	1	1
Opt. 1.3	300	300	∞	230	1	1	1	0	0
Design Decision 2 after choosing option 1.2									
Opt. 2.1	400	∞	0	190	1	0	0	1	0
Opt. 2.2	400	650	∞	200	1	1	1	1	1
Opt. 2.3	450	450	13	205	1	1	1	1	1
Design Decision 3 after choosing option 2.3									
Opt. 3.1	510	510	9.5	205	0	1	1	1	0
Opt. 3.2	500	500	10	225	1	1	1	1	1
Opt. 3.3	850	850	12	300	0	0	1	0	0

In this table the estimations were made using "traditional" crisp numeric expressions and compared to crisp quality requirements. The table shows that the development process results in a system design with only one possible option for the third decision. The crisp evaluations of option 3.1 and option 3.2 are very similar as well as very close to the boundary. When we apply our approach to these evaluations, we expect the overall quality of these options to be

much closer. In the following we introduce the three different types of imperfection into the estimations and evaluate them with perfect and imprecise (fuzzy) requirements.

4.5.1 Selection of Alternatives with Uncertainty in Quality Estimations

First, we introduce uncertainty in the estimations that are made on the expected quality of the final system. The difference between the estimated quality and the eventual quality of the system can have a considerable impact on the design process. We evaluate the uncertain estimations with the crisp requirement specification of section 4.2.5.

Probabilistic Estimations for Performance

Let for our example the performance estimations be based on probability models, rather than crisp numbers. This represents the fact that at any given time the response time of the system depends on the amount of requests that are waiting in the request queue of the Remote Water Sensor. In our case we assume that the exponential distribution, given by $f(x) = \lambda * e^{-\lambda x}$, is used to model the expected response times. Its expectation value is $1/\lambda$, which means that the actual estimation is performed with respect to its parameter λ of the density function. We re-evaluate the results from the table using the definitions for crisp requirements and probabilistic estimations. Note that in this and subsequent tables the new imperfect estimations and evaluations are indicated by bold numbers.

Table 4.6 Decisions Evaluation with Probabilistic Performance Estimations

	λ	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1										
Opt. 1.1	**1/400**	**400**	∞	∞	180	**1**	**0.803**	1	1	0.803
Opt. 1.2	**1/350**	**350**	∞	∞	190	**1**	**0.844**	1	1	0.844
Opt. 1.3	**1/300**	**300**	∞	∞	230	**1**	**0.885**	1	0	0
Design Decision 2 after choosing option 1.2										
Opt. 2.1	**1/400**	**400**	∞	0	190	**1**	**0.803**	0	1	0
Opt. 2.2	**1/400**	**400**	∞	∞	200	**1**	**0.803**	1	1	0.803
Opt. 2.3	**1/450**	**450**	∞	13	205	**1**	**0.764**	1	1	0.764
Design Decision 3 after choosing option 2.3										
Opt. 3.1	**1/510**	**510**	∞	9.5	205	**0**	**0.720**	1	1	0
Opt. 3.2	**1/500**	**500**	∞	10	225	**1**	**0.727**	1	1	0.727
Opt. 3.3	**1/850**	**850**	∞	12	300	**0**	**0.534**	1	0	0

In Table 4.6 describe the performance estimations, which are done based on exponential probability distributions. In the table this is shown by making the maximum estimated response time infinitely large (indicated by ∞). The parameter that indicates which exponential density function is used, is indicated in the second column. The value for Q_2 in the table corresponds to the fraction of the response times that fall in the requirement interval [0, 650]. The evaluation results remain largely the same, although we can see that the use of probabilistic estimations gives us an indication of the risk with respect to the performance quality.

Fuzzy Estimations of Reliability and Cost

The second type of uncertainty that can be introduced in quality estimations is by means of fuzzy estimations. For instance, instead of a total cost of 200 k€ for the Remote Water Sensor, the best specification that can be given is *approximately 200 k€*. For our case we take the estimations for both the reliability and the cost to be fuzzy estimations. In the table below the reliability and cost attributes are expressed and evaluated using triangular fuzzy numbers (see section 4.3.3).

Table 4.7 Decisions Evaluation with fuzzy estimations for reliability and cost

	λ	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1										
Opt. 1.1	1/400	400	∞	∞	**(155,180,205)**	1	0.803	**1**	**1**	0.803
Opt. 1.2	1/350	350	∞	∞	**(165,190,215)**	1	0.844	**1**	**1**	0.844
Opt. 1.3	1/300	300	∞	∞	**(205,230,255)**	1	0.885	**1**	**0.239**	0.212
Design Decision 2 after choosing option 1.2										
Opt. 2.1	1/400	400	∞	**0**	**(165,190,215)**	1	0.803	**0**	**1**	0
Opt. 2.2	1/400	400	∞	∞	**(175,200,225)**	1	0.803	**1**	**1**	0.803
Opt. 2.3	1/450	450	∞	**(12,13,14)**	**(180,205,230)**	1	0.764	**1**	**0.983**	0.751
Design Decision 3 after choosing option 2.3										
Opt. 3.1	1/510	510	∞	**(8.5,9.5,10.5)**	**(180,205,230)**	0	0.720	**0.076**	**0.983**	0
Opt. 3.2	1/500	500	∞	**(9,10,11)**	**(200,225,250)**	1	0.727	**0.5**	**0.5**	0.182
Opt. 3.3	1/850	850	∞	**(11,12,13)**	**(275,300,325)**	0	0.534	**1**	**0**	0

In Table 4.7 the reliability estimation ranges -1 second to +1 of the original, crisp estimation and the cost estimation ranges from -25 k€ to +25 k€ of the crisp estimation. As can be seen in the table, this small variation in the cost and reliability estimation results in a substantially different overall evaluation of the alternatives. For instance, the crisp reliability estimation for option 3.2 was exactly equal to the boundary of the requirement interval. However, with the fuzzy estimation, half of the variance is outside the requirement interval, which leads to a much lower evaluation. This reflects the fact that only a small variation in this case results in an alternative of unacceptable quality. Also we see that option 2.2 is now rated higher than option 2.3 compared to the crisp evaluation, and option 3.2 receives a very low quality evaluation compared with the value in the previous table.

Fuzzy Probabilistic Estimations for Performance

Fuzzy probabilistic estimation are used in the case it is difficult to precisely define the parameters of the density function that is used. Let for our example the estimation of performance be done with an exponential fuzzy probability distribution [Buckley2003]. This means that the parameter λ in $f(x) = \lambda * e^{-\lambda x}$ is replaced by a fuzzy number, denoted by λ_f. In our example, λ is replaced by a triangular fuzzy number $(\lambda-0.0005, \lambda, \lambda+0.0005)$, which corresponds to a variance in the expectation value of the response time. This leads to the following evaluation results:

Table 4.8 Decisions Evaluation with fuzzy probabilistic estimations for performance

	λ_f	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1										
Opt. 1.1	**(1/400-1/2000, 1/400, 1/400+1/2000)**	$\mathbf{f_{400}}$	∞	∞	(155,180,205)	**1**	**0.798**	1	1	0.798
Opt. 1.2	**(1/350-1/2000, 1/350, 1/350+1/2000)**	$\mathbf{f_{350}}$	∞	∞	(165,190,215)	**1**	**0.840**	1	1	0.840
Opt. 1.3	**(1/300-1/2000, 1/300, 1/300+1/2000)**	$\mathbf{f_{300}}$	∞	∞	(205,230,255)	**1**	**0.882**	1	0.239	0.210
Design Decision 2 after choosing option 1.2										
Opt. 2.1	**(1/400-1/2000, 1/400, 1/400+1/2000)**	$\mathbf{f_{400}}$	∞	0	(165,190,215)	**1**	**0.798**	0	1	0
Opt. 2.2	**(1/400-1/2000, 1/400, 1/400+1/2000)**	$\mathbf{f_{400}}$	∞	∞	(175,200,225)	**1**	**0.798**	1	1	0.798
Opt. 2.3	**(1/450-1/2000, 1/450, 1/450+1/2000)**	$\mathbf{f_{450}}$	∞	(12,13,14)	(180,205,230)	**0.872**	**0.758**	1	0.983	0.650
Design Decision 3 after choosing option 2.3										
Opt. 3.1	**(1/510-1/2000, 1/510, 1/510+1/2000)**	$\mathbf{f_{510}}$	∞	(8.5,9.5,10.5)	(180,205,230)	**0.301**	**0.713**	0.076	0.983	0.016
Opt. 3.2	**(1/500-1/2000, 1/500, 1/500+1/2000)**	$\mathbf{f_{500}}$	∞	(9,10,11)	(200,225,250)	**0.438**	**0.720**	0.5	0.5	0.079
Opt. 3.3	**(1/850-1/2000, 1/850, 1/850+1/2000)**	$\mathbf{f_{850}}$	∞	(11,12,13)	(275,300,325)	**0**	**0.522**	1	0	0

In the table f_x stands for a fuzzy number with the highest degree of membership at *x*. This is non-triangular fuzzy number, which is the fuzzy average of the fuzzy probability distribution. For more information, see chapter 2. From the table it can be seen that a fuzzy probabilistic estimation for reliability severely influences the degree of fulfillment for individual quality attributes. Option 3.2 even has an evaluation of 0.079, while in the crisp evaluation it had an evaluation of 1. Clearly this is caused by the fact that all the estimations were very close or even equal to the boundary of the interval of acceptable values, which means that a slight variation has a considerable impact. The results in the table can be depicted in the following design tree.

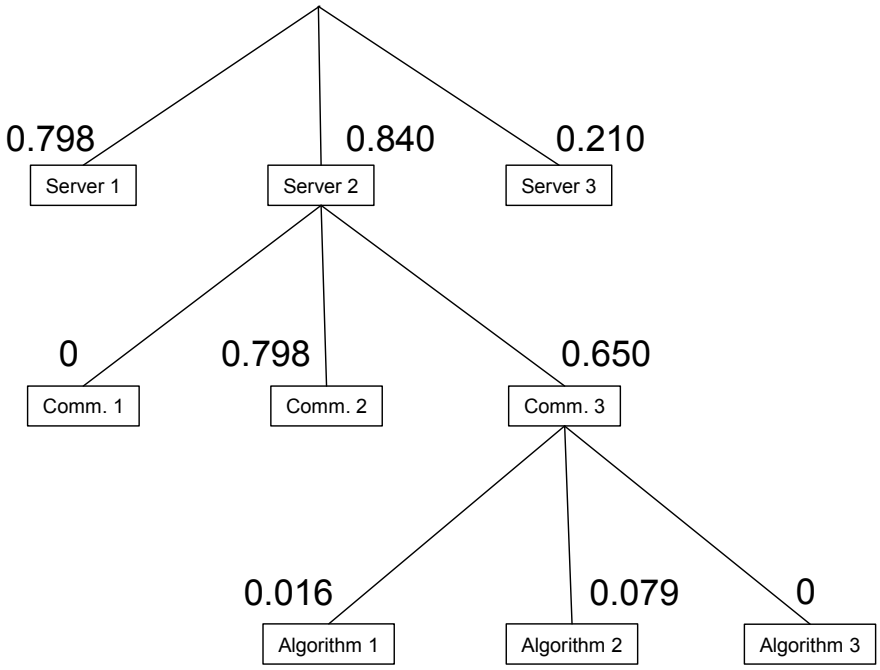


Figure 4.12 Design Tree with Uncertain Estimations

In Figure 4.12, it can be seen that the evaluation of the alternatives during the first two design decisions has been considerably optimistic in the case of crisp requirements. When the uncertainty in the estimations is modeled explicitly using probabilistic and fuzzy set models, the alternatives have a much lower quality evaluation than the crisp case. Also, by including imperfect information in the evaluation process, the evaluation of options 3.1 and 3.2 become almost zero, which implies that is better to continue with the "Comm 2" node, since from here a much better quality is expected.

4.5.2 Selection of alternatives with Impreciseness in Quality Requirements and Uncertainty in Quality Estimations

As in the estimations, imperfection can also manifest itself in the requirements. However, in the case of requirements it represents a certain tolerance with respect to the requirement boundary. For the Remote Water Sensor example we introduce a certain amount of tolerance in the boundaries of PR1, PR2, PR3 and PR4, which is represented by fuzzy requirements.

PR1: The average response time must be within (500, 600) milliseconds

PR2: The maximum response time must be within (650, 750) milliseconds

PR3: In case of failure, the system must be able to keep running for (8, 10) seconds

PR4: The system must cost no more than (225, 235) k€

To analyze how these fuzzy requirements influence the evaluation result, we evaluate the alternatives with crisp, fuzzy and fuzzy probabilistic estimations. For the evaluation with crisp estimations, this leads to the following table:

Table 4.9 Evaluating fuzzy requirements with crisp estimations

	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1									
Opt. 1.1	400	400	∞	180	1	1	1	1	1
Opt. 1.2	350	350	∞	190	1	1	1	1	1
Opt. 1.3	300	300	∞	230	1	1	1	0.5	0.5
Design Decision 2 after choosing option 1.2									
Opt. 2.1	400	∞	0	190	1	0	0	1	0
Opt. 2.2	400	650	∞	200	1	1	1	1	1
Opt. 2.3	450	450	13	205	1	1	1	1	1
Design Decision 3 after choosing option 2.3									
Opt. 3.1	510	510	9.5	205	0.9	1	0.75	1	0.675
Opt. 3.2	500	500	10	225	1	1	1	1	1
Opt. 3.3	850	850	12	300	0	0	1	0	0

In Table 4.9 the evaluations are largely the same as with crisp requirements. However, the evaluation of option 3.1 changes considerably. With crisp requirements for both the average performance and the reliability the estimations did not offer sufficient quality. But since the estimations were very close to the requirement interval, the use of fuzzy requirements changes the evaluation of reliability from 0 to 0.75 since 9.5 is inside the tolerance range of the fuzzy requirement. In an analogue manner the evaluation of the average performance changes from 0 to 0.9. While option 3.2 still has the best overall quality, option 3.1 now is evaluated with considerably better quality than before. In a similar manner the evaluation of option 1.3 changes.

Probabilistic Estimations for Performance

As in the previous section, in the second step we estimate the response times of the design alternatives with exponential probability distributions. When we re-evaluate the design alternatives, this results in the following table:

Table 4.10 Evaluating fuzzy requirements with probabilistic estimations

	λ	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1										
Opt. 1.1	**1/400**	**400**	∞	∞	180	**1**	**0.826**	1	1	0.826
Opt. 1.2	**1/350**	**350**	∞	∞	190	**1**	**0.864**	1	1	0.864
Opt. 1.3	**1/300**	**300**	∞	∞	230	**1**	**0.903**	1	0.5	0.452
Design Decision 2 after choosing option 1.2										
Opt. 2.1	**1/400**	**400**	∞	0	190	**1**	**0.826**	0	1	0
Opt. 2.2	**1/400**	**400**	∞	∞	200	**1**	**0.826**	1	1	0.826
Opt. 2.3	**1/450**	**450**	∞	13	205	**1**	**0.788**	1	1	0.788
Design Decision 3 after choosing option 2.3										
Opt. 3.1	**1/510**	**510**	∞	9.5	205	**0.9**	**0.746**	0.75	1	0.504
Opt. 3.2	**1/500**	**500**	∞	10	225	**1**	**0.753**	1	1	0.753
Opt. 3.3	**1/850**	**850**	∞	12	300	**0**	**0.561**	1	0	0

In Table 4.10, it can be seen that the overall evaluations are somewhat lower, in conformance with the evaluations of probabilistic estimations with crisp requirements. Additionally, we again see that option 3.1 and 3.2 do not differ much with respect to their overall evaluation.

Fuzzy Estimations of Reliability and Cost

As with crisp requirements we now introduce fuzzy estimations for performance for the evaluation with fuzzy requirements. We again take a variance of one second for reliability and 25k€ for the system cost. In the table below the reliability and cost attributes are expressed and evaluated using triangular fuzzy numbers (see section 4.3.3).

Table 4.11 Fuzzy Requirements with fuzzy estimations for reliability and cost

	λ	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1										
Opt. 1.1	1/400	400	∞	∞	**(155,180,205)**	1	0.826	**1**	**1**	0.826
Opt. 1.2	1/350	350	∞	∞	**(165,190,215)**	1	0.864	**1**	**1**	0.864
Opt. 1.3	1/300	300	∞	∞	**(205,230,255)**	1	0.903	**1**	**0.257**	0.232
Design Decision 2 after choosing option 1.2										
Opt. 2.1	1/400	400	∞	**0**	**(165,190,215)**	1	0.826	**0**	**1**	0
Opt. 2.2	1/400	400	∞	∞	**(175,200,225)**	1	0.826	**1**	**1**	0.826
Opt. 2.3	1/450	450	∞	**(12,13,14)**	**(180,205,230)**	1	0.788	**1**	**1**	0.788
Design Decision 3 after choosing option 2.3										
Opt. 3.1	1/510	510	∞	**(8.5,9.5,10.5)**	**(180,205,230)**	0.9	0.746	**0.725**	**1**	0.487
Opt. 3.2	1/500	500	∞	**(9,10,11)**	**(200,225,250)**	1	0.753	**1**	**0.7**	0.527
Opt. 3.3	1/850	850	∞	**(11,12,13)**	**(275,300,325)**	0	0.561	**1**	**0**	0

In Table 4.11, we can see that the fuzzy estimations and fuzzy requirements have considerably influenced the evaluations compared to the completely crisp case. Intuitively, the evaluations describe the situation more accurately, since the estimations for a number of alternatives differed only slightly while their evaluations where completely different.

Fuzzy Probabilistic Estimations of Performance

Finally, we evaluate the design alternatives with the fuzzy requirements, while using fuzzy probabilistic estimations for the performance. We again assume an exponential distribution with fuzzy parameter *(λ-0.0005, λ, λ+0.0005)*, which corresponds to a variance in the expectation value of the response time. This leads to the following evaluation results:

Table 4.12 Decisions Evaluation with fuzzy probabilistic estimations for performance

	λ_f	Average Performance	Maximum Performance	Reliability	Cost	$Quality_1$	$Quality_2$	$Quality_3$	$Quality_4$	Overall Quality
Design Decision 1										
Opt. 1.1	**(1/400-1/2000, 1/400, 1/400+1/2000)**	$\mathbf{f_{400}}$	∞	∞	(155,180,205)	**1**	**0.908**	1	1	0.908
Opt. 1.2	**(1/350-1/2000, 1/350, 1/350+1/2000)**	$\mathbf{f_{350}}$	∞	∞	(165,190,215)	**1**	**0.933**	1	1	0.933
Opt. 1.3	**(1/300-1/2000, 1/300, 1/300+1/2000)**	$\mathbf{f_{300}}$	∞	∞	(205,230,255)	**1**	**0.956**	1	0.257	0.246
Design Decision 2 after choosing option 1.2										
Opt. 2.1	**(1/400-1/2000, 1/400, 1/400+1/2000)**	$\mathbf{f_{400}}$	∞	0	(165,190,215)	**1**	**0.908**	0	1	0
Opt. 2.2	**(1/400-1/2000, 1/400, 1/400+1/2000)**	$\mathbf{f_{400}}$	∞	∞	(175,200,225)	**1**	**0.908**	1	1	0.908
Opt. 2.3	**(1/450-1/2000, 1/450, 1/450+1/2000)**	$\mathbf{f_{450}}$	∞	(12,13,14)	(180,205,230)	**0.872**	**0.881**	1	1	0.881
Design Decision 3 after choosing option 2.3										
Opt. 3.1	**(1/510-1/2000, 1/510, 1/510+1/2000)**	$\mathbf{f_{510}}$	∞	(8.5,9.5,10.5)	(180,205,230)	**0.655**	**0.848**	0.725	1	0.403
Opt. 3.2	**(1/500-1/2000, 1/500, 1/500+1/2000)**	$\mathbf{f_{500}}$	∞	(9,10,11)	(200,225,250)	**0.829**	**0.853**	1	0.7	0.495
Opt. 3.3	**(1/850-1/2000, 1/850, 1/850+1/2000)**	$\mathbf{f_{850}}$	∞	(11,12,13)	(275,300,325)	**0**	**0.678**	1	0	0

In Table 4.12, the most obvious changes with respect to the crisp evaluation remain option 1.1, with an evaluation larger than 0, and options 3.1 and 3.2 with almost equal evaluation. From the application of imperfect information models in the quality based evaluation of design alter-

natives, we have now seen that the evaluation results more accurately reflect the risks and intuition of the software engineer. To analyze the influence on the design decisions, we now depict the design tree that results from the evaluations of Table 4.12.

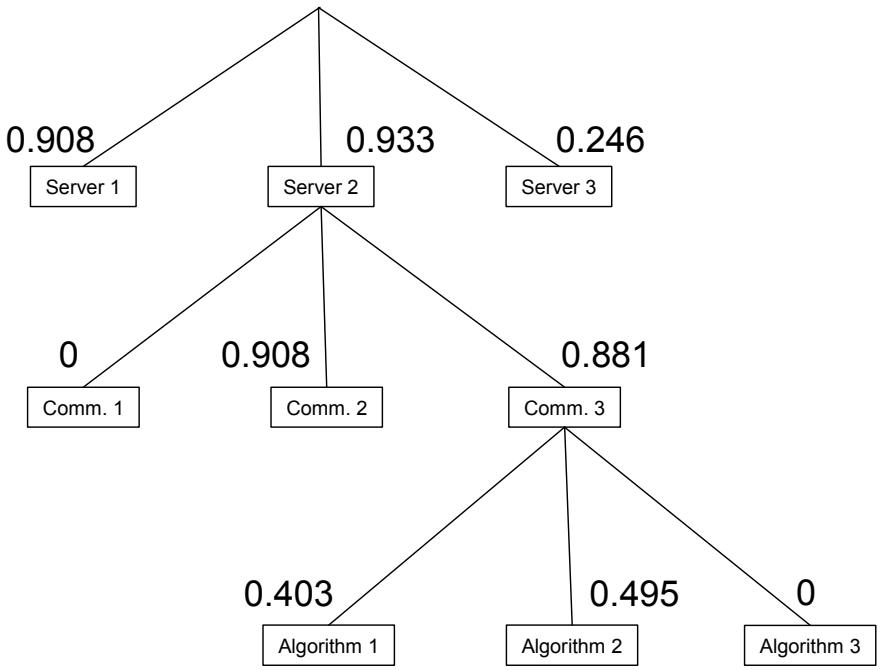


Figure 4.13 Design Tree with Uncertain Estimations and Imprecise Requirements

The sequence of design decisions and the selection of design alternatives with crisp requirements and estimations is depicted in Figure 4.13. The evaluation of the design alternatives has been replaced with the evaluations from Table 4.12. The design process with crisp requirements and estimations resulted in a system design with Server 2, Comm. 3 and Extrapolation Algorithm 2, which was the only one with acceptable quality. In the design tree with imperfect information we see that this node has an expected quality of 0.495, which does not differ much from option 3.1. Additionally, and even more importantly, we see that for the second design decision a node was selected with quality 0.881, while there was another option with a higher expected quality. The application of the design tree approach with imperfect information models would, in this example, have resulted in an alternative design than with crisp requirements and estimations.

4.6 Related Work

4.6.1 Traceability of Design Decisions in Software Engineering

Keeping track of the design decisions that are taken during the design process is not new. In [Parnas1976], the concept of tracing design decisions in a tree structure was proposed in order to identify the commonality and differences between intermediate program designs. This concept also underlies the design tree approach. However, our approach builds on this concept with the definition of configurable design strategies and the possibility of quality-based evaluation of design with both perfect and imperfect information. In the field of requirements traceability the relationship between intermediate design artifacts and the originating requirement(s) are made explicit. The models that have been proposed in this field can be classified according to the specific type of information they aim to capture, such as functional or

non-functional tracing, forward or backward tracing, etc. In case the design trees are used to capture design decisions, the solution is in the area of non-functional requirements tracing. Most approaches that haven been proposed to trace design decisions are based on decision tree models. In [Potts1988] and [Ran1996] alternative approaches are described for capturing design decisions and their motivations, which are similar to the design tree approach. Design artifacts are captured using a graph structure, as well as relevant design considerations. However, in contrast with the design tree approach, it is not possible to traverse the graph structure in order to decide on the subsequent design trajectory in a configurable manner. In [Cmitile1992], design decisions are recorded as annotations to enhance software maintenance. These annotations are used to trace the decisions that have been taken with respect to transforming an intermediate design into a design for a specific implementation language. In general, it can be said that requirement traceability approaches are custom made for a particular application area, and contain domain-specific model attributes. This makes it difficult to reuse and compare traceability models. For this purpose work has been done in creating reference models for requirements traceability [Ramesh2001]. The model aims to provide the relevant elements of a traceability model, to which only the domain specific elements need to be added. By adjusting the design tree model to conform to the basic elements of the reference model, these respective qualities can also be achieved. In the area of design rationale management, many approaches have been proposed, which can be used to capture relevant information that is the result of the design process. Depending on the nature of the design process, different types of approaches are used. In [Regli2000] a distinction is made between a feature-oriented and a process-oriented approach for capturing design rationale. In feature-oriented approaches, design rationale is captured in domains that are well-known and standardized. Process-oriented approaches emphasize design rationale as a history of the design process, and are used in domains where problems are vague and the solutions poorly understood. Examples of such approaches are [Lee1991] and [McCall1991]. However, where design rationale management approaches aim to capture the intuitions and validations of design decisions, they do not explicitly consider imperfection, even when they are aware of the fact that the domain contains this. In the Design Tree Model the design rationale based on the expected quality for each design decision is captured, and the approach supports imperfection in both requirements and estimations.

4.6.2 Modeling Imperfect Information in Design Processes

The most well-known area in software engineering in which the potential consequences of imperfect information are considered is risk management [Karolak1995]. In this area the influence of probabilistic events is analyzed in, for instance, software design processes. However, the techniques that are proposed in this field address a different type of imperfection than our approach. In our approach we try to facilitate imperfection in requirement specifications and quality estimations, and we have identified different types of imperfection that can occur. As such, our approach is not in particular a risk management approach, but rather a refinement of software development activities. In [Gregoriades2005], a scenario-based assessment method for non-functional requirements is presented. This approach focuses in particular on the human errors that can be introduced by slips of mind, difficulty of tasks, etcetera. The probabilities of these errors, and their effect on the correctness of the non-functional requirements is then assessed by use of Bayesian Belief Nets. The work presented in [Bubenko1994] focuses on the identification and removal of imperfection in requirements specification. This is achieved by the definition of a number of metamodels that reflect the business, their goals and the requirements. From these, a formal (perfect) definition of the requirements is derived. In [Egyed2006] an approach is proposed, which enables the systematic analysis of the removal of ambiguous specifications. By modeling the available information and restrictions using constraint networks, design alternatives can be evaluated with respect to consistency. In other disciplines, the influence of imperfect information on design activities is also recognized. For example, in

the field of mechanical engineering extensions have been proposed that can express fuzzy requirements in a manner comparable to the one presented in this chapter [Antonsson1996]. This approach presents the Level Interval Algorithm to reason with the representations for imperfection.

4.7 Discussion

How do we acquire the applicable fuzzy sets and probability distributions to describe imperfection in quality requirements and estimations?

The accuracy of the result of the design tree approach depends on how well the imperfection models describe the situation at hand. However, the usage of this approach is not a straightforward activity. While it is true that the use of imperfection models adds an extra level of difficulty to the design process, in our early experiments the definitions were quite natural for the users. The variance that might exist in quality requirements such as performance was easily captured by defining the boundaries of triangular fuzzy numbers. In addition, it should be noted that probability distributions have long been used to model, for instance, performance of computer systems with a probabilistic nature. Additionally, fuzzy set theory offers the possibility to use linguistic variables [Zadeh1975] to refer to standard definitions of fuzzy sets within a particular area. While the actual definition of the fuzzy sets and/or probability distributions is by no means trivial, it is important to note that in current methods all kinds of crisp design rules are used which are not justifiable. For example, in requirements specification methods nouns are identified as candidate classes, even while this is not generally accurate. From this perspective, using probability theory and fuzzy set theory, while requiring additional insights, can be considered to be more precise.

Can the inclusion of imperfect information in software development lead to systems that do not fulfil any requirements?

The models that are proposed in this thesis enable the consideration of imperfect information during software development. The specific nature of these models allows the software engineer to include multiple opinions from various stakeholders, which are modeled using, for example, fuzzy sets. With the inclusion of the inputs and opinions of multiple stakeholders, it could be possible that the restrictions on the final system become so numerous that it becomes impossible to fulfil any of them. While this prospect is not unthinkable, it should also be noted that the extensions and models that have been proposed are not aimed at this purpose. Requirement specifications should be made as precise as possible, and stakeholders should agree (to a certain degree) on what they expect from the software system. The goal of our approach is to support the design process with support for imperfection, such that the impact imperfection can have is minimized. Obviously, this is only possible within certain limits, which means that the initial requirements that are provided by the stakeholders should be a workable starting point. To facilitate the difficulties that can be encountered during this initial definition process, future work in this area can include for example conflict resolution techniques and precedence mechanisms.

Is the order in which design decisions are addressed important for the design tree approach?

As a prerequisite for the design tree approach an ordered set of design issues is assumed as input for the software design process. Since the selection of a design alternative for a given design issue can influence the available design alternatives and quality expectations for subsequent decisions, it seems necessary to address the design issues in the correct order for the design tree approach to work. However, from an optimization point-of-view this is not the

case. The design strategies always select the best node according to their respective interest, which in worst case scenarios means that the entire principle design tree will be explored. In the case that the order of design issues hampers the development process, this could mean that the design strategy advises to revisit this level of the design tree very frequently. While this requires considerable effort, eventually the correct alternatives will be selected and a satisfactory design will be found. Nonetheless, the order of design issues should be considered carefully, since a logical order with respect to the problems that are addressed will decrease the number of iterations. In addition, the frequent reiteration to a particular level of the design tree can indicate a misordering in the set of design issues. By analyzing the iterative behavior, software engineers can become aware of these problems and establish an improved order among the design issues.

Can fuzzy logic/fuzzy set theory and probability theory model the imperfection in quality requirements and estimations appropriately?

In our approach we use probability theory and fuzzy set theory to model the imperfection that can occur in the software development process. It can be questioned how well these models are able to capture the nature of the imperfection that can occur in design information. The nature of the imperfection does not necessarily correspond to the way in which imperfection is modeled in probability and fuzzy set theory. But while it is true that these models do not always reflect the actual nature of imperfection that can be found, it is certain that these models address the issue of imperfection more accurately than ignoring the imperfection in the design information, and trying to resolve it at later stages solely by iteration and incremental design. With the use of probability and fuzzy set theory we cover part of the imperfection that can occur in design information with well-known imperfection models, which address these types of imperfection in a correct manner.

4.8 Conclusions

In section 4.2 imperfect information in quality requirements and quality estimations and the cascading of errors in design decisions were identified as two important problems in the design of software systems. The first problem can lead to making wrong decisions during the design process, since quality assessments do not represent the current situation accurately. The second problem is a direct consequence of the sequential nature in which design decisions are taken, which causes errors in individual decisions to influence the correctness of all decisions hereafter. Additionally, even when it becomes clear that the current design has not been the right choice, it is not easy to step back through the design process and to determine the point from which to continue.

We have shown that imperfect information can be managed by capturing the nature of the imperfection. To accomplish this, we have made the explicit distinction between impreciseness in quality requirements and uncertainty in quality estimations. By capturing both types of imperfect information and their specific character with applicable models such as probability theory or fuzzy set theory, the design alternatives considered during the design process can be evaluated more accurately. Furthermore, the means of comparing different types of imperfect information are defined, to enable the software engineer to evaluate design alternatives in much the same manner as in current approaches. This has been demonstrated by applying the approach to an example case, where two design alternatives were estimated to have very similar qualities. In the traditional evaluation method one alternative was evaluated as being unsatisfactory, since the quality attributes were just outside the quality constraints. When the design alternatives were evaluated using our approach, the alternatives showed that the quality was quite comparable, much like what was expected. Also, our approach indicated that traditional

design approaches could not distinguish between two design alternatives, while our approach offered specific insights into the strong points and risks of each design alternative.

In addition we have shown that the design process can be supported by tracing the design decisions that are made. For each design decision the considered alternatives are logged, and the evaluations are made explicit. To accomplish this, we have facilitated the use of imperfection models in quality requirements, which restrict the allowed behavior of the system, and quality estimations, which describe the expected behavior of the system. The relationship between these elements is captured by a tree structure, which can be traversed in an algorithmic manner, such that the design space can be explored systematically. This approach is completed by the ability to define configurable design strategies, that can offer decision advice based on managerial motives, such as development time minimization or quality maximization. With these design strategies, iterative design is supported in a systematic manner by means of exploration of the design tree. By combining the design strategies with support for imperfect information models, corrective design through incremental design steps can be considered in the context of this imperfection. With this model, the state of the available information can be captured more accurately than is supported by traditional development methods. The reasoning and optimization capabilities of the design tree model ensure that the imperfection is considered in a systematic and correct manner. The design tree model ensures that the decision making process based on this information considers the imperfection and its consequential risks accordingly.

The design tree approach combined with the imperfect information models creates additional effort for the software engineers during the software development process, when performed manually. To support the software engineer in the application of this approach, tooling has been developed as part of this research. The tooling supports the software engineer with the tracing of design decisions and contemplated alternatives with support for imperfect information. The tooling that has been implemented for the approach is described in chapter 6. In chapter 7 we explore the applicability and usability of the approach by applying the tooling in a pilot study. The results of this pilot study are used to evaluate the approach proposed in this chapter.

CHAPTER

5

"Without even the safety valve of dreaming, he focused his prescient awareness, seeing it as a computation of most probable futures, but with something more, an edge of mystery--as though his mind dipped into some timeless stratum and sampled the winds of the future."

- From Frank Herbert's Dune [Herbert2005]

SOFTWARE PROJECT MANAGEMENT WITH PROBABILISTIC MARKET DEMANDS

5.1 Introduction

In general, software requirements do not remain constant during the lifetime of a software system. The new requirements force the software engineers to adjust the software system accordingly, which can range from minor revisions to complete redesigns of the existing software. Although there are differences in the definitions and the impact, several case studies have shown that a considerable amount of project costs is spent on this kind of software maintenance [Kniesel2002]. Changes in requirements do not only impact software systems, but can also result in the need to reschedule the software development process. For example, a change in market demands can make it necessary to redistribute human resources to ensure that the required software systems are delivered on time. For software that is developed in stand-alone projects this is not necessarily problematic, since for instance it can be possible to renegotiate the delivery date. However, with the increased focus on application frameworks and product line architectures, software developers are faced with multiple products and product families, each of which are demanded at different points in time. It is up to the software engineer and the project manager to deliver the desired products on time, or they will miss out on the opportunity. Timely delivery is, therefore, very important for the success of software products, but the effective scheduling of the implementation is hindered by the uncertainty about the changes that can occur in future market demands.

This chapter proposes an approach for systematic derivation of resource allocation schedules, which can be used during the development of product families under uncertain market demand expectations. By estimating changes in market demands and constructing models for the representation of demand scenarios and allocation decisions, the approach determines resource allocation advice based on the workstate and demand characteristics.

5.2 Resource Scheduling Problems due to Uncertain Market Demands

5.2.1 Introduction

In recent years, software has been applied in an increasing number of areas. In order to address the increasing demand for software products, software design has focused on the systematic reuse of generic parts and components with, for example, application frameworks [Fayad1999] and product line architectures [Pohl2005]. Both approaches aim to come to a design from which a range of similar products can be derived, with product lines facilitating the reuse on an architectural level and application frameworks on an implementation level. To facilitate this reuse of functionality, a distinction is made between *commonalities* and *variabilities*. Commonalities are components or architectural entities that will be used in a considerable number of the products. These commonalities become reusable assets in the design, which can be used as building blocks of core functionality. The variabilities are the properties that can differ from product to product, and as such can not be captured in reusable assets. By using the commonality components and extending them with product specific attributes, a range of products can be produced more efficiently. This approach is in particular successful when a range of similar products should be delivered that share large parts of functionality.

While the implementation of product families with these approaches becomes more cost-effective, their design and implementation is a difficult and costly operation. In addition to the implementation of reusable assets, reuse and adoption during the production of new products must be facilitated. Nonetheless, it is important that the framework implementation does not interfere with the implementation of these products, to ensure the timely delivery of products with respect to the market demands. However, typically the amount of work exceeds the available resources, and, as a result, the implementation order of the commonalities and products needs to be scheduled, so that the implementation and delivery of products is possible at the time at which they are demanded.

5.2.2 Scheduling Software Development Processes

Assume that a large set of components has to be implemented in a time span of 2 to 4 years. These components will be used to create several different software systems to fulfil specific market demands. The base functionality of these systems is provided by reusable components, which means that a dependency relationship exists between these entities. For the timely delivery of systems it is crucial to deliver the essential components first, before the customers demand systems that incorporate these components. As a result, a higher priority must be given to the components that are needed for products that are demanded first. However, the determination of an effective implementation schedule is severely hampered by the complexity of the dependency relations and the uncertainty about future market demands. To ensure that these inputs are considered accordingly, an approach is needed that can systematically compare resource allocation schedules with respect to probabilistic market demand expectations.

To illustrate these problems, we consider a software company that produces and maintains software systems to be used by insurance companies. It is assumed that the software company delivers a dedicated software system for each insurance product. In addition, the software company aims to market the same software system to multiple insurance companies worldwide. Insurance companies can tailor these systems according to their specific policies. Generally, insurance companies have a large customer base, with a variety of insurance products and policies. For the software company, to a certain degree it is possible to deal with the complexity of evolution by adopting application frameworks and object-oriented and component-oriented techniques. The components provide the common functionality and can be reused multiple times. From these components basis products are derived on a per customer, which means

products can be instantiated many times from a single set of components. Since there may be many options for scheduling the implementation effort, ideally the project managers should be able to compare all the relevant options and select the best configuration that fulfils the timing and resource constraints. For the example case, this means that the expected optimal planning of resources for a given time span should be determined to maximize profit in selling insurance software systems.

A number of software development methods suggest prioritisation of requirements, e.g. [Jacobson1999]. For this purpose, several publications and tools have proposed decision-making processes that support prioritization and scheduling of component implementation trajectories. However, the proposed approaches assume a fixed and stable set of requirements and therefore are not directly suitable for dealing with the probabilistic nature of the changes in market demands. While configuring software processes with respect to available resources is not new [Podorozhny1999], in contrast to existing methods also the demands for future systems and changes in requirements must be considered.

Most of the aforementioned approaches are based on ranking approaches such as the analytic hierarchy process (AHP) method [Karlsson1997] [Salo1997] [Forman2001]. In the AHP method, a number of entities are rated based on pairwise comparisons. Each pair of elements is rated with respect to each other based on some criterion. This pairwise comparison process is repeated for all criteria that are relevant for the ranking. In the next step the criteria themselves are compared in a pairwise manner, and the resulting rating of the criteria is used together with the ratings of the entities per criterion to establish an overall rating among the entities. The AHP method has been applied to various kinds of problems such as resource allocation and risk assessment. The main problem with approaches like AHP is that a prioritization of entities is achieved by comparisons that are based on human intuition. In the case of changing market demands, approaches like AHP are not usable since they can not take the probabilistic nature of the changes in market demands into account. Rather than establishing a rating amongst the components to be implemented, the project manager needs scheduling advice that identifies the most important components based on the expected market situations. In this chapter, we propose a model that captures the uncertainty of future market demands using a graph-based structure that includes probabilities. Additionally, we propose a model that represents the possible resource allocation schedules. The combination of these two models enables the evaluation of the possible allocation schedules with respect to the probabilistic market demand expectations. As a result, we can determine the schedule that gives an optimal result with respect to a particular goal such as cost or implementation time. The novelty of the approach lies in the possibility to explicitly consider probabilistic changes in the market demand while scheduling the implementation of complex software systems. The specification is done based on probabilistic market demand scenarios, which can automatically be generated from demand expectations. The model ensures that the decision making process based on this information considers the probabilistic changes and their consequential risks accordingly.

5.3 Optimized Allocation of Resources

5.3.1 Introduction

Consider a software company that produces and maintains software systems to be used by insurance companies. It is assumed that the software company delivers a dedicated software system for each insurance product. In addition, the software company aims to market the same software system to multiple insurance companies worldwide. Insurance companies can tailor these systems according to their specific policies. Generally, insurance companies have a large customer base, with a variety of insurance products and policies. Of course, the insurance product characteristics are not static but they evolve in accordance with the needs of society. For example, a growing trend in the insurance market is the demand for tailored and personal-

ized insurances. For the software company, to a certain degree it is possible to deal with the complexity of this evolution by adopting application frameworks, object-oriented and component-oriented techniques. Application frameworks are programs that capture the generic parts of several software systems and encapsulate them in reusable components. Actual software systems can be produced by composition of several of these components. The components are identified by discovering the variabilities and commonalities in the problem domain.

The software company has to schedule the implementation of the application framework, and this must be aligned with the demand for insurance products. To achieve the implementation, the software company has a number of software engineers that together form the *resource pool*. From this resource pool, software engineers can be assigned on a per week basis, which means the minimal amount of resources that can be assigned is 40 person-hours. This minimum amount is referred to as the *block size*. For example, with four employees the resource pool every week equals four person-weeks. The block size is equal to one person-week, since a person can only be assigned to one task at the time.

The difficulty of scheduling the implementation of application frameworks is the result of two influences: the uncertainty about market demand expectations and the vast number of possible decisions with respect to resource allocation. Project managers can assign the available resources in many different ways, but it is very difficult to systematically compare the allocation schemes, since they must be evaluated against uncertain market demand expectations. To address these problems, we propose an approach that consists of three steps:

1. *Determine probabilistic estimations for market, cost and profit changes*
2. *Construct the possible market scenarios and resource schedules that need to be considered*
3. *Evaluate all possible schedules with respect to the scenarios and select the most profitable schedule*

In the first step of this approach the relevant properties of the market and the application framework are defined. Based on these definitions, in the second step, a systematic representation is made of the possible demand scenarios and resource allocation schedules. In the third step the combination of these representations is evaluated to determine the schedule that maximizes the expected profit. In this section, we first define a graph-based representation for probabilistic demand scenarios, followed by a model for resource allocation schedules. Finally, we combine these two models and define an optimization approach that determines a profit-optimized production plan.

5.3.2 Modeling Uncertain Market Demands using Scenarios

The approach proposed in the previous paragraph identifies the need for a model that captures the possible demand configurations that can come from the market. Since the implementation of the application framework is scheduled in a finite amount of steps or points in time, we have to consider the market demand for our products at these points. We define the *time horizon* to be the amount of decision or time points that need to be considered for the resource allocation schedule. The time points are numbered *0, 1, ..., Time Horizon-1*. In our definition of the market, the demand for products can change from one time point to the next, which means that every product can be demanded by a certain amount of customers at each time point. The software company at any point can decide to implement one or more products for these customers, at which points they add the value of these contracts to their portfolio. When the software company decides not to implement products, the potential customer will be served by other software companies and the chance for a contract is lost. Let the products be numbered *0, 1, ..., $\#_{Products}$-1*. A demand is defined to be a row of numbers with length $\#_{\text{Products}}$, where each number represents the demanded amount of the respective product. To describe the market demand, we define the demands at a particular point in time as a *demand state*. A demand state

is a pair, consisting of a time point and a demand. Additionally, we define a scenario to be a mapping from time points to demands. For each scenario *Sc*, *Sc(i)[p]* denotes the demand at time *i* for product *p*.

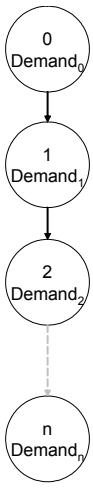


Figure 5.1 Scenario Graph

We represent scenarios in a systematic manner using a *Scenario Graph*. In a scenario graph, each node represents a market demand state and there exist links from states with time *i* to states with time *i+1*. The depth of the graph corresponds to the amount of time points in the time horizon. In Figure 5.1 a scenario graph is depicted, which contains a single market demand scenario. In this graph no events are considered that lead to different demand states, which means this is a deterministic description of future market demands. To include probabilistic changes in market demands, a state at time i can be succeeded by multiple states at time i+1 and each arrow in the scenario graph has a probability, represented by a number larger than 0 and smaller or equal to 1. Furthermore, the sum of the probabilities of all outgoing arrows from a single node must sum up to 1.

In a scenario graph, each path from the root node to a leaf node represents a single scenario. It is now possible to represent probabilistic predictions of future market demands using a scenario graph. For example, let *DemandC* and *DemandD* be two different demands. In addition, consider that the probability going to a demand state (t+1, DemandC) from (t, DemandD) is given by $\beta(t)$, and going to (t+1, DemandC) from (t, DemandD) by $\alpha(t)$. The scenarios that can occur over a time horizon of three decision points, assuming the starting demand state is (0, DemandC) are depicted in the scenario graph in Figure 5.2.

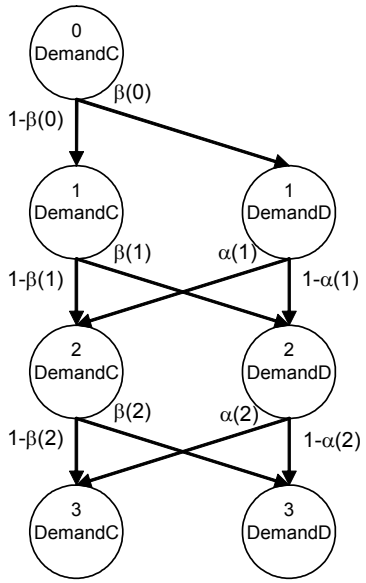


Figure 5.2 Scenario Graph

There are eight possible scenarios. As a reference these eight scenarios and their probability of occurrence is listed in Table 5.1.

Table 5.1 Demand Scenarios

Sc(0)	Sc(1)	Sc(2)	Sc(3)	Probability
DemandC	DemandD	DemandD	DemandD	β(0)*(1-α(1))*(1-α(2))
DemandC	DemandD	DemandD	DemandC	β(0)*(1-α(1))*α(2)
DemandC	DemandD	DemandC	DemandD	β(0)*α(1)*β(2)
DemandC	DemandD	DemandC	DemandC	β(0)*α(1)*(1-β(2))
DemandC	DemandC	DemandD	DemandD	(1-β(0))*β(1)*(1-α(2))
DemandC	DemandC	DemandD	DemandC	(1-β(0))*β(1)*α(2)
DemandC	DemandC	DemandC	DemandD	(1-β(0))*(1-β(1))*β(2)
DemandC	DemandC	DemandC	DemandC	(1-β(0))*(1-β(1))*(1-β(2))

In the first four columns of this table, the demands at this time point are described. Note that the scenario graph is very similar to a probabilistic automaton [Stoelinga2002].

5.3.3 Modeling Allocation Strategies using Sequential Allocation

The second part of our approach consists of a model that captures the possible human resource allocations that can be chosen during the implementation time of the components and products. Let the components of the framework be numbered *0, 1,..., $\#_{Components}$-1*, and let the products be numbered *0, 1,..., $\#_{Products}$-1*. For each component and product, a certain amount of resources is required before it is completed. The resources are assigned from the pool of available resources, represented by a number, to components and products that need to be implemented, by means of a *decision*. A decision consists of two rows of numbers, one with length $\#_{Components}$ and one with length $\#_{Products}$. The numbers in the first row represent the amount of resources that are allocated to the corresponding component, and the numbers in the second row correspond to the amount of resources allocated to the corresponding product. In addition, we define a *production plan* to be a mapping from time points up to TimeHorizon-1 to decisions. For each production plan *ProdPlan*, *ProdPlan(i).Components[c]* denotes the resources allocated to component *c* at time *i* and *ProdPlan(i).Products[c]* denotes the resources allocated to product *c* at time *i*. The resources are assigned from the resource pool, which is represented by a number. The minimum amount of resources that can be allocated per decision is the block size. We define a *Decision Graph* as a means to represent production plans. In a decision graph, each node represents a *workstate*, which we define to be a time point and the remaining *workload*. A workload consists a row of numbers with length $\#_{Components}$ and a row of numbers with length $\#_{Products}$. Each of these numbers represents the amount of resources that should be allocated before the component or product is finished. Each arrow represents a decision that is taken from the particular workstate. The depth of the graph corresponds to the amount of time points in the time horizon. The workload of the new state is the result of deducting the assigned resources from the required resources in the current workload.

For example, consider the implementation of product *C* and product *D*. We need two components to assemble these products, components *A* and *B* respectively. Component A requires two weeks to complete and component B one week. Product C and D are refinements that are made on a per customer basis; Product C can only be implemented if component A is available, and Product D only when component B is available. For Product C as well as product D one person-week is needed for the implementation. We see that $\#_{Components} = 2$, $\#_{Products} = 2$. Additionally, consider the case that we have one employee and therefore can allocate one person-week per decision. We attain the decision graph that is depicted in Figure 5.3.

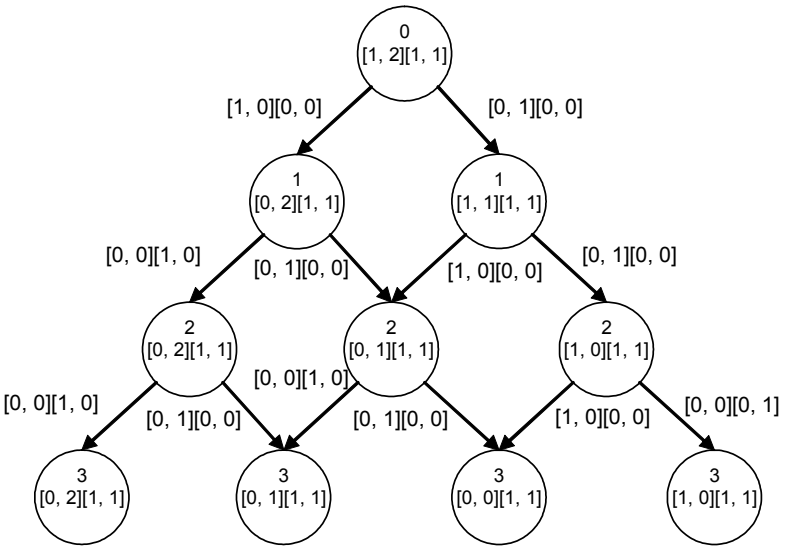


Figure 5.3 Decision Graph

Here, the notation [p, q][r, s] in a node denotes *p resources needed for A, q for B, r for C and s for D*. The notation [j, k][l, m] an arrow denotes the decision *j resources allocated to A, k to B, l to C and m to D*. In this decision graph each path from the root to a leaf node in this graph represents a production plan. In each node the workload is indicated. After a component has been completed, the corresponding number in the workload becomes zero. Products are provided to customers, which means that in subsequent states new instances can be created for new customers. Whenever a product is completed by a decision, in the next stage a new instance of this product can be implemented. If for the implementation of a product instance k resources are required and for the current instance another l resources are required, the allocation of m resources results in *(m+k-l) div k* product instances. After this decisions, i for the completion of another instance of the same product *k-(m+k-l) mod k* resources. The production plans that are represented by this decision graph are described in Table 5.2.

Table 5.2 Production Plans

ProdPlan(0)	ProdPlan(1)	ProdPlan(2)
[1, 0][0, 0]	[0, 0][1, 0]	[0, 0][1, 0]
[1, 0][0, 0]	[0, 0][1, 0]	[0, 1][0, 0]
[1, 0][0, 0]	[0, 1][0, 0]	[0, 0][1, 0]
[1, 0][0, 0]	[0, 1][0, 0]	[0, 1][0, 0]
[0, 1][0, 0]	[1, 0][0, 0]	[0, 0][1, 0]
[0, 1][0, 0]	[1, 0][0, 0]	[0, 1][0, 0]
[0, 1][0, 0]	[0, 1][0, 0]	[1, 0][0, 0]
[0, 1][0, 0]	[0, 1][0, 0]	[0, 0][0, 1]

In the columns of this table the possible resource assignments for each point in time are described. For our example this is the case for both Product A and Product B. In the table we have only listed the feasible production plans for our example case.

5.3.4 Integration of the Decision Graph and Scenario Graph for Resource Allocation Optimization

To come to scheduling advice that considers the probabilistic nature of the changes in the market demand, the information in the decision graph and the scenario graph needs to be considered simultaneously. To describe the combined information in a systematic manner, we propose to merge the two graphs into a combined graph. In a combined graph, each node has a workstate as well as a demand. For the computation of the scheduling advice, the expected profit of each production plan needs to be determined. Each arrow represents a decision that can be taken from the particular state.

For each decision the resulting state is defined by a set of possible new states, each of which will be reached with a specific probability. The nodes of the combined graph are given by the following relation:

> Given time *i*, for every decision graph node *(i , w)*, and every scenario graph node *(i, d)*, a node exists in the combined graph *(i, w, d)*.

So every node at time i in the scenario graph is combined with every node at time i in the decision graph. The arrows between the nodes in the combined graph are given by the following relation:

> There is an arrow with probability p from (i, w_1, d_1) to $(i+1, w_2, d_2)$ in the combined graph *if and only if* there is an arrow with probability p from (i, d_1) to $(i+1, d_2)$ in the scenario graph and there is an arrow from (i, w_1) to $(i+1, w_2)$ in the decision graph

The combined graph for our example is depicted in Figure 5.4.

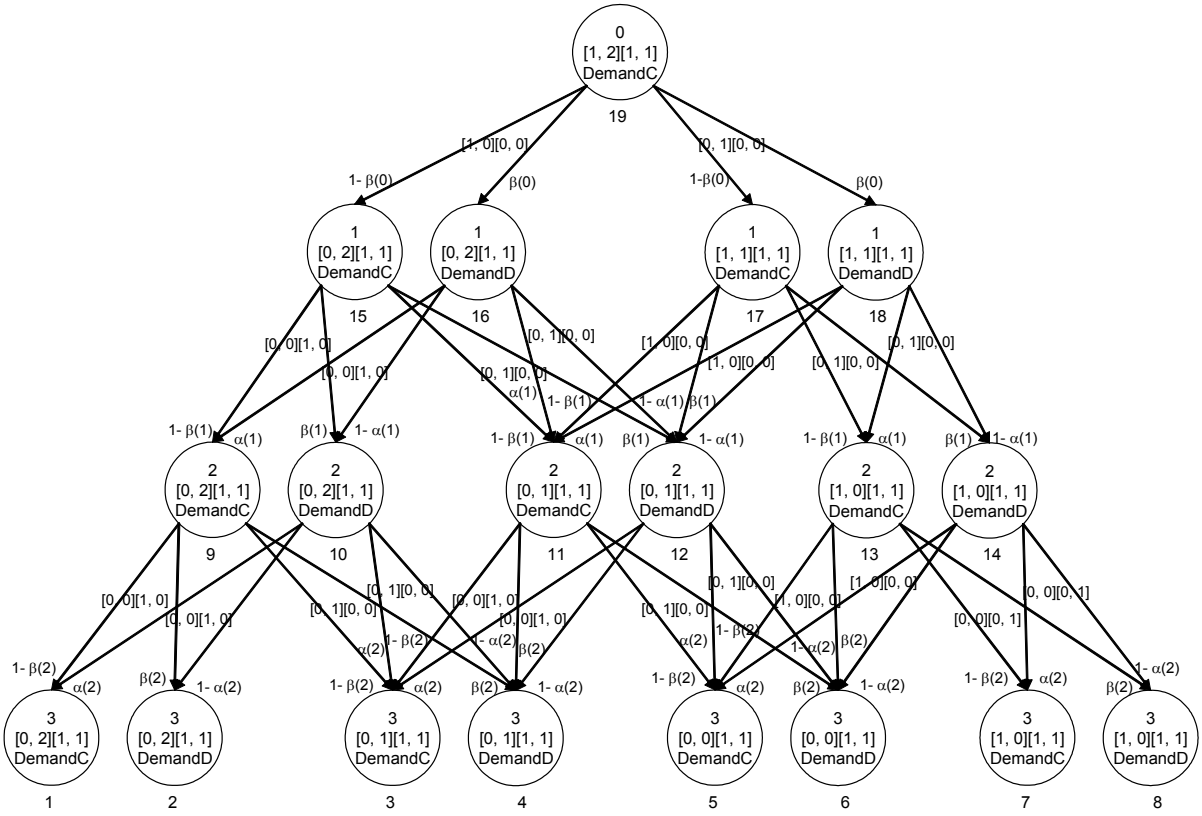


Figure 5.4 Combined Decision- and Scenariograph

The combined graph also is very similar to a probabilistic automaton, like the scenario graph. In this combined graph, each allocation decision is followed by a random change in the market demand, in accordance to a probability distribution. For instance, when in the start state decision [1, 0][0, 0] is taken, in the next point in time DemandC occurs with a probability of 1-β(0) and DemandD occurs with a probability β(0). It can be seen that for this example each state in the decision graph is replaced with two nodes, one for each demand state, since at every point in time one of two possible demands can occur. The process of a decision followed by the occurrence of a demand state describes the dynamic probabilistic market conditions during the implementation of the components.

Our approach returns scheduling advice, which means that the resulting schedule must be able to indicate the best scheduling decision (the advice) for each state in the combined graph (the condition). This is achieved by ordering the decisions in each state based on the *expected profit*. In accordance with market description, the direct profit for a decision D in state S corresponds to the *turnover* from the products that are completed by the decision minus the *cost* of the resources that are used. The turnover is the sum of the turnover per product. The direct profit of choosing decision D from state S is given by a reward function R:

$$R(S, D) = 0 \quad \text{, if } S \text{ is at the end point on the time horizon}$$

$$R(S, D) = \left(\sum_{P} Price(P) * Amount(P, D)\right) - (ResourceAmount(D) * CostPerResource)$$

Here Amount(P, D) is the number of products P that is produced as a result of decision D, and the summation runs over all products. Note that Amount(P) is not greater than the demand of P in S, since D is a feasible decision. As we have identified earlier, to compute the expected profit of a decision we have to consider the maximum expected profit of each possible subsequent state, multiplied by the probability of reaching it. For this purpose we define:

$$Val(S, D) = R(S, dD) + \sum_{S'} p(S' \mid S, D) MaxVal(S')$$

where

$$MaxVal(S) = \max_{D} Val(S, D)$$

When we examine the structure of the combined graph in Figure 5.4, we can see that for the evaluation the value of particular states are needed multiple times. For example, the value of state 11 is needed for the evaluation of states 15, 16, 17 and 18. Due to this structure we will to apply *dynamic programming* [Bellman1961] [Larson1968] in the evaluation of the states and their decisions. This essentially means that once a particular state has been evaluated with respect to all of its decisions, the result will be stored and reused every time it is needed for the evaluation of other states. We can now evaluate the function Val for our example using the functions defined above. The results of this computation are given in Table 5.3.

Table 5.3 State Evaluations of the Combined Graph

S	I: [1, 0][0, 0]	II: [0, 1][0, 0]	III: [0, 0][1, 0]	IV: [0, 0][0, 1]
9	-	-CostPerResource	Price(C) -CostPerResource	-
10	-	-CostPerResource	-CostPerResource	-
11	-	-CostPerResource	Price(C) -CostPerResource	-
12	-	-CostPerResource	-CostPerResource	-
13	-CostPerResource	-	-	-CostPerResource
14	-CostPerResource	-	-	Price(D) -CostPerResource
15	-	-CostPerResource + (1-β(1))*MaxVal(11) + (β(1))*MaxVal(12)	Price(C) -CostPerResource + (1-β(1))*MaxVal(9) + (β(1))*MaxVal(10)	-
16	-	-CostPerResource + (α(1))*MaxVal(11) + (1-α(1))*MaxVal(12)	Price(C) -CostPerResource + (α(1))*MaxVal(9) + (1-α(1))*MaxVal(10)	-
17	-CostPerResource + (1-β(1))*MaxVal(11) + (β(1))*MaxVal(12)	-CostPerResource + (1-β(1))*MaxVal(13) + (β(1))*MaxVal(14)	-	-
18	-CostPerResource + (α(1))*MaxVal(11) + (1-α(1))*MaxVal(12)	-CostPerResource + (α(1))*MaxVal(13) + (1-α(1))*MaxVal(14)	-	-
19	-CostPerResource + (1-β(0))*MaxVal(15) + (β(0))*MaxVal(16)	-CostPerResource + (1-β(0))*MaxVal(17) + (β(0))*MaxVal(18)	-	-

In this table, the leftmost column contains the number of the state in the combined graph, which corresponds to the numbers in Figure 5.4. The four main columns, indicated by the thick line, represent the four possible decisions that can be taken, *[1, 0][0, 0]*, *[0, 1][0, 0]*, *[0, 0][1, 0]* and *[0, 0][0, 1]*, corresponding to the decision in the decision graph. The table gives the values for Val(S, D). As indicated before, depending on the state it is not possible to take all four decisions. The decisions that are not feasible for a particular state are indicated with "-" in the table. Note that the MaxVal for each state is equal to the column with the highest Val value for that particular state. Also note that MaxVal for states 1, 2, ..., 8 is equal to 0, and these states are therefore omitted in this table.

When we examine, for example, the computation of Val for choosing action *[0, 0][1, 0]* from state 15, this value is computed as follows. According to the definition of R we sum up the turnover of all demanded products we can produce and subtract the cost for the used resources. We produce one instance of product C with our decision, which results in a turnover since C is demanded in this state. From this turnover we deduct the cost for producing C. This results in a value *Price(C)-CostPerResource*. To compute the value for Val, we now include the expected maximum profits for all states that can be reached from this state by choosing this decision. We reach state 9 with a probability 1-β(2) and we reach state 10 with a probability of β(2). The value of Val for choosing *[0, 0][1, 0]* from state 15 thus becomes *Price(C)-CostPerResource+(1-β(1))MaxVal(9)+(β(1))MaxVal(10)*.

5.4 Case Study: The Insurance Products Framework

5.4.1 The Insurance Products Framework

In section 5.3.1 we have introduced a software company that produces and maintains software systems to be used by insurance companies. To deal with the variety and complexity of producing insurance products for a variety of companies, this software company has captured generic functionality in an application framework. For the design of this application framework the software engineers have performed an analysis of the insurance domain. During this domain analysis, the typical elements of insurance products are identified and related by means of the feature diagram depicted in Figure 5.5.

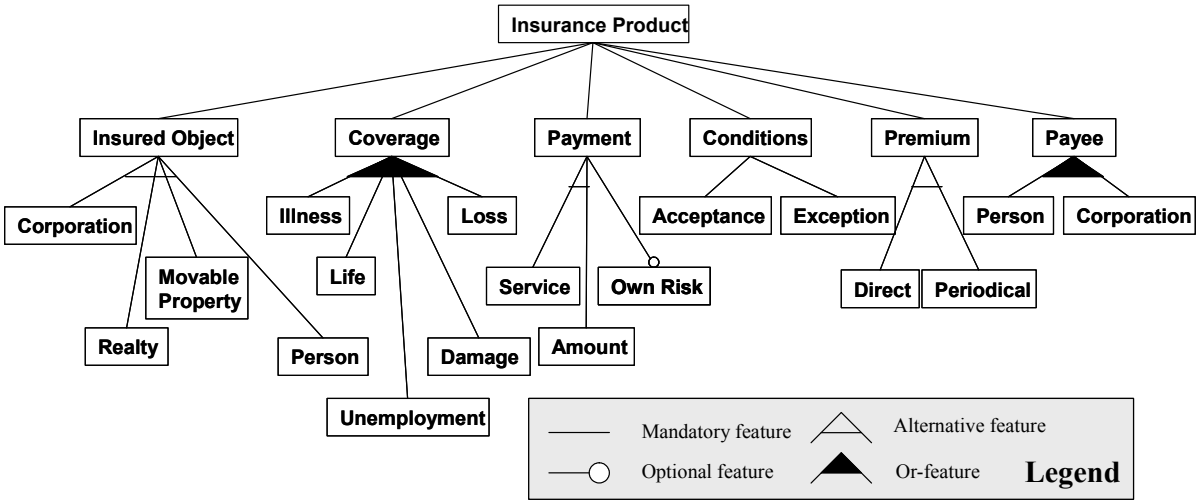


Figure 5.5 Feature Diagram of Insurance Products

In this figure, a feature diagram of insurance products is depicted. A feature diagram represents both the commonality and variability of a product [Kang1990]. In Figure 5.5, six different parts of insurance products are shown: insured object, coverage, payment, conditions, premium and payee. Each of these parts can be represented by several entities. For example, an insured object can be a person, corporation, realty or a movable property. The symbols that are depicted in the legend are used as restrictions on the variability. The feature diagram distinguishes four types of relations: mandatory features, optional features, alternative features and or features. The difference between alternative and or features is that the alternative relation is an exclusive or (only one of the involved features can be included), where any subset of features can be selected within an or relation. Using these specifications, a large variety of insurance products can be defined, ranging from bicycle insurances to tailored insurances for large corporations. In this chapter, to avoid confusion, products that are sold by software and insurance companies are referred to as insurance software systems and products, respectively. In order to deal with the developing market, the software company has decided to design and implement an application framework, from which it can easily derive product variations. The application framework is based on the feature diagram in Figure 5.5, and is divided into three layers. In Figure 5.6 the components and their dependencies of the application framework are depicted.

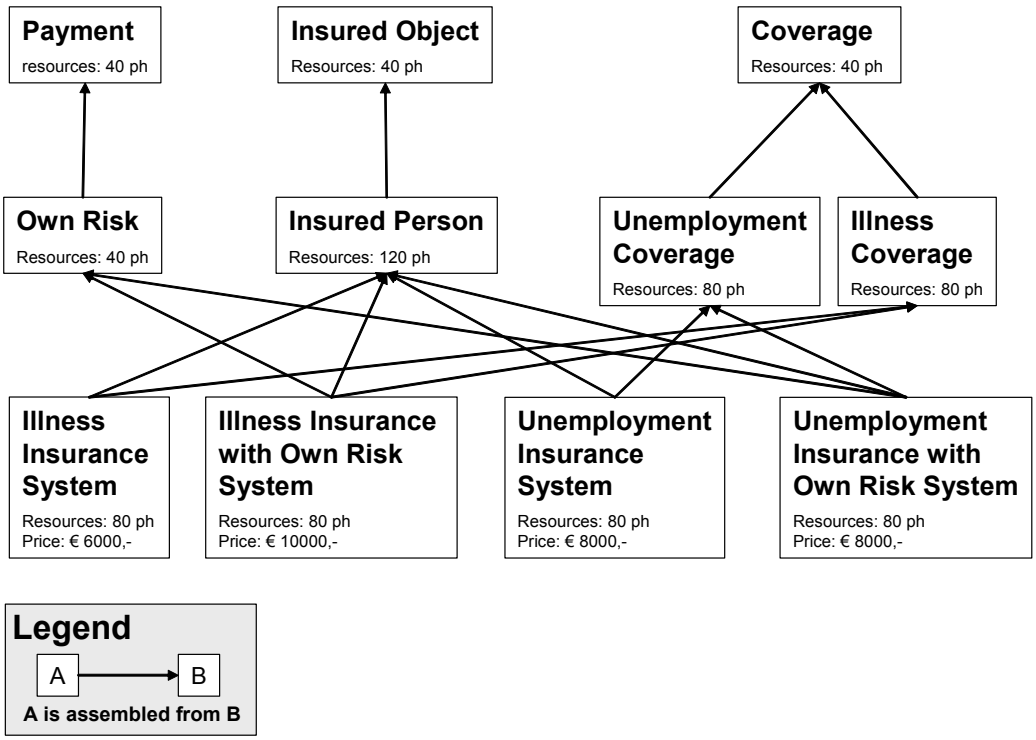


Figure 5.6 Insurance Application Framework Components

At the top level the most basic components are displayed, which correspond to the basic features in the feature diagram. In the middle layer, the components are depicted that will be used to assemble final products. These components are refinements of the base components. Finally, at the bottom the products are depicted, that the software company aims to derive from the framework components. In the figure, also the needed resources for the implementation of the components is indicated. For products, the implementation time for a single instance as well as the selling price is indicated.

5.4.2 Modeling the Market Demands and Production Plans

The implementation and transfer to the application framework takes time, and as a result the software company might miss opportunities for selling their insurance systems. Therefore the company decides to determine the optimal implementation schedule with respect to the expectations of the market demands. For this purpose the software company has analyzed the market expectations for the next six weeks, and based on this analysis three mutually exclusive market expectations have been identified: an *optimistic* outlook, a *moderate* outlook and a *pessimistic* outlook.

Optimistic Outlook

In the optimistic outlook there will be demand for a variety of products by many insurance companies. This increase in demand is based on the possibility of new legislative decisions with respect to illness insurances. In the optimistic outlook, ten insurance companies will demand:

- *Illness Insurance System*
- *Illness Insurance with Own Risk System*

Moderate Outlook

The moderate outlook predicts that insurance companies want to upgrade their service to their customers by offering own risk options in their insurance products. In the moderate outlook, four insurance companies will demand:

- *Illness Insurance with Own Risk System*
- *Unemployment Insurance with Own Risk System*

Pessimistic Outlook

In the pessimistic outlook the pending legislative decisions with respect to illness insurances causes the insurance companies to postpone their demand for illness insurance systems. As a result only two insurance companies will order unemployment insurance products:

- *Unemployment Insurance System*
- *Unemployment Insurance with Own Risk System*

The probability of going from an optimistic to a pessimistic outlook or vice versa is comparably small. The probabilistic changes in the market demand are modeled using the scenario-graph in Figure 5.7 and are based on the defined outlooks.

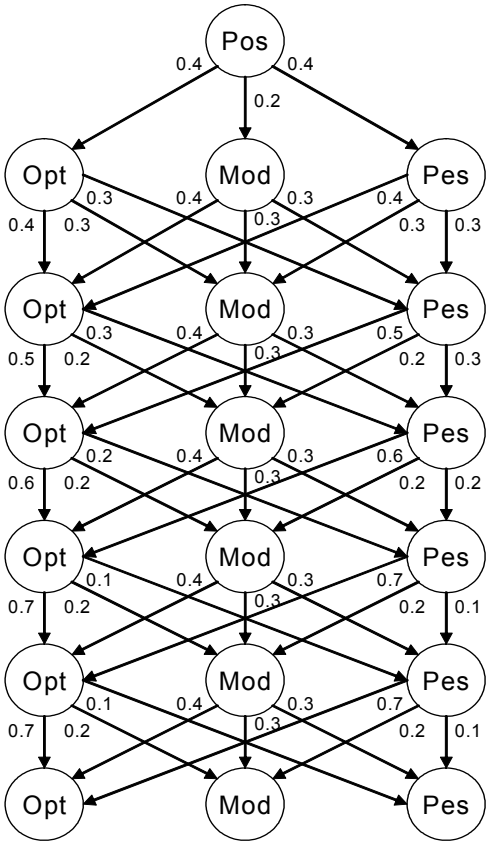


Figure 5.7 Scenario-graph for Insurance Market Estimations

In this scenario-graph, for each week three new states can occur, the optimistic outlook, the moderate outlook and the pessimistic outlook. For the period in which the application framework is supposed to be implemented, this scenario-graph describes all the possible scenarios that can occur. Each path from the start state to a leaf state represents one possible scenario.

The next step in our approach is to model the possible production plans that can be chosen for the implementation of the application framework. The software company at this time employs five software engineers, who can be scheduled on a per week basis for 1 personweek (which equals 40 person-hours). The hourly wage of the software engineers is € 12. The software company has the choice to allocate software engineers to the project where only the allocated resources need to be paid from the perspective of the project. The resource pool is therefore 200 and the block size is equal to 40. Based on this information and the framework components and their dependencies, the production plans can be derived using the definitions for the decision-graph. Due to the size of the decision-graph, a graphical depiction is omitted here.

5.4.3 Determining the Scheduling Advice

The final step is to determine scheduling advice with respect to the optimal production plan for the implementation of the application framework. This is achieved by merging the scenario-graph and the decision-graph into the combined graph according the definitions in section 5.3.4. Finally, for each state in this combined graph the best decision is computed and stored in the combined graph together with the expected profit of this decision. The relevant part of the combined graph now is a *scheduling advice*, which means that it contains the best scheduling for each situation that can occur according to the modeled inputs. The scheduling advice can be queried for the best decision by indicating the current state, being the amount of work done and the current market situation. The complete scheduling advice for our example is too large to include, so we examine a small portion of the scheduling advice. In the portion we examine, the first two weeks have passed, and in these two weeks, five components have been completely implemented: *Payment, Insured Object, Coverage, Insured Person* and *Illness Coverage*. In the third week, the project manager is faced with the question how to allocate resources in the third week.

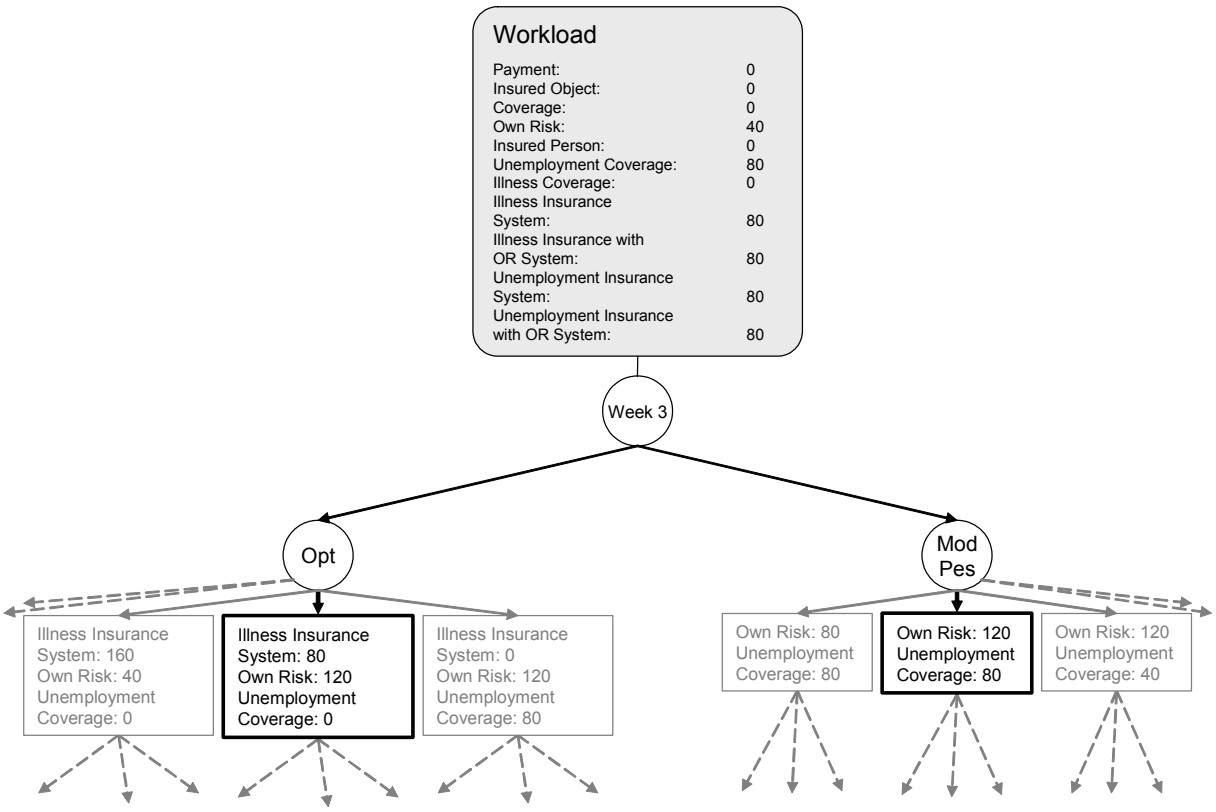


Figure 5.8 Portion of the Scheduling Advice

In Figure 5.8, we see the workload that remains at the beginning of week three in the grey box. The best resource allocation decision for this week depends on the market demand that has occurred during this week. On the left side of the picture, the best decision is given in case of the optimistic demand outlook. In the picture, three of the possible decision are depicted including the best decision, indicated with thick lines. Note that for example the option of allocating 200 person-hours to produce Illness Insurance Systems for 4 companies is ignored, since this option in the short run would have a higher profit but over the period of six weeks has a greater risk. On the right, the best decision is given for either the moderate or pessimistic demand outlook. Note also, that it is possible to have a best decision where no resources are allocated at all. This is the case, when the costs of the current decision will not be compensated by expected profit within the time horizon.

5.4.4 Relevance and Validity of Scheduling Advice

The scheduling advice, that results from the application of our approach, can be used by project managers to coordinate resource allocation. In the complete production plan the best decisions are known for *each* state, which means that even when the project manager does not select the best decision, it is possible to give allocation advice. Only when the initial inputs change, such as changes in the scenario-graph or the time horizon of interest, the scheduling advice must be recalculated. While the scheduling advice offers allocation advice for all the modeled demand scenarios, it should be noted that this advice is based on the expectation values of the probabilities. The consequence of this is that the validity of the decision support depends on the accuracy of the market demand scenarios. Only when the probabilities and demand states give an accurate description of the future changes in market demands, the decision will give an accurate result. Therefore, while an accurate representation of the future mar-

ket demands leads to useful scheduling advice, these restrictions need to be considered during the allocation of resources.

5.5 Related Work

5.5.1 Software process configuration management

To cope with the constantly changing customer requirements, software products and software processes must be re-configured frequently. Software configuration management aims to manage the evolution of a product. Our approach can be seen as complementary to software configuration management techniques. Software configuration management is a broad domain and covers the entire life cycle. Further, software configuration management processes can be defined for specific domains as well. An approach for resource management for distributed multiagent systems is presented in [Podorozhny1999]. By taking into account a wide range of resource and entity types, the resource management can be described and optimized accurately. This approach, however, is not equipped to deal with uncertainty in the changes of the market. Our approach aims to schedule the available resources while considering probabilistic changes in market demands and project context. In addition to software configuration management, several researchers have focused on the configuration of processes. The basic issues of process configuration are located in the choice of the development process and the alignment of the process configuration. In [Osterweil1997] [Osterweil1998] it is argued that software development processes can be managed and configured in much the same way as software products.The common assumption of this approach is that processes should be considered as products and be configured with respect to product quality goals. All these approaches focus on specific aspects of software configuration management, to which our approach is complementary. Our approach focuses on the support of uncertain changes in market demands with respect to the allocation of implementation resources, which is generally missing in the conventional software process configuration approaches.

5.5.2 Requirements Engineering

Scheduling requirements with respect to the order in which they should be implemented is not new. There have been various research activities on this topic. However, research activities generally focus on fixed requirements; prioritization is realized after the requirements are determined. The work described in [Regnell1992] focuses on prioritization of software requirements with respect to the quality of decision-making. The basic assumption here is that new requirements should either be accepted or rejected. By analyzing and simulating the acceptance/rejectance rate, the decision quality can be improved and only relevant requirements are selected for further consideration. The difference with our approach lies in the fact that the approach in [Regnell1992] acts as a filter and results in a set of requirements that should be implemented. The remaining relevant requirements are also considered equally important. However, these requirements still need to be prioritized with respect to implementation and resources, since there still might be dependencies among them. Also, the approach focuses on the best requirements set for the next release, while our approach is more aimed at finding optimal schedules for implementation with respect to changes along the time span of a project. In [Karlsson1997] a methodology is proposed, which prioritizes requirements based on an analysis of the costs and the profits. For each requirement, the relative cost and value are determined using the analytic hierarchy process. This model then allows comparison of the requirements based on these properties, so that one can be ranked over the other with respect to these two aspects. By considering the requirements of engineers, customers, users and software engineers, an accurate requirements model can be made. The proposed methodology in [Karlsson1997] focuses on prioritisation with respect to value and cost, and with that it aims to

satisfy market demands in general. The methodology does not explicitly address the delivery constraints that are imposed during the software development process. Also the model does not support analysis of requirements that share dependencies. In our approach this is specifically addressed by the framework dependencies.

5.5.3 Optimization Models

Many different optimization models and approaches have been defined to address specific problems in the area of design, most of them having a mathematical origin. Generally speaking an optimization model consists of four parts: a subject to be optimized, the options to be considered, a comparison criterion and an ideal situation description (or goal). The possible options are evaluated with respect to their goal using the criterion. For instance, there are learning-based optimization models such as neural networks [Haykin1998] or genetic algorithms. The problem addressed in this chapter falls into the area of *aggregate production planning*, a particular type of problem that falls into the category of supply chain management, based on models from operations research. For the interested reader a more elaborate introduction into aggregate production planning and scheduling can be found in [Voß2003]. In this book an overview is given of the components of supply chain management and a number of models are introduced and discussed for production planning and scheduling. The book also proposes a number of extensions for optimization beyond project boundaries and support for tardiness.

5.6 Discussion

We have presented a method that can help project managers in determining the estimated optimum process development schedule that delivers the best expected profit. Given probabilistic product demand scenarios, resources, application framework structure and components dependencies, the demand scenarios and production plans can be derived and evaluated. This method consists of the following steps from the project manager point-of-view:

1. *Define the market demand states and the event probabilities*
2. *Define the components to be implemented and their dependencies*
3. *Define the available resources and the cost per resource*
4. *Define the time horizon and the reward specification*

After these four steps have been completed, our approach derives the scenario graph and decision graph, and combines them. Based on the provided inputs the scheduling advice is computed. In the following, we evaluate our approach with respect to the following concerns:

When are the results from the resource allocation approach accurate?

When the implementation of application frameworks and product lines should be planned, there is always the possibility of missing out on potential sales during the implementation period. In the case that the changes in demand are known for this period, planning the implementation activities is a straightforward activity. However, as has been identified in this chapter, the demands coming from the market are not known precisely, which means the project manager is faced with imperfect information. The presented approach describes the imperfection by means of events that have a probability description with respect to their occurrence. In general, short-term market estimations can be made with reasonable accuracy. For prolonged periods of time simple probabilistic estimations will become less reliable. In addition, it can be

difficult to identify the appropriate probability definitions, which is a reason to expand the capabilities of the model with respect to working with other imperfection models.

The proposed approach can be used at various levels of detail, depending on elements of interest to the project manager. In particular, the scheduling advice offers advice for all the states that can be derived from the provided information. This means that it is only necessary to recalculate the optimization when one of these inputs is no longer valid. In addition the production plan can be used to gain extra insights in, for instance, worst-case scenarios by means of analyzing 'what-if' situations. This helps the managers to gain better insight in the planning and risks of software development processes under imperfect information.

Will the proposed resource allocation approach scale to industrial sized problems?

To calculate the optimal result, the defined approach computes and evaluates all possible states that can be derived from the inputs. This can cause very large state spaces and lengthy computations. This is known as the curse of dimensionality [Bellman1961]. To address this problem, within the field of optimization theory various research activities have been carried out. Examples are state space minimization techniques by assessing the relevancy of states and by making trade-offs between accuracy of the result and computation size, or optimization for parallel computing [Chung1992] [Larson1965] [Larson1968]. The presented approach provides a valuable input to the resource planning process in which the scarce resources have to be planned in a time span. However, for industry-sized projects the approach introduces considerable computational effort.

To resolve this, tool support is needed that can be used to perform the computations in the specific context of the company. We have developed a prototype of tooling that is usable within an industrial context, which is described in chapter 6. In addition to the described approach, the tool provides flexibility in considering new data, for example, which may be available due to better market estimates. The tool recalculates the optimal schedule from the time that new data are considered relevant. In addition, the tooling can provide scheduling advice by providing a querying mechanism for the scheduling advice.

5.7 Conclusions

In this chapter the activity of scheduling the implementation of application frameworks and product lines is identified as an area that suffers from the existence of imperfect information. Typically, the implementation of the reusable parts of this type of systems requires considerable resources, which means that it is not possible to implement products that generate a profit simultaneously. Ideally, the implementation schedule should allow for the commitment to the production of products at the moment a considerable demand comes from the market. However, since it is not exactly known what the demand will be in the future, project managers should schedule the implementation trajectory based on imperfect information. In addition, the scheduling activity is complicated further by the complex dependency structures that can exist between reusable framework parts and the products that can be derived from them.

We presented a model capable of working with imperfect estimations of changes in demands coming from the market and determining scheduling advice for the implementation of product lines and application frameworks. The imperfect nature of the future market demands is addressed by identifying a number of events or outlooks that can occur during the period that implementation will be performed. Each of these events is attributed with a probability description, that indicates the probability that this particular event will occur during the implementation trajectory. In order to restrict the number of allocation possibilities over the period in which the application framework is implemented, the presented approach explicitly considers the dependency structure of the reusable assets and the derived products.

The application of the approach results in a production plan, which contains advice for all the possible production states and market demands that can be derived from the provided inputs. Project managers can query the production plan to determine the best allocation decision for the current state of the implementation trajectory. Additionally, it is possibly to use the production plan to explore situations that are not likely to occur, but can hold significant risk. Using the result of our approach in this manner gives project managers useful insights into worst-case situations and "what-if" scenarios. With this approach it is possible to explicitly consider probabilistic changes in the market demand while scheduling the implementation of complex software systems. The underlying optimization model ensures that the decision making process based on this information considers the probabilistic changes and their consequential risks accordingly.

Future research activities concern further support for the application in different settings with respect to product lines and application frameworks. In particular, the scheduling of reverse-engineering and adjustments to existing product line architectures is a logical extension of the approach presented in this chapter. In addition, automated support is required for convenient usage of the approach when scheduling software projects. This is in particular true for the computation of the advice, easy navigation of the scheduling advice, and using the production plan as a means to simulate alternate market and production situations. In chapter 6 we present tooling support for the approach proposed in this chapter. These tools assist the software engineers and project managers in the application of the approach.

CHAPTER

6

"Tensions are oddly distributed here tonight, Jessica thought. There's too much going on of which I'm not aware. I'll have to develop new information sources."

- From Frank Herbert's Dune [Herbert2005]

TOOL SUPPORT FOR IMPERFECT INFORMATION

6.1 Introduction

In this thesis we have identified that imperfection is an integral element of software design processes, which in most cases can not be avoided or resolved. As a result imperfection information must be managed inside the software design process, so that the software designers are aware of this imperfection and can use it to their advantage. To facilitate the inclusion of imperfect information in software development activities, in the previous chapters we have defined three optimization approaches, which support software engineers in a variety of design decisions. We have also identified that the manual application of these models can become too cumbersome to manage in an industrial context, which makes automated tool support indispensible. As a proof-of-concept we have developed a set of tools that implement our approaches. The toolset consists of three parts, the *Artifact Tracer*, the *Design Decision Tracer* and the *Resource Allocation Optimizer*, that offer support for capturing the relevant (imperfect) information during various activities of the design process, and, in addition, can apply the optimization approaches for decision support.

In this chapter, we introduce the tools that have been developed. The remainder of this chapter is as follows: In section 6.2 we describe the workflow and architectures of the individual tools. In section 6.3 we identify a number of attention points for the tooling and implementation of the models and we conclude the chapter in section 6.4.

6.2 The SPOT Toolset

To support software engineers in the application and use of our imperfection models, we have implemented the SPOT (Software Product & Process Optimization Techniques) Toolset on the Java^{tm} platform. In this toolset, three different tools are provided which correspond to the approaches proposed in this thesis:

- **Artifact Tracer Tool:** implements the approach based on the Artifact Trace Model that was presented in chapter 3. The tool assists the software engineer during the refinement steps and performs the optimization steps for the trade-off between stakeholder interests and implementation effort. The tool offers support for fuzzy functional requirement specifications.
- **Decision Tracer Tool:** traces the design decisions and contemplated alternatives during the software development process according to the design tree approach as described in chapter 4. Decision support is offered based on the design strategies with support for imperfect quality requirements and estimations.
- **Resource Allocation Optimizer Tool:** calculates scheduling advice for the allocation of resources based on the approach of chapter 5. The tool offers scenario simulation and allocation support based on probabilistic market demand specifications.

In this chapter we introduce the tools of this toolset. We introduce the design and global architecture of each tool. In addition we give a short overview of the user interface functions. It is important to note that this chapter will not contain an exhaustive description of the toolset, but gives a general notion on how the proposed approaches are supported by tool prototypes. A complete background on the tool design and user manuals can be found at [SPOT2007], from where the tools can also be downloaded.

6.2.1 The Artifact Tracer Tool

The Artifact Tracer Tool implements the artifact trace approach with support for imperfect requirement specifications as described in chapter 3. For this model the initial (imperfect) requirement specification is provided by the stakeholder. In subsequent steps the software engineer refines these requirements to a system architecture. The general design of the Artifact Tracer Tool is shown in Figure 6.1. In this abstract description the models and processes are represented as rectangles and ellipses, respectively. Each of the roles indicated at the top of the diagram provide or receives information from the tool.

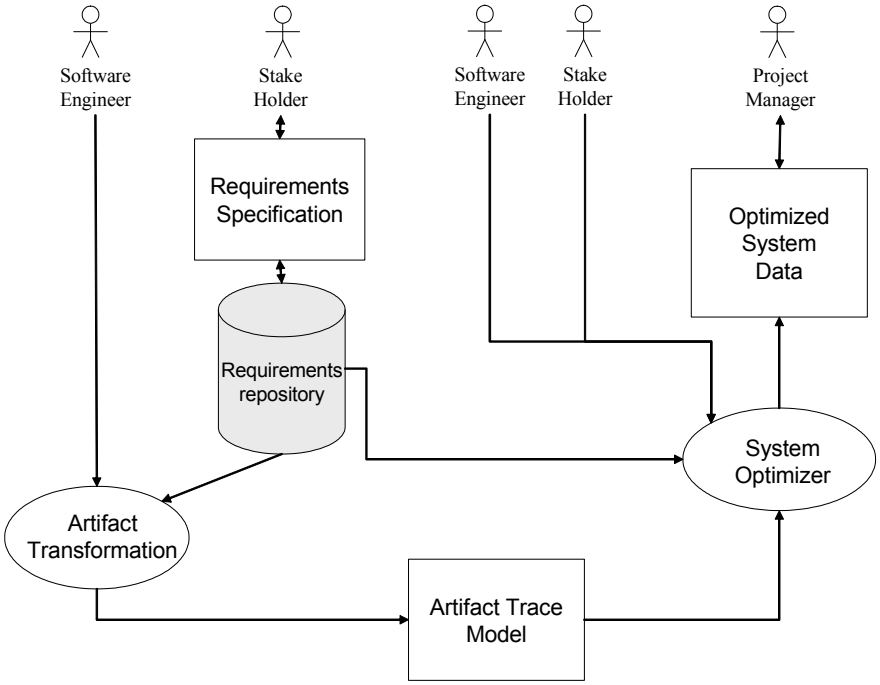


Figure 6.1 Artifact Tracer Tool Conceptual Design

The Artifact Tracer Tool requires the stakeholder(s) to provide the requirements specification for the software system that should be designed. These requirements are stored in the requirements repository, together with the annotations for stakeholder interests, such as relevance or urgency (for more information see chapter 3). The process *Artifact Transformation* takes the requirements from the requirements repository as input and together with the software engineer refines the requirements to components. The resulting data of this refinement process is the *Artifact Trace Model*. In the final step the *System Optimizer* process determines the optimal system design based on the Artifact Trace Model, for which the tool requires the optimization criterion and restrictions from the stakeholder and the software engineer. The resulting data *Optimized System Data* is then presented to the project manager.

Architecture and User Interface of the Artifact Tracer Tool

The architectural design of the Artifact Tracer Tool is divided into two areas, the components for the Artifact Trace Model and the optimizer. An abstract view of this architecture is given in Figure 6.2.

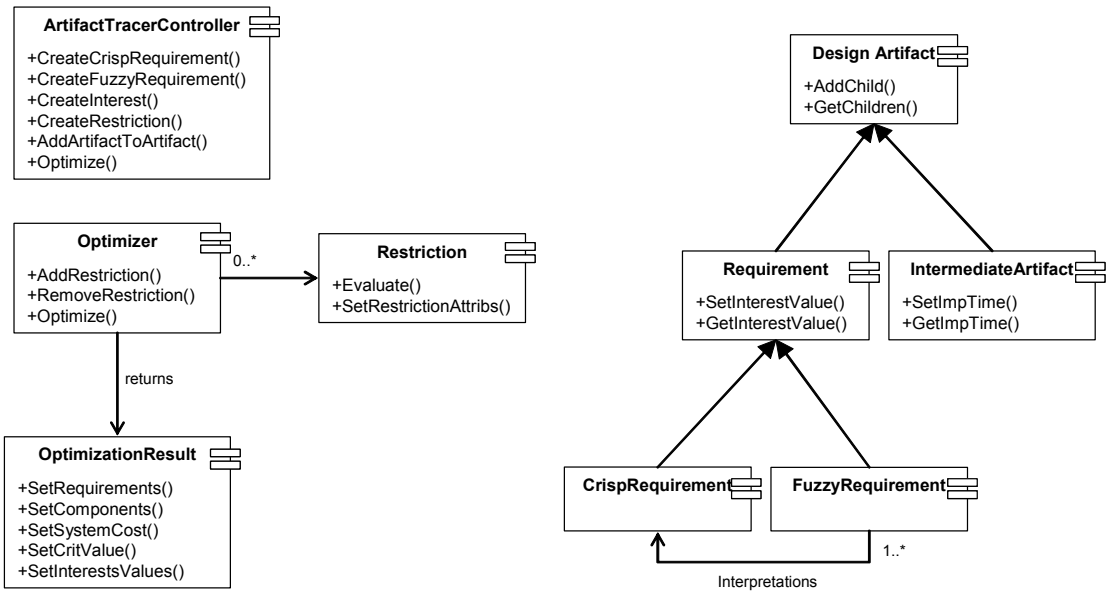


Figure 6.2 Abstract Architecture of the Artifact Tracer Tool

In this figure the righthand side depicts the components for the Artifact Trace Model. It can be seen that both crisp and fuzzy requirements are considered to be design artifacts, just as other intermediate artifacts. Additionally, crisp requirements are used to represent alternative interpretations of fuzzy requirements. The *Optimizer* component offers the possibility to define a number of restrictions on the allowed values of stakeholder attributes. The Optimize() function determines the optimal set of requirements to be implemented, which is achieved by evaluating each set of requirements using the instance of the Artifact Trace. The tool is designed according to the Model-View-Controller pattern, which means the Artifact Trace is instantiated by the controller based on the inputs from the user interface.

The Artifact Tracer Tool contains two algorithms to optimize system designs based on stakeholder interests. The first algorithm is a straightforward implementation, which evaluates every subset of the fuzzy requirement specification according the approach defined in chapter 3. In the same chapter we have defined a heuristic optimization approach, which is also supported by the tool. In accordance with this definition the heuristic algorithm evaluates a limited amount of subsets of the fuzzy requirement specification, in order to reduce the computational complexity. For the examples defined in this thesis the computational complexity is well within the capabilities of the Artifact Tracer Tool for both the full and the heuristic optimization. The necessity for the heuristic optimization should become clear from application of the tool within an industrial setting.

The user interface is the means of interaction for the software engineer with the tracing and optimization functionality with the Artifact Tracer Tool. The main interface of the Artifact Tracer Tool is depicted in Figure 6.3.

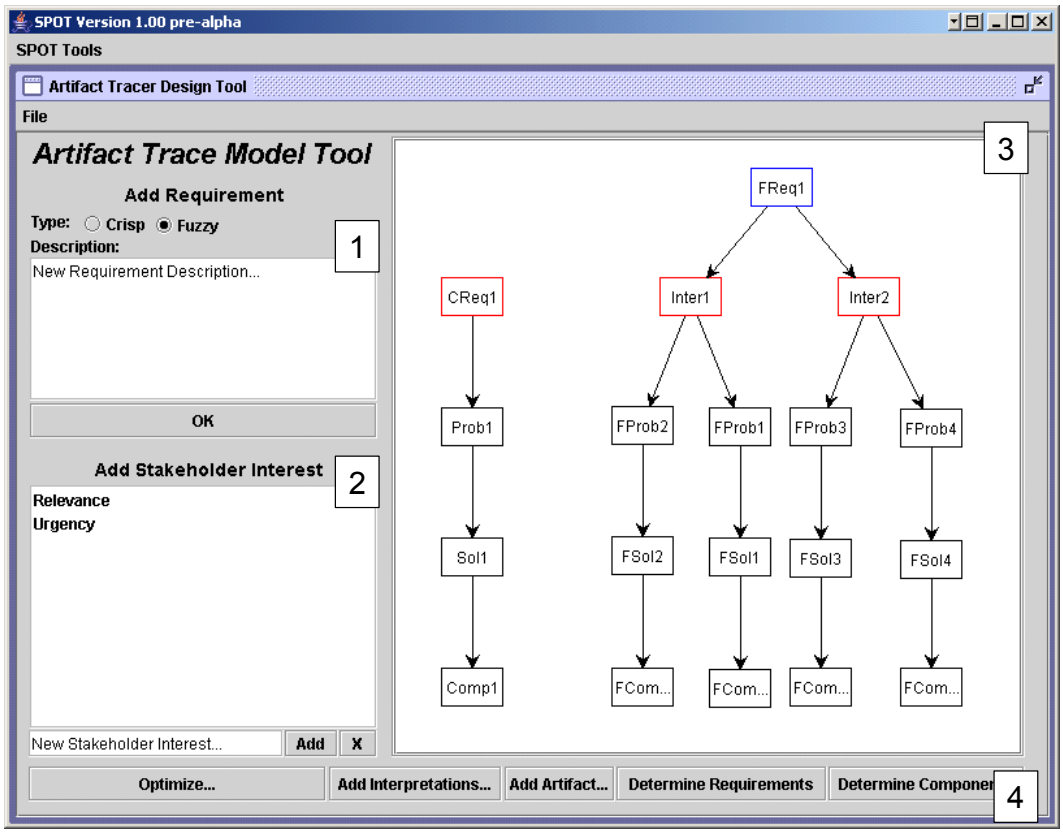


Figure 6.3 The Artifact Tracer Tool Interface

The main window of the Artifact Tracer Tool consists of four parts: the Requirement Modeler (1), the Stakeholder Interest Modeler (2), the current Artifact Trace (3) and the action-buttons (4). In the *Requirement Modeler* the stakeholder and software engineer can define both fuzzy and crisp requirements by means of textual descriptions. The new requirements are depicted in the Artifact Trace. In the *Stakeholder Interest Modeler* the applicable interests of the stakeholders are defined by textual descriptions as indicated in Figure 6.3. The values for each stakeholder interest can be set by means of the dialog window for the definition of interpretations. In the *Artifact Trace* sub-window the current state of the artifact trace is depicted. At the top of the trace the requirements are depicted, red for crisp requirements and blue for fuzzy requirements. The interpretations of fuzzy requirements and refinements that have been defined by use of the action buttons are reflected in this artifact trace window. The *Action Buttons* are used to perform design activities such as the refinement of artifacts or the definition of interpretations for fuzzy requirements.

The Artifact Tracer Tool supports the optimization of software design according to the approach that has been defined in chapter 3. Based on the requirements, stakeholders interests and artifact refinements, the software engineer can search for the optimal system based on self-defined criteria and restrictions. The resulting optimal system is presented with a result dialog such as the one depicted in Figure 6.4.

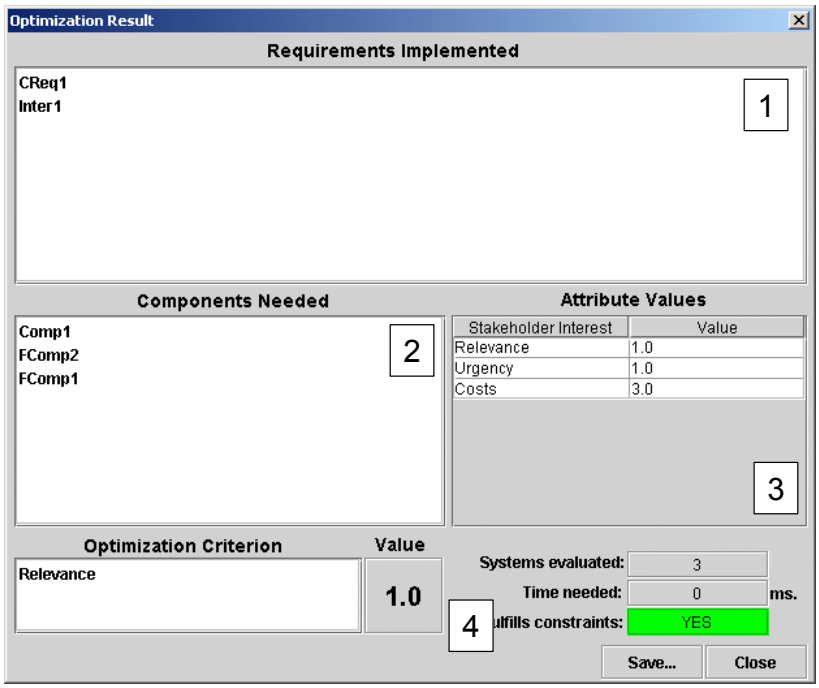


Figure 6.4 Optimization Result Dialog

In this result dialog window, the result of the optimization is depicted, which describes the system properties that result from the optimization process. It displays the system that best suits the optimization criterion and restrictions by the requirements that are fulfilled (1), the components that are needed for these requirements (2), and the value of the stakeholder interests (3) that are computed according to the definitions in chapter 3. In addition, it displays the value of the optimization criterion, the number of systems that have been evaluated and the time that was needed for the evaluation of these systems (4). Finally, it indicates whether this optimal result satisfies all the constraints that have been specified on the stakeholder interests. The optimization result can be saved in a plain text representation.

6.2.2 Decision Tracer Tool

The Decision Tracer Tool implements the Design Tree approach with support for imperfection in quality requirements and estimations as described in chapter 4. In this tool the stakeholder provides the (imperfect) quality requirements, and the software engineer identifies the design issues that need to be resolved. The software engineer then provides and evaluates design alternatives and, based on the derived design tree, the tool provides the optimal design state from which to continue. The architecture of the Design Decision Tracer is shown in Figure 6.5.

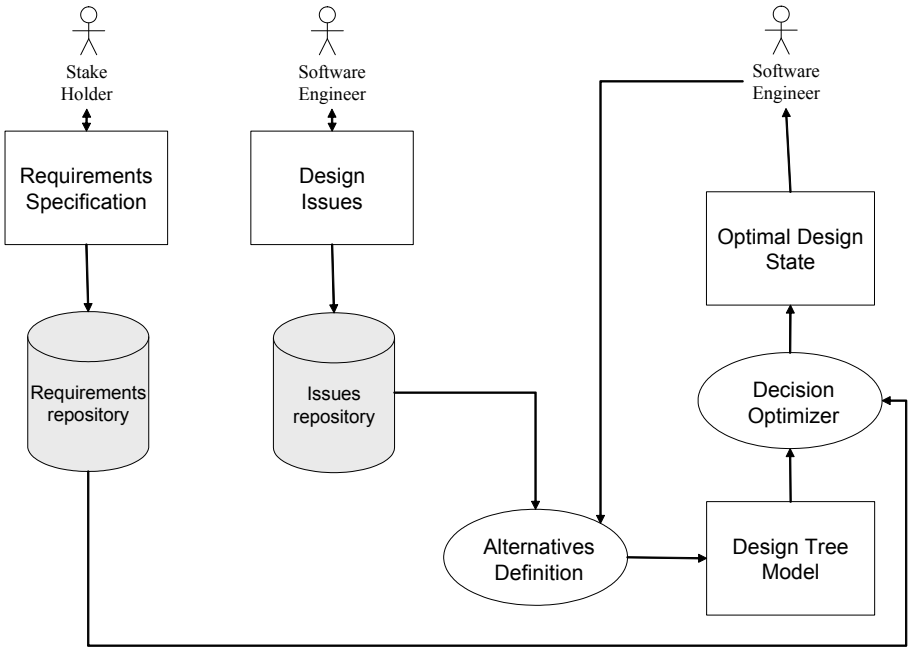


Figure 6.5 Conceptual Design of the Decision Tracer Tool

The Design Decision Tracer requires that the stakeholder provides the initial quality requirements specification for the system that must be designed. The software engineer is supposed provide the design issues that must be resolved. This information is stored in the *Requirements Repository* and *Issues Repository* respectively.The second step is the identification of design alternatives for each individual design issue. The *Alternatives Definition* process takes the issues from the Issues Repository as input and, with the help of the software engineer, identifies the alternatives for the current design issue, and estimates their respective quality attributes. The resulting data of this process is the *Design Tree Model* data. The *Decision Optimizer* process takes the current design tree as input and determines the best design state from which to continue the design process. The *Optimal Design State* data result is presented to the software engineer, who now can continue determine alternative solutions for the next design issue and repeat the optimization process.

Architecture and User Interface of the Decision Tracer Tool

The Decision Tracer Tool requires the non-functional requirement specification from the stakeholders as well as the design issues that need to be resolved. Based on these inputs the design decision are traced and optimized. As a result, the architecture of Decision Tracer Tool reflects these two elements, as can be seen in Figure 6.6.

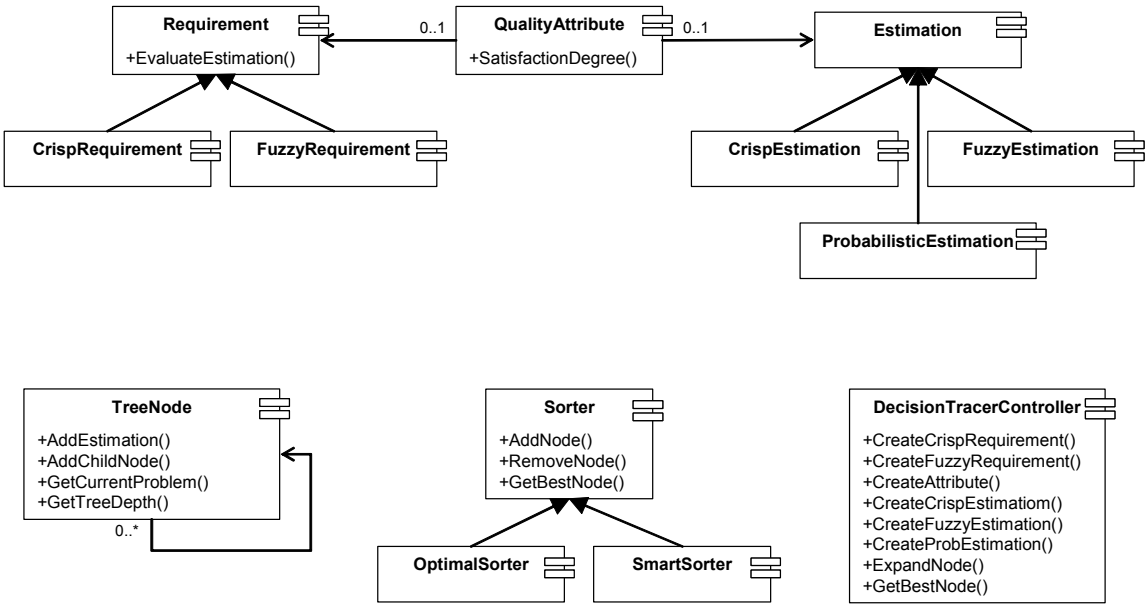


Figure 6.6 Abstract Architecture of the Decision Tracer Tool

In the top half of this figure, the components for modeling quality requirements and estimations is depicted. It can be seen that requirements and estimations are seen as attributes of a quality attribute component. The function *SatisfactionDegree()* returns the degree of satisfaction for its associated requirement and estimation according to the operators defined in chapter 4. The *TreeNode* and *Sorter* components implement the functionality for the Design Tree approach. The design tree is built by linking a number of *TreeNode* instantiations, each of which contains estimation for the defined quality attributes. Every time a new node is added, the *Sorter* component implements the sorting functionality that is described in section 4.2.4. It updates the ranking of the nodes and removes nodes when necessary. This activity is performed at each definition of new tree nodes, which means that the computational complexity of the optimization is neglecible.

The user interface of the Decision Tracer is comprised of three user tabs, the *Process Parameters Tab*, the *Designer Tab* and the *Design Tree Tab*. In the Process Parameters Tab the quality requirements for the software design process are defined, as well as the initial design issues that are to be resolved. In the Designer Tab in the predefined order the design issues are resolved in a step-wise manner. The Design Tree Tab can be used to inspect the design tree that results from the identified alternatives and the decisions that are taken.

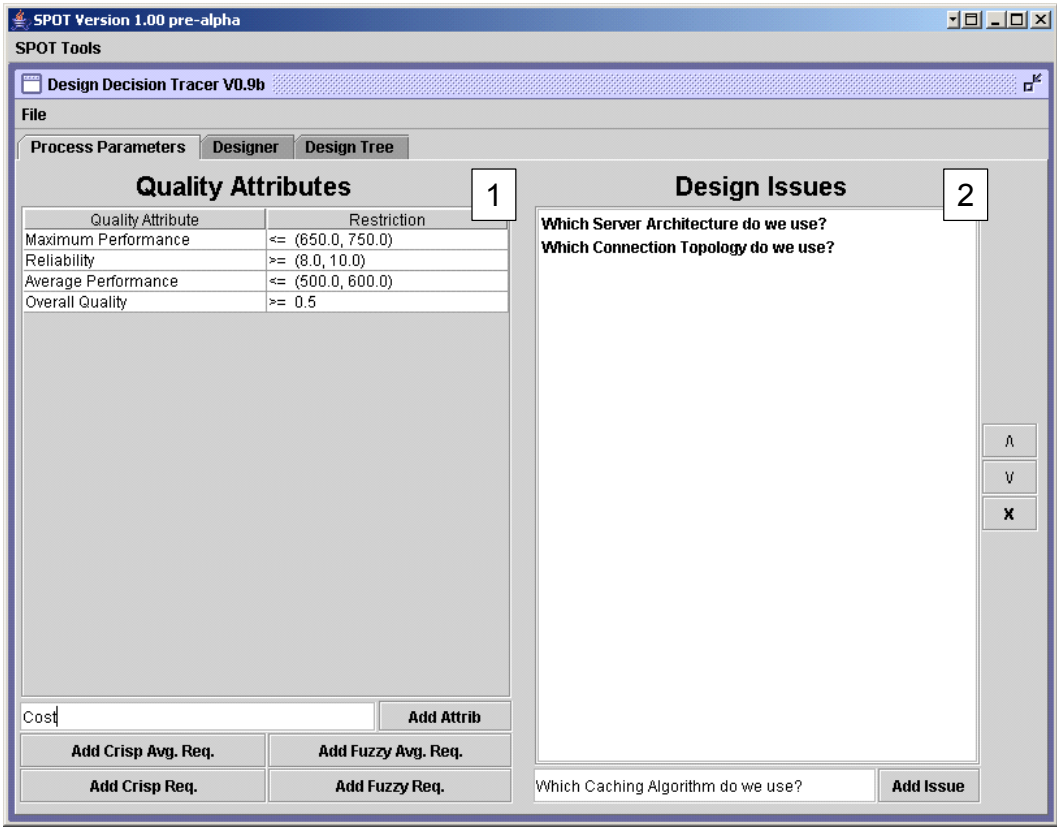


Figure 6.7 The Process Parameters Tab

The first tab in the DecisionTracer is the Process Parameters tab. In the Quality Attributes part (1) the relevant quality attributes are defined by means of a textual description. On these attributes both crisp and fuzzy requirements on the allowed average and boundary values can be defined. In the Design Issues part (2) the software engineer defines the design issues that need to be resolved using a textual description, and the order in which these issues are to be addressed. In the case that new design issues arrive as a result of resolving a design decision, the new issues can be added in the Designer Tab after which they are included in all subsequent design states.

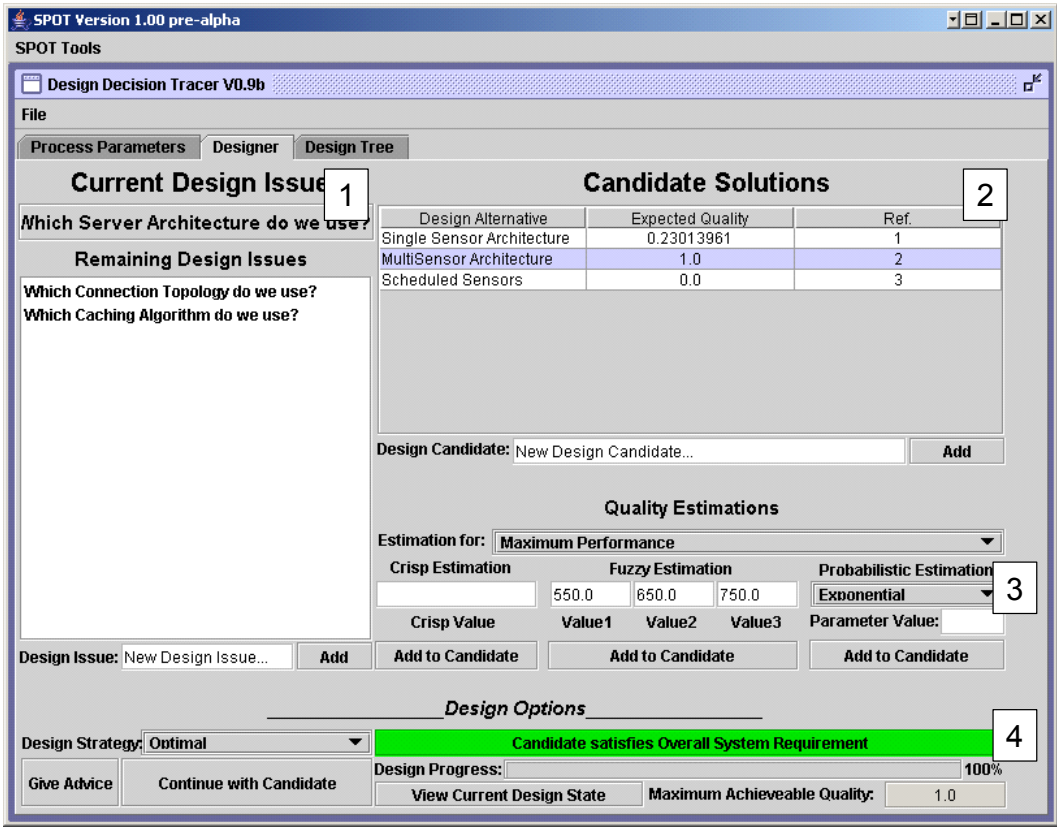


Figure 6.8 The Designer Tab

The Designer Tab is the tab at which the design issues are resolved in sequence. At (1) the current design issue is displayed, and below it is the list of design issues that must be resolved after the current one is completed. In the picture the current design issue is "Which Server Architecture do we use?", after which two more design issues need to be resolved. Candidate solutions for the current Design Issue are entered at (2), again using a standard textual description. After this definition the estimated quality of each quality attribute is entered using crisp, fuzzy or probabilistic models at (3). In the picture a fuzzy estimation is defined for the maximum performance of the MultiSensor Architecture. Using the controls at (4) the design state can be examined and the tool can offer decision support on the design state from which to continue.

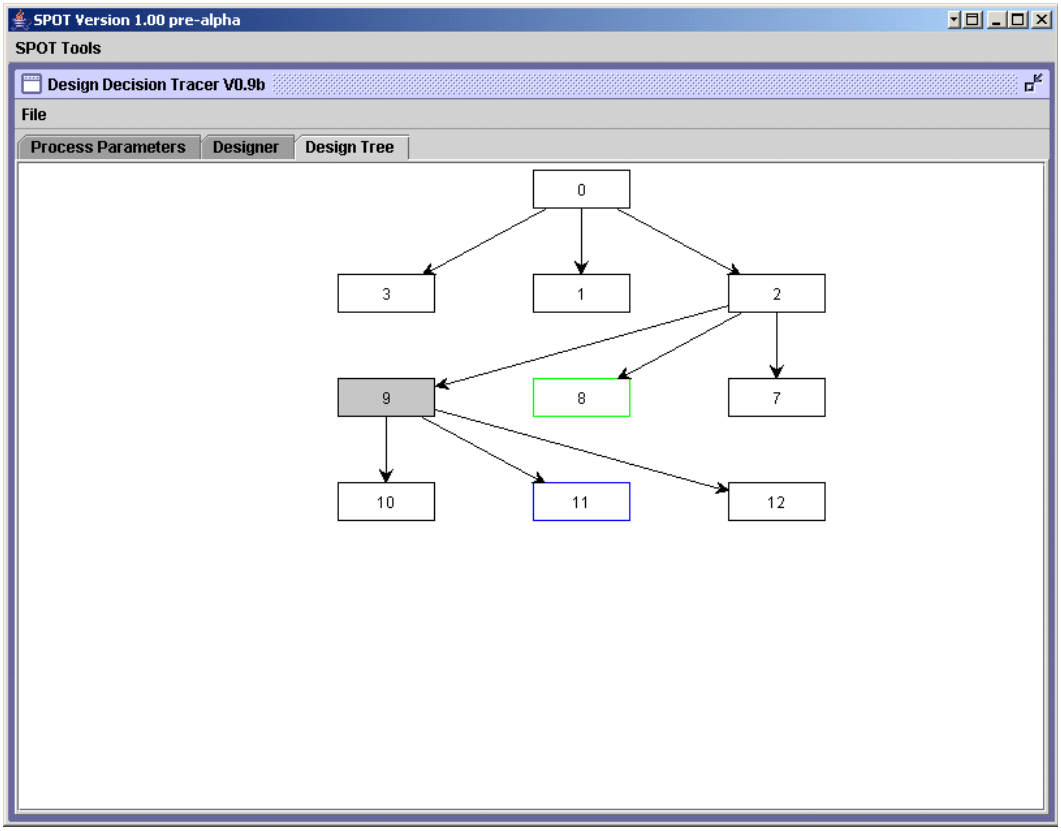


Figure 6.9 The Design Tree Tab

In the Design Tree Tab we can find the design tree representation of the design issues and contemplated alternatives. While it is not necessary to understand the Design Tree approach to use the Decision Tracer, it is possible to inspect the design tree of the current situation at any time in the Design Tree Tab. In the design tree one node is colored grey (9). This node represents the design state in which the design process currently resides. In addition, the tool indicates the optimal nodes with respect to two design strategies defined in chapter 4. In the figure node 8 indicates the best node to continue from according to the Optimal Design Strategy. Node 11 indicates the best node according to the Smart Design Strategy.

6.2.3 Resource Allocation Optimizer

The third tool in the SPOT Toolset is the Resource Allocation Optimizer, which implements the resource allocation optimization model that is described in chapter 5. The components and the dependencies must be provided by the software engineer, the personnel manager is supposed to provide the resources and constraints. The market analyst is required to provide the market demand states, events and probabilities. Based on these inputs, the tool provides scheduling advice and a simulation environment with which the project manager can schedule the implementation trajectory. The architecture of this tool is shown in Figure 6.10.

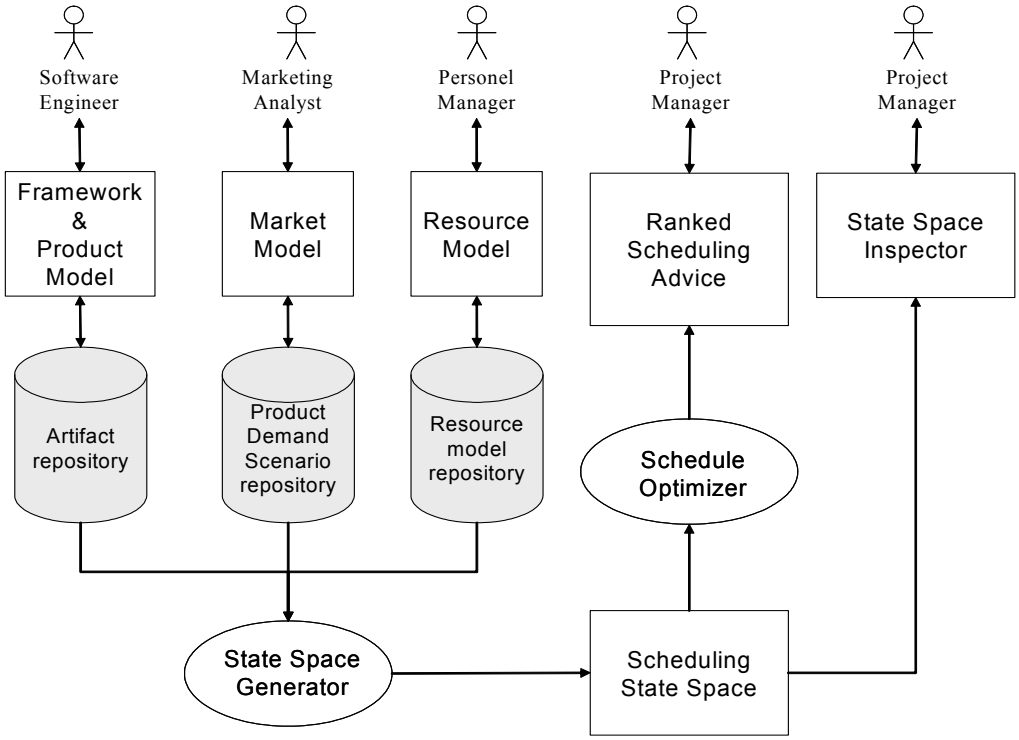


Figure 6.10 Conceptual Design of the Resource Allocation Optimizer

The tool requires artifacts (framework parts and software systems), software system demand scenarios and resources to be modelled by the software engineer, marketing analyst and personnel manager, respectively. The personnel manager must also define the goals of the optimization problem, such as minimization of cost, maximization of profit, etc. The process *Decision Space Generator* retrieves the necessary information from the artifact repository and the process *Scenario Space Generator* retrieves the necessary information from the Market Model Repository. Together they generate the data *Scheduling State Space*.

The next step is the determination of scheduling advice. The process *Schedule Optimizer* takes Scheduling State Space as input and generates the data *Ranked Scheduling Data* as output, which can be presented to the project manager in various formats. In addition, the project manager can run simulations by using the *Schedule Simulator*, which results in the data *Scenario Simulation Data*.

Architecture and User Interface of the Resource Allocation Optimizer

The Resource Allocation Optimizer requires the software engineer, marketing analysis and personnel manager to provide the application framework, market demand expectations and resource model. Based on this information the tool generates the combined graph and computes the scheduling advice. The architectural design of the Resource Allocation Optimizer is divided into four parts to reflect this structure.

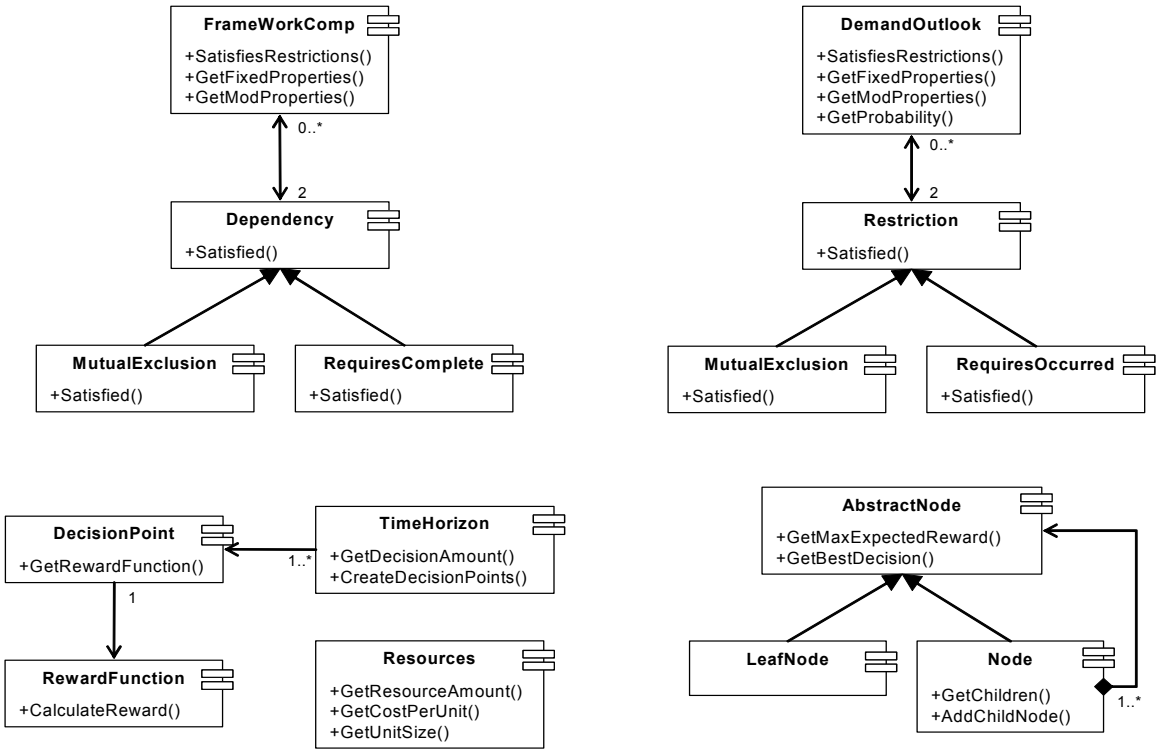


Figure 6.11 Abstract Architecture of the Resource Allocation Optimizer

In Figure 6.11, an abstract representation of the Resource Allocation Optimizer architecture is depicted in a standard UML-type description. In this architecture FrameWorkComp is used to represent the components of the application framework. Dependencies between these components can be described by the Dependency class. The DemandOutlook class is used to describe future market demand state and their probability of occurrence. In the case that two outlooks have a restriction on each other, such as mutual exclusion, the restrictions can be modelled using the Restriction class. The optimization properties of the Resource Allocation Optimizer that are modeled are the time horizon and the available resources. The time horizon consists of a number of DecisionPoints, each of which can have a individual RewardFunction. The available resources are modeled by the Resources class. The combined graph is modeled by the structure of classes at the right bottom of the picture. Since the computation of the reward depends on the type of the state (see chapter 5 for details), a distinction in the architecture is made between leaf nodes and non-leaf nodes.

The optimization algorithm of the Resource Allocation Optimizer offers two approaches to come to the scheduling advice. The first approach is the complete generation of both the scenario graph and decision graph as defined in chapter 5, and computing and solving the combined graph for the scheduling advice. The advantage of this approach is that the computed graphs can be inspected and used for the simulation of what-if scenarios. The second approach directly computes and solves the combined graph by recursive computation of the states of scenario and decision graph. While the time complexity of this approach is equal to the first approach, the space complexity is vastly reduced since it is not necessary to compute the complete scenario and decision graph before the scheduling advice can be determined. As has been indicated in chapter 5, the generation and evaluation is performed with complexity reduction techniques such as backtracking and dynamic programming.

The Resource Allocation Tool consists of three modeling tools, the *Parameter Modeler*, the *Event Modeler* and the *Optimization Modeler*. The application framework and the dependencies between the components are modeled in the Parameter Modeler. Additionally the avail-

able resources are modeled in the Parameter Modeler. In the Event Modeler the demand states and the probabilities of the transition events are modeled. The optimization parameters such as the time horizon and the reward specification for choosing a decision from a state is defined in the Optimization Modeler. The result of the optimization can be explored using the *Simulator Tool*.

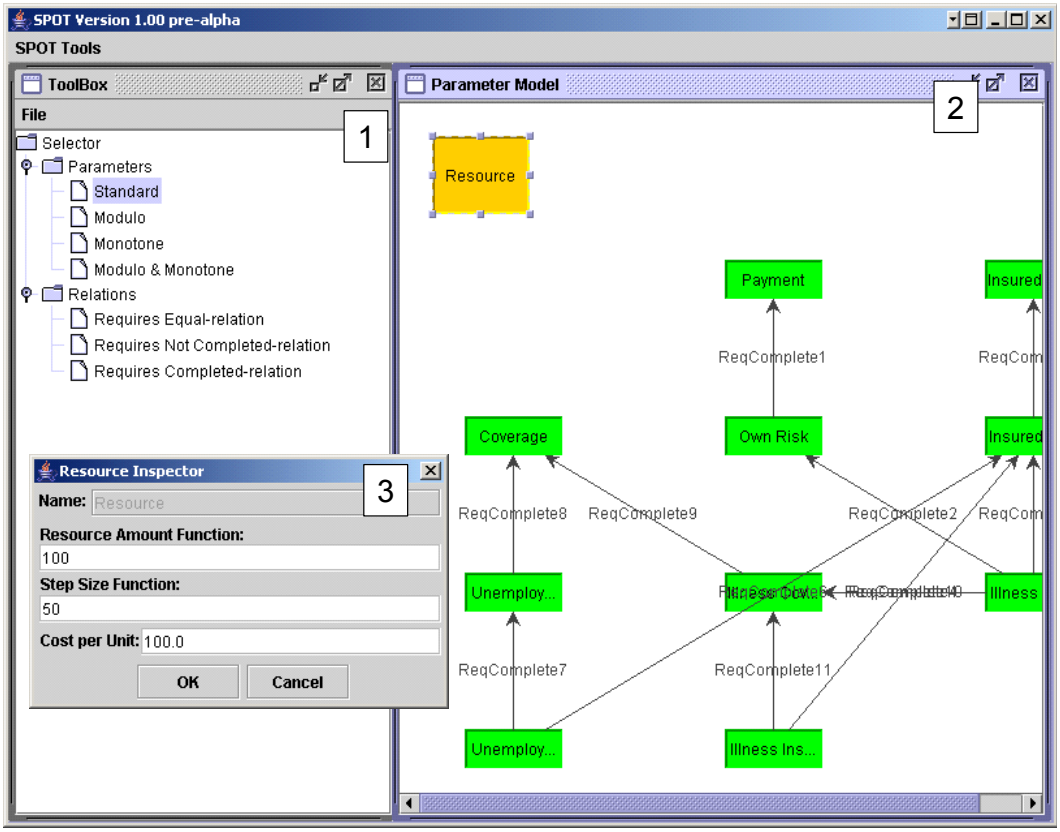


Figure 6.12 The Parameter Modeler

In Figure 6.12, the Parameter Modeler is depicted. The sub-window on the left side (1) displays the modeling tools for the definition of the application framework components. The tool distinguishes between framework components and products, based on which relevant properties can be defined such as cost. Also several types of relationships between components can be defined, such as a completion-dependency or a mutual-exclusion dependency. In the Parameter Modeler (2) the framework parts as well as the resources are modeled. In the figure the dialog-box for modeling the resources (3) is depicted. In this dialog box the resource amount function is specified with a regular expression language to facilitate the change of resources over time. The stepsize function indicates the minimal amount of resources that need to be assigned. The cost per unit is described using a numeric expression. While this part of the tool in the prototype is a stand-alone implementation, in an integrated environment the dependency information could for instance be acquired from UML models.

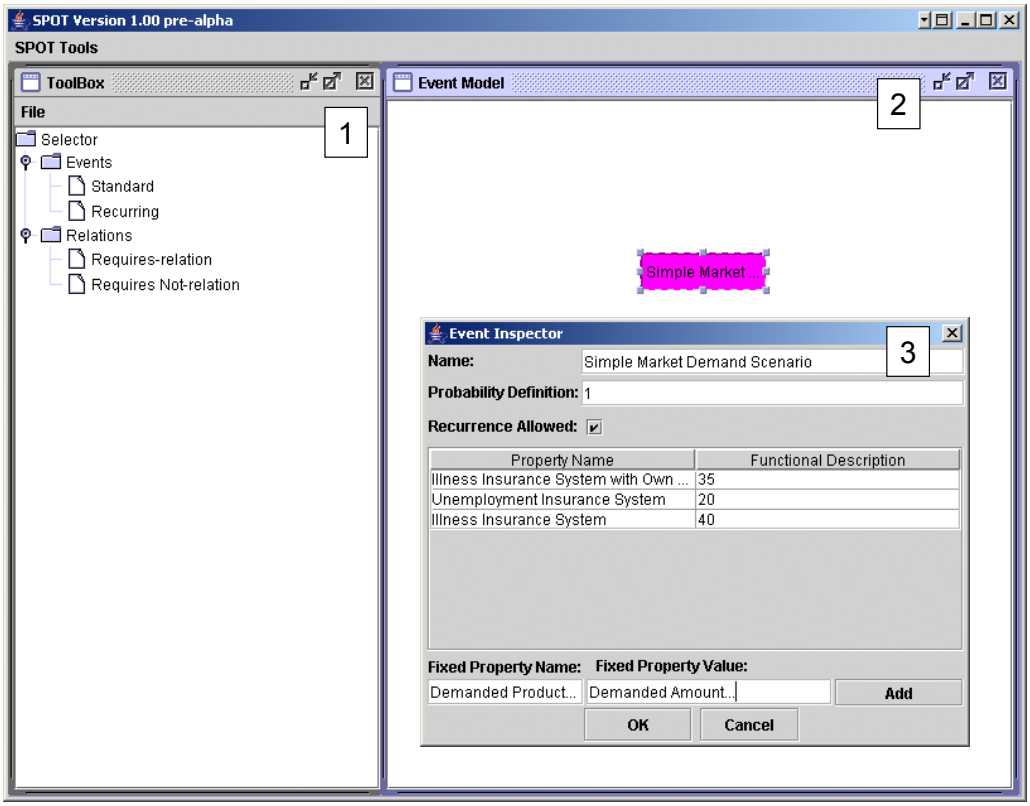


Figure 6.13 The Event Modeler

In Figure 6.13, the Event Modeler is depicted. Like the Parameter Modeler, the sub-window on the left side displays the modeling tools for the definition of the events and dependencies between them (1). In (2) the demand states and their relations are depicted. The event inspector (3) enables the modeling of the demand state that result from the event. The probabilities can be set as well as the demands for individual products in the dialog box that is also displayed in the figure. In this dialog box the demand state is named and given a probability function using a regular expression. In addition a number of properties can be defined such as the demand for a particular set of products. The name of these products must correspond to the products in the parameter modeler. The demanded amount again is a regular expression.

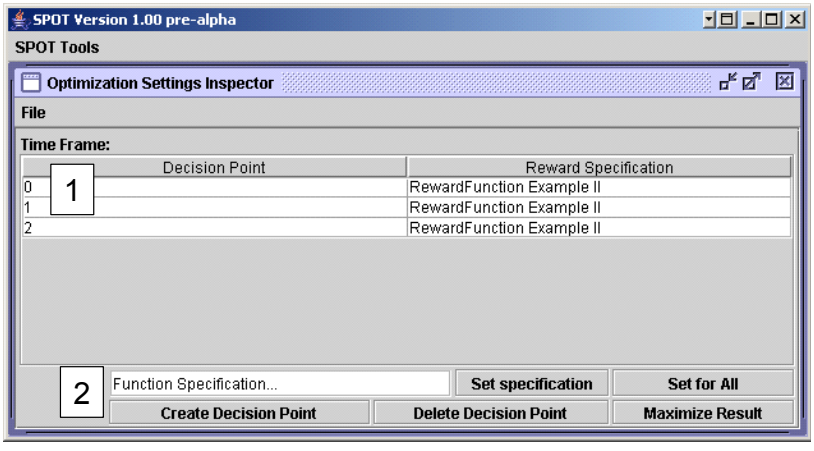


Figure 6.14 Optimization Modeler

The Optimization Modeler depicted in Figure 6.14 is used to model the time horizon and the reward specification. The decision points are indicated in the time frame (1) and describe the amount of times at which allocation decisions must be taken. The Optimization Modeler supports a reward specification that can differ per decision point. At (2) the reward specification is described using regular expressions. The reward specification in this picture refers to a predefined function, but in the future full regular expression support is planned.

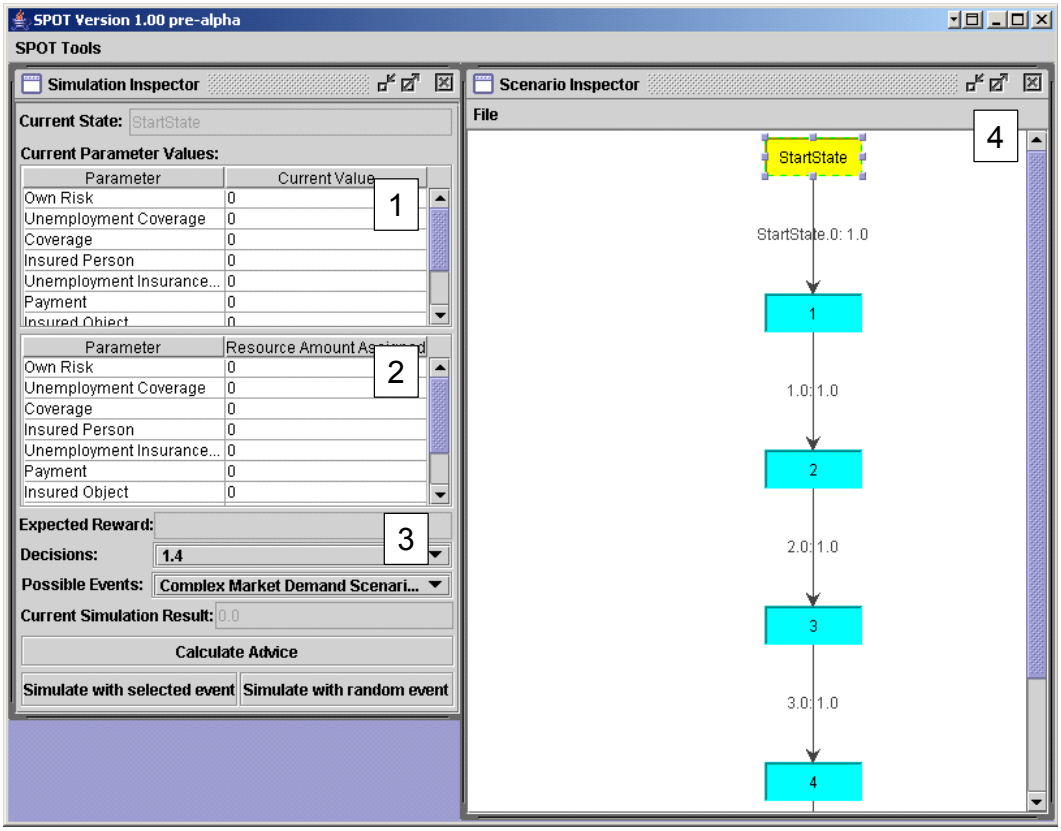


Figure 6.15 The Optimization Simulator

The Simulator Tool depicted in Figure 6.15 enables the project manager to explore "what if" scenarios that might occur during the implementation of the application framework. The Simulator Tool consists of two sub-windows. The left sub-window is the simulation inspector and displays information on the current state of the simulation, such as profit result, amount of work done, etc. At (1) the current work state is depicted, with for each component the amount of resources that have been assigned to it. At (3) the decisions that can be taken from the current state can be explored. For each decision the expected reward is indicated. The resource allocation details are displayed in (2). By selecting or randomizing the new demand state and selecting a decision, the software engineer can step through the implementation trajectory. The right sub-window (4) displays the scenariograph that describes the possible demand states and the current demand state. In this picture a deterministic scenariograph is depicted.

6.3 Implementation Issues and Points of Interest

The SPOT Toolset that implements the models in this thesis provides a valuable addition for the application of our approach. Although the toolset at the moment is aimed at demonstration purposes and being a proof-of-concept, it is already usable for supporting small to medium-

large software design processes. However, to fully utilize the added value of our approach, in this section we will shortly discuss some considerations with respect to tooling support.

Complexity and Performance

The models in our approach all describe (in part) an optimization problem. And as with all non-trivial optimization problems, the complexity and tooling performance can severely hamper the usefulness of the approach. The complexity of the optimization problems in our approach is typically exponential, which means that a straightforward implementation will arguably not scale up to industrial sized software design problems. To handle the complexity issues in our approach we have defined heuristics that reduce the exponential complexity at the expense of finding an approximation rather than the optimal solution.

In the SPOT toolset the models have been implemented including the heuristics proposed in this thesis. While the time performance was still very adequate with respect to the example cases that were analyzed, the usage of the heuristics resulted in a considerable increase in performance. With the use of recursive evaluation, the space complexity (i.e. memory usage) remained fairly minimal, which enables the usage of smart computation techniques that can reduce time complexity at the expense of space complexity, such as dynamic programming. How the tools perform in industrial-sized software projects remains to be seen however. At this moment the actual increase in complexity needs to be assessed as well as typical usage scenarios. For the future, a complete empirical assessment is planned based on the pilot study that is described in chapter 7.

Usability and Understandability

Another attention point, in addition to computational complexity, is the usability and understandability of our approach. While it is clear that our approach can assist software engineers during the design process, this will be most effective when tool support presents our approach in a very effective, understandable and usable manner. The design assistance that is offered should minimize the overhead for the software engineer and maximize the benefits.

In our pilot study we found that the participants appreciated the models defined in this thesis, because they made explicit the activities in design processes that were implicit before. As a side-effect this caused the participants to consider these steps of the design process in minute detail. The extra overhead caused by the mathematical computations of our models was largely handled by our toolset, and after a small number of iterations the usage of imperfection models become quite natural. The main effort of tool support therefore should lie in a ergonomically well-designed user interface, which enables to software engineer to use our models comfortably. Nonetheless, to assess the usability and understandability the toolset offers in the application of our approach, a more thorough analysis is required. For the future, a complete empirical assessment is planned based on the pilot study that is described in chapter 7.

6.4 Conclusions

The models that have been defined in this thesis enable software engineers to consider and include imperfect information in the software design process. To relieve the added workload of the application of our approach during software development and to ensure proper application of the imperfection reasoning models, we have presented a toolset that supports the software engineer. Our toolset consists of three tools: the Artifact Tracer, the Decision Tracer and the Resource Allocation Optimizer. The Artifact Tracer tool implements the Artifact Trace Model approach with support for fuzzy requirements, design optimization for multiple stakeholder interests and heuristic optimization capabilities. The Decision Tracer implements the Design Tree approach with support for imperfect quality requirements, imperfect quality estimations and multiple design strategies. Finally, the ResourceAllocator implements the resource alloca-

tion approach with support for multiple market demand states and probabilistic event descriptions. These tools have been implemented as a proof-of-concept and as a testbed to validate the applicability of our approach. In chapter 7, the applicability as well as the usability of the tooling as well as the approaches are evaluated by means of a pilot study. Based on this pilot study, an outline is given for a complete empirical validation of the models and tools proposed in this thesis.

CHAPTER 7

Halleck had spoken in Paul's ear: "Odd sort of fellow. Has a precise way of speaking--clipped off, no fuzzy edges--razor-apt." And the Duke, behind them, had said: "Scientist type."

- From Frank Herbert's Dune [Herbert2005]

EVALUATION AND CONCLUSIONS

7.1 Introduction

In this chapter, we evaluate our approach by means of a pilot study and summarize the results. In section 7.2, we examine the validity and applicability of our approach by means of a pilot study. The results of this pilot study are described in section 7.3. In section 7.4, we reflect on the main research issue: imperfect information in software design processes. We derived three problems: considering imperfection in functional requirement specifications, imperfection in quality evaluations and design alternative selection, and imperfection in scheduling resources for component implementation based on market demand expectations. We recapitulate the approaches we have defined for the identified problems in sections 7.5, 7.6 and 7.7 respectively. In section 7.8, we reflect on the benefits and drawbacks of our approach. Finally, in section 7.9 we define an outline for the future work on imperfect information in software design processes.

7.2 Validity and Applicability of our Approach

7.2.1 Introduction

With the approaches that have been proposed in this thesis, the imperfection in requirement specifications and estimations can be isolated and described. Using the reasoning mechanisms we have defined, this imperfect information can be included and considered during software development. Obviously, the usefulness of the proposed models and reasoning mechanisms largely depends on how well they reflect and support the development process. A key issue of our research is therefore to assess the validity and applicability of our approaches in a real world setting. To achieve a fair evaluation of the real world performance of our models, an empirical experiment is required, where software design is performed under controlled conditions. By comparing the resulting software architectures of a traditional and imperfection supported development processes, the added value of the latter can be evaluated. However, the setup and execution of an empirical experiment requires considerable planning and for this type of approach it is difficult to evaluate experimentation results. The main cause for this difficulty lies in the absence of measures that can be used to evaluate the results of the experiment. In typical empirical validations in the field of software engineering, well-known quantifiable properties such as lines of code or McCabe complexity can be used to compare for instance source code of software modules. Other metrics that are used are based on coupling, cohesion, etcetera. The evaluation of our approach is not as straightforward, since here the effectiveness of the underlying development process must be assessed. For this type of evaluation metrics are not readily available.

To explore the requirements and attention points for empirical assessment of our approach, we have conducted a pilot study, which provided insights and feedback for the definition of a full empirical experiment. This pilot study was conducted as part the master course *Advanced Design of Software Architectures-I* and consisted of groups of students that performed an architecture design based on imperfect requirement specifications. The architectural design was performed in two iterations; in the first iteration the architectural design was performed without means to model imperfection in the requirements. In the second iteration, the imperfection in both functional and quality requirements and quality estimations was modeled and considered during a redesign of the software architecture. During the second iteration the design activities were supported by the tools of the toolset that was introduced in chapter 6. In the following we describe the goal and setup of the pilot study. The results and conclusions as well as a starting point for empirical evaluation are given in section 7.3.

7.2.2 Goal and Setup of the Pilot Study

The models that are described in this thesis aim at preventing faulty or premature decisions in software design processes by explicitly modeling imperfection in requirements and estimations. To determine whether these approaches and the tools that implement them are successful, a pilot study has been conducted within the master course ADSA-I. Empirical validation of models for real world application is typically very difficult within an artificial setting such as master courses, which is identified in [Carver2003]. The causes for this lie, for example, in the fact that student behavior typically does not correspond to the behavior of software engineer during software development and the overall difference in the working environments. The results of empirical validation in software courses, therefore, not necessarily reflects validity in real world application. Based on these limitations and the partial knowledge on the empirical validation of this particular type of model, we have opted for a pilot study that gives a qualitative evaluation of our approach as well as insights on the elements that are needed to perform a complete empirical validation.

The pilot study was conducted with the following goal: *"We want to analyze the impact of explicitly modeling imperfection on the optimality of the software design"*. To achieve this goal, the students were asked to design the architecture of a software system based on one of two available example cases. In both example cases the descriptions and requirement specifications were deliberately left vague and unclear to simulate imperfect information received from the stakeholders. To compare the results from a crisp requirement interpretation and design with the results of applying our imperfection approach, the pilot study was divided into two phases:

1. *Working out an example case with the design tree and artifact trace model but without the support for capturing the imperfection present in the requirement specifications.*
2. *Working out an example case with the design tree and artifact trace model and with the models for capturing the existing imperfection supported by the SPOT Toolset*

In the first phase, the students worked out one of two example cases according to a simple software design process based on the Artifact Trace Model and the Design Tree approach. The example cases contained considerable imperfection in several functional and non-functional requirements. However, in the first phase the students were given no means to address this, which forced the students to make explicit assumptions on the meaning and interpretations of the imperfect information in order to come to an architecture design. In the second phase, the students were introduced to the extensions of the Artifact Trace Model and the Design Tree Model that enables software engineers to capture and evaluate imperfect information. Based on these extensions, the students revised their architecture design by applying the developed toolset to the example case. As a result, in the second phase the students had to identify imperfection in the requirement specifications explicitly, and model it accordingly.

Due to the limited time available inside the course, the design steps of the pilot study were minimized in order to come to an architecture design at a fairly high level of abstraction. To guide the students during the design phase, a simplified version of the analysis synthesis process Synbad (see chapter 2) was introduced. In this simplified design process a single refinement iteration is performed from requirement to a high-level component structure. In addition the students were expected to identify three problems in the second refinement step that should be analyzed using the design tree approach. For each of these problems at least three design alternatives should be considered and evaluated according to the design tree approach. A schematical depiction of the simplified design process is given in Figure 7.1.

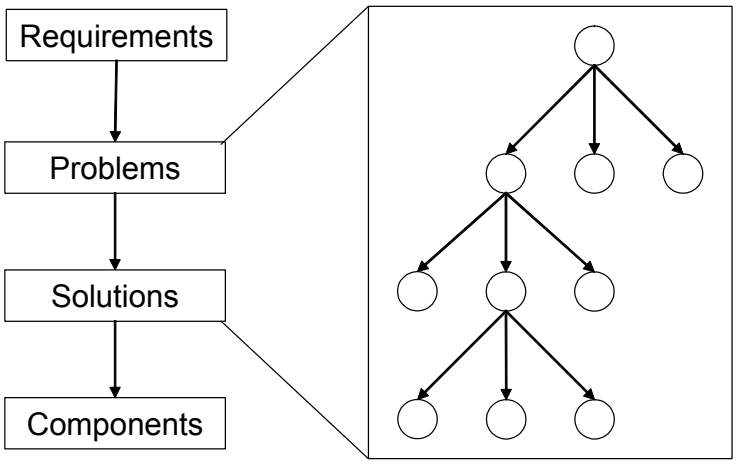


Figure 7.1 Simplified Software Design Process for the ADSA-I Experiment

It can be seen that the simplified software design process only covers a small part of the activities that software engineers perform during actual software design. However, in accordance with the goal of the pilot study, the simplification does not remove the typical problems that software engineers encounter when they are faced with imperfect information. Rather it isolates and underlines these problems and as such the study is a useful means for a qualitative evaluation of our approach.

The evaluation of the results of the pilot study is based on analysis of the models that result from the design process. In addition, the students were asked for their views on the applicability and usability of the approach as well as the toolset that supports it. To ensure a fair evaluation the students were explicitly asked to assess the approach and toolset with respect to the following points:

1 *Ease of use*

2 *Extra insights gained by the approach*

3 *Most relevant parts and strong points*

4 *Least relevant parts and weak points*

By collecting the results of the student evaluation and by analyzing the architecture design that result from the application of our imperfection models we draw initial conclusions with respect to the validation of our approach. In addition, we propose an outline and considerations for setting up and conducting an empirical experiment with which this type of research can be validated.

7.2.3 Examples for the Pilot Study

For the pilot study, 24 students were divided into six groups consisting of four people. Each group was assigned one of two possible example cases. These example cases are modified versions of the examples that have been used in the previous chapters. In particular the requirement specifications have been altered to ensure differentiation in the example case, as well as different imperfection. Below the descriptions are given of the two example case that were used.

Example Description I: Storm Surge Barrier

Consider a storm surge barrier designed to protect a moderately populated urban area. The barrier has to be closed only in case of absolute necessity; otherwise the cargo transport can be hampered unnecessarily. However, leaving the barrier open during storm situations can result in immediate danger for the population. Since the decision to close the barrier is a complicated task, it has been decided to incorporate a computer-controlled system for this purpose. The control system should make a decision every 5 seconds, based on numerous inputs such as weather forecasts, changes in the water level, tides, etc. whether or not it should go to an alert state. This example focuses on the data collection and storage facility (DCS). The description that is provided by the stakeholders for this particular part is as follows:

"The DCS shall support the collection and retrieval of decision data by the decision support system. This should be achieved by communication with the data providing entities, such as water sensors, meteorological institutes, etcetera. The acquired data that is gathered and

stored describes information about its direct and indirect geographical vicinity. The data storage must be usable in a convenient manner and must be usable for the display system. To communicate the stored data, the DCS must be accessible from various locations in a uniform manner.

Note that both the functional and non-functional requirement specifications are variants of the example case in chapter 4. We summarize the functional requirements for the DCS from this specification as follows:

1 *The DCS system shall collect information from various sources and locations*
2 *The DCS system shall store the information in a convenient manner*
3 *The DCS system shall support data formatting for the display system*
4 *The DCS system shall be accessible from various geographical locations with a uniform interface*

In addition the following non-functional requirements are provided by the stakeholders:

1 *The DCS must be able to retrieve and format data on average within 500 milliseconds after initiating the request.*
2 *The DCS must be able to retrieve and format data on average within a maximum of 650 milliseconds after initiating the request.*
3 *The DCS shall be able to store and retrieve the data of the last four weeks*
4 *The cost of this system must not exceed 200K euros.*

Imperfection in Example I

The imperfection in the DCS example at the first stage is predominantly present in the functional requirements specification, where terms like “various“ and “convenient” are used to describe desired properties of the system. Note that in the non-functional requirement no explicit imperfection is present. However since an abstract software architecture is designed means that it is difficult to estimate the expected quality properties of the completed system, which introduces imperfection in the quality estimations in a natural manner.

Example Description II: Traffic Management System

Consider a Traffic Management System (TMS), designed to monitor and regulate the traffic flow on a national scale. To utilize the infrastructure fully and to plan the future of the traffic systems, a new TMS is being developed. The system is supposed to provide the necessary technical support for monitoring, controlling, managing, securing and optimizing the traffic flow effectively. Since this scale and scope of the TMS is too large to consider completely, we will focus on the section which handles task allocations based on scenarios and available traffic information. The description that is provided by the stakeholders for this particular part is the following:

"The TMS must support the convenient allocation of tasks by the system operators. This is achieved by the definition of scenarios that can take place. The Task Allocation part should gather and store information about traffic in its direct and indirect geographical vicinity. The defined tasks and scenarios should be dynamically modifiable according to the newest traffic developments. To communicate the tasks and actions, the TMS should be able to access its connected roadside systems. In addition, the TMS should support systems operators in identifying tasks and actions that will normalize traffic flow as fast as possible."

Note that both the functional and non-functional requirement specifications are variants of the example case in chapter 3. We summarize the functional requirements for the TMS from this specification as follows:

1 *The TMS system should collect information from various sources and locations*
2 *The TMS system should store the task and scenario information in a convenient manner*
3 *The TMS system should support data formatting for the display system*
4 *The TMS system should be accessible from various geographical locations with a uniform interface*

In addition the following non-functional requirements are provided by the stakeholders:

1 *The TMS must be able to communicate the actions to the roadside system within on average 500 milliseconds.*
2 *The TMS must be able to communicate the actions to the roadside system within a maximum of 650 milliseconds.*
3 *The TMS should be able to maintain the connection to the roadside system for at least 10 seconds in case of failure.*
4 *The cost of this system must not exceed 200K euros.*

Imperfection in Example II

The imperfection in the TMS example is similar to the imperfection of the DCS example. Imperfection is predominantly present in the functional requirements specification, where terms like "various" and "convenient" are used to describe desired properties of the system. Again in the non-functional requirement no explicit imperfection is present, but the fact that an abstract software architecture is designed means that it is difficult to estimate the expected quality properties of the completed system.

7.3 Results of the Pilot Study

7.3.1 Introduction

The pilot as described in the previous section was conducted in the course between the 11th of December 2006 and the 19th of January 2007. In the first lecture the students were introduced to the Artifact Trace Model and the Design Tree approach as means to trace intermediate

design artifacts, design decisions and contemplated alternatives. Also the students were familiarized with the example cases and the simplified design process. During the first stage of the pilot study the students performed the architectural design of their respective example case, and traced the activities. At this point the imperfection models were not known to the students, which simulated the design of software without support for imperfect information. In a presentation session the students presented and discussed the results of their design activity, as well as the difficulties that were experienced during this process. In the second lecture the imperfection models for the Design Tree and the Artifact Trace Model were introduced to the students. Based on the identified difficulties during the presentation session the problems of imperfect information were introduced and addressed. Based on these extension the students revised their initial design activities by explicitly identifying and modeling imperfection in the example cases. To support these activities the redesign was executed as a lab assignment where under supervision the students used the toolset to describe and revise the architectural design.

The results of this pilot study can be divided into two parts, which correspond to the two activities that were performed by the students. In section 7.3.2 we examine the results from the artifact trace models and design tree models application without and with imperfect support respectively. We evaluate these result with respect to the influence the imperfect requirements had on the initial design, the added complexity of using our approach, the benefits of using our approach and the applicability and usability of the toolset. In section 7.3.4 we use the results and evaluations to define the starting point for empirical validation of our research.

7.3.2 Imperfect Information Models in the Pilot Study Setting

The first phase of the pilot study introduced the students to the two example cases, which were intentionally left vague and ambiguous in their description. The students were therefore forced to resolve the imperfection in the requirement specifications in the most applicable manner according to their judgment. The resulting architectures from this design steps differed considerably for the groups, since the chosen interpretations of the imperfection could not be resolved with the stakeholders. Interestingly, a number of students did not explicitly identify the imperfection in the requirement specifications, but immediately jumped to their own interpretations without being aware of this step. After introducing the concept of imperfect information to the students and the models in our approach, the students revised their initial designs while explicitly modeling the imperfection in the functional and quality requirements.

Imperfect Requirement Specifications used in the Artifact Trace Model

First we examine the results of the modelled imperfection in the functional requirement specifications. In particular we are interested to see how the starting point, which was identical for all groups, results in different interpretations and designs. In Table 7.1 and Table 7.2 we summarize the identified requirement interpretations for the Traffic Management System and the Data Collection and Storage System respectively.

Table 7.1 Interpretations for the Traffic Management System

Requirement Description	
1	**The TMS should collect information from various sources and locations**
1.1	The TMS should collect formatted information from other computers
1.2	The TMS should collect raw information from sensors and cameras
2	**The TMS should store the task and scenario information in a convenient manner**
2.1	The TMS should store the task and scenario information in a human-readable format
2.2	The TMS should store the task and scenario information in a standard convertible format
3	**The TMS should support data formatting for the display system**
3.1	The TMS should supply textual data to the display system
3.2	The TMS should control the display system using geographical data
4	**The TMS should be accessible from various geographical locations with a uniform interface**
4.1	The TMS should have clearly defined interfaces
4.2	The TMS should be accessible through a plug-in architecture

In this first table the interpretations for the functional requirements of the Traffic Management System are given. Previously, we have identified that in the second requirement the term "convenient" is a vague expression as well as the "various sources" for information in the first requirement. In the interpretations we see that the term convenient was interpreted as "convenient for the software engineer". Also we see that for all the requirements alternative interpretations were proposed, even while the first impression would not directly indicate a need for it.

Table 7.2 Interpretations for Data Collection and Storage System

Requirement Description	
1	**The DCS system should collect information from various sources and locations**
1.1	The DCS system should go through different systems and fetch the data
1.2	The DCS system should receive information from various source systems
1.3	The DCS system should be able to both receive and fetch data
2	**The DCS system should store the information in a convenient manner**
2.1	The DCS system should store the information in an XML-based format
2.2	The DCS system should store the information in a binary format
2.3	The DCS system should store the information in a human-readable format
2.4	The DCS system should store the information in a centralized manner
2.5	The DCS system should store the information in a distributed manner
2.6	The DCS system should store the information in a hybrid manner
3	**The DCS should be accessible from various geographical locations with a uniform interface**
3.1	The DCS system should be accessible from various geographical locations through a stand-alone client-side application
3.2	The DCS system should be accessible from various geographical locations through a web-based interface

In Table 7.2 the alternative interpretations for the requirement specification of the Data Collection and Storage System are given. Since different groups performed the design of the DCS, also other interpretations were given. In this case requirement 4 was not considered for interpretations, but especially the term “convenient“ in the second requirement led to many alternative interpretations. We see that interpretations 2.1, 2.2 and 2.3 interpret “convenient“ in terms of the storage format, where the interpretations 2.4, 2.5 and 2.6 focus on the storage structure that will be used.

In addition to the definition of alternative interpretations, also a number of stakeholder interests were defined, based on which the system was evaluated. The following stakeholder interests were identified by the students: Relevance, Learnability, Response Time, Cost, Adaptability, Extensibility, Reliability, Maintainability. While relevance is frequently mentioned in the literature describing the Artifact Trace Model, the others were not. As such, these attributes indicate attributes that are of real interest to the stakeholder. However, most of them lean towards actual quality attributes rather than higher-level stakeholder interests such as relevance.

Imperfect Requirement Specifications in the Design Tree Model

The second element of the pilot study consisted of identifying a number of design issues that need to be resolved, and trace the contemplated and selected alternative solutions using the Design Tree approach. Since the groups started with identical requirement specifications uniformity to a certain extent was expected in the design issues that should be resolved. This is strenghtened by the fact that the requirement specifications for both systems are fairly similar. Below the design issues are listed for the Traffic Management System and the Data Collection and Storage System.

Design Issues for the Traffic Management System

- How to collect information from various sources and locations?
- How can the system store task and scenario information?
- How can task information be stored in a usable and adaptable manner?
- How can scenario information be stored in a usable and an adaptable manner?
- How can data be formatted to display on screen?
- How can the system become scalable?
- How can the system be connected with other systems?
- How do we support multiple sources reporting information simultaneously?
- How do we support multiple types of information?
- How do we format data?

Design Issues for the Data Collection and Storage System

- Which communication network do we use?
- How do we schedule the information retrieval?
- How do we extrapolate data in case of failure?
- How do we compress the data?
- How do we gather the information from the sensors?

- How do we provide a user interface to the DCS?
- How do we store data?
- Where do we store data?
- How do we control the data retrieval?

The design issues that were identified for the TMS and the DCS again show overlap, since the functional requirements for both systems are very similar. However, there is still a rather large variety in the design issues that were found. In the literature provided to the groups, the quality requirements were similar to the requirements in the example cases. As a result the identified quality attributes mostly correspond to the attributes identified in the literature. However, a number of quality attributes were fairly new, such as training time, completeness and communication speed, which can be seen as refined attributes of, for instance, general performance or usability. The "traditional attributes" such as cost and overall performance were mentioned by all the groups as relevant quality attributes. The identified quality attributes are: maintainability, total cost, communication speed, training time, reliability, average performance, overall performance, usability and completeness.

7.3.3 Pilot Study Evaluation

The pilot study gave a first insight into the applicability of our approach, and the issues that need to be considered for decision support systems based on our models. The results and experiences of the students during the design of the software architectures were much in line with our assumptions on the influence of imperfect information on the software design process. In the case that the software designers are presented with incomplete information either (implicitly) assumptions are made on the intended meaning or clarification is sought from the stakeholder. For this pilot study this meant a number of enquiries from the students about the actual meaning of the requirements specifications. The setup of the experiment to introduce the concept of imperfect information at a later stage clarified and underlined the problems that occur as a consequence. In the following, we evaluate the results of the pilot study with respect to the use of the imperfection models as well as the use of the tooling support for the application of the models.

Evaluation of the use of Imperfection Models

First we evaluate the use of the proposed models with respect to their ability to support software design with imperfection information. The way in which the imperfect information in the TMS and the DCS case were addressed by the students initially, confirmed the supposition that imperfection is rarely identified and resolved explicitly. Rather, assumptions on the implied meaning are made, while most students where not aware of this step. The application of the Artifact Trace Model made this implicit step visible, which caused the groups to rethink the requirement specifications they were given. By modeling alternative interpretations, the number of system designs that are considered increase considerably. During the pilot study the number of system designs that were analyzed for the TMS was 45 and for the DCS even 101. While most of these system are closely related, they typically are variants of system designs that would not be considered without modeling imperfection in the requirement specifications. The explicit link that is made between requirements and components improved the insight of the groups into the system design, and the optimization capabilities of the Artifact Trace Model enforced a structured decision making process. The modifiability of the optimization configurations of the Artifact Trace Model enable groups to experiment with several optimization criteria. This particular aspect was well received, since the optimized system designs in particular cases contrasted to the intuitive judgment of the students. This forced a closer

inspection of the proposed systems, and in particular cases revealed modeling mistakes or component combinations that were overlooked. It was proposed by the students that for the identification of alternative interpretations it would be beneficial to have examples based on domain knowledge or previous design activities in the same area.

As with the Artifact Trace Model, it was well appreciated that the Design Tree Model provided means to explicitly identify and evaluate design alternatives. As a result, the groups identified and considered considerably more alternatives than normally would have been the case. The Design Tree approach was especially successful in distinguishing between alternatives that were "barely" and completely acceptable according to the quality requirements. The added effort that is needed for its application was identified by the students as the greatest difficulty. In particular, the fact that for every identified alternative quality estimations need to be made was problematic, which was solved only in part by the current toolset. The tracing capabilities of the design tree were also appreciated since they gave valuable insight into the considered and selected alternatives. Additionally, the scale on which quality attributes are estimated is very important. For specific quality attributes such as performance or cost the scale is well-known (response-time or person-hours). However, for other quality requirements this is often not the case, which can complicate the use of the model.

When we evaluate the overall approach both the Design Tree Model as well as the Artifact Trace Model made the students aware of the existence and impact of imperfect information in software design processes. However, the main difficulty in the pilot study was caused by the fact that the students were not capable of acquiring and defining proper numeric input for the methods. Where software engineers can assess the expected quality attributes based on their experience, for students this was much more difficult. As a result it is difficult to compare the architectures that result from the optimizations to the architectures without considering imperfect information, since their applicability largely depends on the validity of the inputs. Additionally, for both models the resulting systems on occasion were not applicable since the combinations of alternatives not always reflects a desirable system.

Evaluation of the Tooling Support for Imperfection Models

Second we examine the usefulness of tooling support for the proposed imperfect information models. As mentioned in the previous section the effort needed to apply the imperfection models can be considered the most prominent difficulty of our approach. To test the ability of the SPOT Toolset of managing this added complexity, it was tested during the pilot study. Also the effectiveness of the implementation was tested in this manner, since decision support for software development processes should accurately provide the software engineers with the desired information. During the design of the architectures of the TMS and DCS example cases the tools demonstrated their use in reducing the complexity of the application of our approach. However, since the example cases were rather small and the pilot study limited in the available time, the scalability of our approach can not be completely assessed based on these results. The first indications from this pilot study indicate that the toolset, combined with the proposed improvements, is capable of managing the increased workload.

The way in which tooling support is implemented and offered to the software engineers can be used to ensure the proper usage of the proposed imperfection models. By defining a strict workflow, and supporting this with automated decision support and design knowledge, the software engineer can use the models without being faced with an unacceptable increase in design effort. In the current toolset this workflow is not yet complete enough to ensure the proper usage of the functionality. Since the toolset was implemented as a proof-of-concept it is not always clear for the user which inputs are needed, and how to provide them. The ergonomic improvements proposed by the students were therefore aimed at structuring the complex inputs understandably and the reuse of available information to support the software engineer in the understanding of the software design process.

7.3.4 Starting Point for Empirical Validation of Imperfection Models

The pilot study we have performed to assess our approach has given us a first indication of the benefits and drawbacks. It also provided insights on attention points for conducting an empirical experiment of the proposed models. In this section we identify the points of attention which will be used as the starting point for empirical validation of our approach.

Goal Definition

The most important element of an empirical experiment is the definition of a measurable goal. For our approach there are aspects that can be assessed: the first is the applicability of our approach and the second is the level of support that is offered by implemented tool sets. However, from the pilot study it has become clear that manual application of our approach in an experimental setting quickly becomes unmanageable. It is therefore advisable to evaluate the approach and the supporting toolset in a single experiment setting.

The goal of the approach is to support imperfect information in software design processes, with the aim to reduce the amount of work needed at the moment the imperfection is removed by an external influence. Therefore the goal of the experiment should be: *the analysis of the impact of imperfection with respect to the effort needed to adjust to new interpretations of imperfect requirements*.

Evaluation Model

To ensure an objective evaluation of the experiment results, the comparison of resulting architectures should be well-defined. The main goal of our approach is to make software design more resilient to imperfect information by including imperfection descriptions in the software design. The benefit of such a design over a traditional design would therefore lie in its ability to support alternative interpretations of the imperfect information. As a result, the amount of effort needed to adjust the design to the refined requirement specification should therefore in general be reduced.

To measure this improvement, a metric needs to be defined that captures the effort of adjusting designs to refined requirement specifications. Such a metric in particular should focus on either the amount of time and effort needed or the costs generated by the resolved imperfection in the requirement specification. Naturally, such a metric will depend on the flexibility and experience of the design team as well as the flexibility of the software design and the type of project. Therefore, to draw valid conclusions from the empirical experiment the evaluation needs to be performed over multiple design teams and projects.

In addition, the sensitivity of the model also needs to be assessed, in particular with respect how the values that result from the models can be used. By experimenting with small variations the required accuracy for the models need to be assessed.

Overall Experiment Structure

For the empirical validation the experiment should faithfully capture the industrial setting in which imperfect information can manifest itself. To analyze the benefits of our approach first a software architecture needs to be designed based on an imperfect requirement specification by a team with and a team without imperfect information models and tooling support. In the second phase the imperfection in the requirement specification is systematically removed, after which both groups have to adjust their designs accordingly. After each adjustment step the effort needed by the teams is measured and summarized. For the assessment in general this

process should be repeated for multiple imperfect requirement specifications as well as with additional groups to eliminate the influence of experience and “lucky“ choices.

Experiment Setting and Test Group

The setting of the experiment and the groups performing the system design need to be considered carefully. The results from the pilot study suggest that the example cases containing imperfect information must be based on actual requirement specifications since the lack of a real-world background could keep the experiment setting too abstract, which leads to undesirable results. Therefore it is advisable that example cases are defined based on existing projects and modifications of existing requirement specifications. In this manner also the architecture resulting from the experiment can be compared to the actual design from the case.

Additionally it is very important that the groups performing the design activities in the empirical experiment are sufficiently experienced in the design of software systems and architectures. Since the imperfection models require relevant estimations and inputs from the software engineers in order to provide meaningful decision support, it is vital for the groups to have the relevant design experience and insights to provide these inputs.

7.4 The Problem of Imperfection in Software Design Processes

In this thesis we have identified the existence of imperfect information as a prominent problem during software development. Under the best of circumstances the design of software systems has proven to be very difficult, but typically software development is performed under circumstances that are considerably less than ideal. Requirement specifications are subject to changes, expectations can misrepresent the quality of the completed system and the input received from stakeholders can be vague and unclear. We have identified that imperfect information can originate from many sources, such as the stakeholders, the software engineers, measurements, etcetera.

While it is acknowledged by most software development processes that providing precise requirement specifications is vital for the delivery of high quality software, only few offer explicit support for the definition of such requirements. Nonetheless, modern software design processes are aware that requirements are likely inaccurate and subject to change, and incorporate mechanisms to address the consequences. Most software development processes use iterative design as the primary means to address consequences of imperfect information. By evaluation of the design after the completion of development phases, software engineers can acquire new information and adjust the design accordingly. Depending on the process, iterative cycles can range from a number of weeks to six months or a year. In chapter 1, we have identified that the iterative cycle does not necessarily resolve all the problems caused by imperfect information. For incremental design to succeed, we have identified four conditions that must be fulfilled:

All-Isolation Requirement: For the design to be corrected effectively with iterative design, all concerns must be orthogonal; otherwise concerns that are influenced by imperfect information cannot be isolated and made perfect eventually.

All-Always Requirement: For the design to be corrected effectively with iterative design, all requirements must be frozen; All requirements must always stay the same.

All-at-Once Requirement: For the design to be corrected effectively with iterative design, all unknowns of a dependent design part must be resolved at the same time, otherwise there will be always some influence of imperfect information

All-Infinite Requirement: For the design to be corrected effectively with iterative design, all the required resources must be infinitely available, otherwise iterations cannot be applied until the imperfections are resolved.

Obviously, in any realistic setting these requirements can not all be fulfilled, from which we conclude that imperfect information can not be fully resolved by iterative design. In addition, in order to maximize the benefit of incremental design, iteration should be performed in a systematic manner. However, while this is already difficult in ideal circumstances, due to a limited understanding of imperfect information iterative development can become even less effective.

To contain the impact of imperfect information, we have proposed a generic approach, which identifies and describes imperfection that is present in information sources within the software development process. These descriptions are then considered and used in subsequent steps of the development process, such as design decisions, in order to minimize the chance of making wrong decisions and guiding iteration cycles. To support the inclusion of imperfect information in decision making processes, the imperfection models are used in decision support approaches that describes software design activities. As opposed to traditional software development methods, this approach offers the possibility to explicitly model the imperfection that exists in the available information. With the definition of reasoning models that consider the influence of the imperfection in the decision making process, it becomes possible to assess the risks an opportunities that are introduced into the development process.

In accordance with the literature, we have divided imperfect information into two categories depending on their nature, impreciseness and uncertainty. In addition, we have addressed these types of information accordingly in our models. To offer specific support, in this thesis we have addressed imperfect information in three areas of the software development process; imperfect functional requirements, imperfect quality requirements and estimations, and imperfect market demands, each of which is implemented in the SPOT Toolset. In the following we shortly recapitulate the individual approaches.

7.5 Resolving Imperfect Functional Requirements and Trade-off

In chapter 3, we have identified that one of the most important sources of imperfect information in the software development is the functional requirement specification. In practice, it has proven to be very difficult to define or attain accurate information when it is required. As a result, the requirement specification typically contains vague statements to circumvent the lack of knowledge about the system, which are likely to change in the near or distant future. Modern design methods rely on iteration to correct the changes in the requirements and the advances in insight as the development process moves forward, however as has been identified in chapter 1, incremental design can only correct imperfect information effectively under ideal circumstances.

To address the problems caused by imperfect information, we have proposed to address the imperfection in functional requirement specifications by means of *fuzzy requirements*, which can be used in combination with incremental design. A fuzzy requirement replaces vague and ambiguous descriptions in requirement specifications by a fuzzy set. Each element in the fuzzy set is a possible interpretation of a single ambiguous description, and is attributed with membership values that represent evaluations of stakeholders, such as relevance or urgency. During the subsequent steps, all interpretations are included in the design process as normal requirements, which means the system design fulfills a broadened requirements specification. In the case new insights or changes in requirements occur, the adjustment of the design during an iteration should consist of removing the invalidated interpretations and the associated intermediate design artifacts and components. To avoid that the inclusion of alternative interpretations in the development process leads to an unacceptable increase in the implementation effort, at

any point the design can be streamlined by evaluating requirement combinations based on stakeholder interests.

To facilitate the evaluation and removal of alternative interpretations from fuzzy requirements, we have introduced the Artifact Trace Model. This model describes the relationships between intermediate design artifacts, such as problem definitions or components and classes. By explicitly tracing the decomposition and overlap that result from the design steps, it is possible to determine for each requirement and interpretation which components and/or classes are needed and vice versa. In addition, we have defined an approach to compute stakeholder interest values for the implementation of a given set of components, based on the requirements and interpretations that are satisfied by this implementation. The approach offers decision support to the software engineer by systematically exploring and evaluating all the combinations of alternative interpretations based on the implementation effort and stakeholder interest values. To facilitate a trade-off analysis approach, which can offer decision support from the various views that are of interest to the software engineers and stakeholders, we have defined an optimization approach that evaluates alternative system designs based on the identified optimization criteria, while considering restrictions on other attributes. With the specification of restrictions and optimizations, it possible to define optimizations that represent typical interests, such as a maximization of relevance while not exceeding a certain budget, or the minimization of cost while still having sufficient relevance. The approach is fully configurable with respect to the importance of stakeholder interests, which makes it possible to accurately represent the considerations for the trade-off.

With this approach we have reduced the necessity to completely fulfil the *All-At-Once-Requirement* and the *All-Always-Requirement*, which have been defined in chapter 1. The first requirement stipulated, that for successful iteration all the imperfect information must be removed at the same design moment. This restriction is less stringent when our approach is used, since at every point the imperfection is known and modeled. Therefore the design becomes more flexible, which makes it less vulnerable to future changes. The All-Always-Requirement requires that all the requirements must always describe the same information. With the inclusion of multiple alternative interpretations of vague information, it becomes less likely that the arrival of new insights fall outside the requirement specification. As a result, the requirement specification more closely adheres to the All-Always-Requirement, which facilitates incremental design.

7.6 Supporting Imperfection in Quality Evaluations

In chapter 4 we have identified quality requirements and quality estimations as the second important area, where imperfect information can have a severe influence on the software development process. During the design of software systems, quality requirements are used as a benchmark for the evaluation of candidate solutions. By assessing how well the expected quality attributes adhere to the defined quality requirements, it is possible to evaluate multiple design alternatives and select the one that has the best “quality fit“. In most modern design processes this evaluation and comparison step is not indicated explicitly, although a number of approaches have been proposed as a starting point [Kazman1994] [Kazman1998]. The correctness of the evaluation of design alternatives depends on the accuracy of both the quality requirements as the quality estimations. However, as with the definition of functional requirements, the definition of quality requirements is subject to imperfection due to a lack of knowledge and insights. Even more so, imperfection is intrinsically part of quality estimations, since they are an approximation of the quality values of the resulting system that offer no guarantee about their accuracy. Whenever one of these elements, or both, is misrepresented by seemingly accurate values, the evaluation based on these values can lead to invalid results and wrong design decisions. While the usage of quality requirements and estimations for the assessment of design alternatives infers a considerable risk, there is no alternative method to resolve

design issues. Similar to functional design, adjusting design decisions by means of iterations requires ideal circumstances to be performed effectively.

To address the problems identified in chapter 4, we have extended the expressiveness of quality requirements and estimations with models for describing the imperfection that can be present. Depending on the nature of the imperfection, for example a variance, tolerance, or probabilistic dependency, software engineers can choose from probabilistic, fuzzy set and fuzzy probabilistic models to give an accurate description of the imperfect nature of the available information. The introduction of these models for the specification of imperfect quality requirements and estimations requires the definition of comparison operators to ensure a uniform evaluation of design alternatives. To facilitate this evaluation mechanism, we have defined the concept *degree of fulfilment*. This is a number in the range [0, 1] that indicates the degree to which an estimation satisfies its respective requirement. Using only either zero or one corresponds to the classical crisp case, where one means "completely satisfies" and zero means "completely does not satisfy". The evaluation of imperfect estimations with crisp or imperfect requirements results in a degree of fulfilment that lies between zero and one, and is an indication of the risk that is involved with design alternatives. In chapter 4 we have defined comparison operators for all combinations of imperfection models that result in a uniform description of the degree of acceptance. The operators in chapter 4 for comparing fuzzy requirements and fuzzy estimations are based on triangular fuzzy numbers and semi-trapezoidal fuzzy intervals, but in appendix B a generic approach for its derivation is given.

To complement the imperfection models in chapter 4, we have defined the Design Tree Model, a trace model that records the sequence of the design issues that have been addressed, and the alternatives that have been considered. In addition, for each alternative the quality evaluation is stored, which has full support for imperfection in both requirements and estimations. Our approach provides the software engineer with advice on which alternatives to select, but also on which state the design should roll back to in the case the current design is no longer satisfactory. The Design Tree model supports this approach with configurable strategies, which are aimed at for instance maximization of quality or minimization of development time. With this approach, the influence of three of the requirements for successful iterative design is reduced. The All-At-Once-Requirement and the All-Always-Requirement, as with the Artifact Trace Model approach, is reduced since it is possible to maintain and use imperfect information during the evaluation of design alternatives. This makes it possible to perform reasonable evaluations at most points of the development process. Finally, the restriction of the All-Infinite-Requirement is somewhat contained by the decision support for the selection of design alternatives. Since typically software development is performed with limited resources, optimal usage of these resources is very desirable. With the optimization capabilities of the Design Tree approach, a more effective approach to iterative design can be facilitated.

7.7 Project Scheduling under Probabilistic Market Demands

The third approach that is proposed in this thesis, addresses to presence of imperfection in market demand expectations. Since software systems are being applied in an increasing amount of environments and they become increasingly more complex due the level of sophistication that is required, software projects become very hard to manage. As a consequence of the size of software projects, during release planning and resource scheduling many inputs have to be considered simultaneously. In particular in the field of application frameworks and product lines the implementation schedule is very important, due to the dependencies between reusable assets and actual products.

Since products in such environments are typically assembled from reusable components, it is vital that all these components have been implemented before these products are demanded. This means, that during the planning the expected market demands need to be considered.

However, in chapter 5 we have identified that it very difficult to make a correct production of the market demands in the future, due to the uncertain information on the occurrence of events that influence the demand. In chapter 5, we have proposed an approach that can determine scheduling advice for the implementation of software products based on market demand expectations, component dependencies and resource restrictions.

The imperfection support in this approach, lies in the models that are made of market demand expectations. We propose to model the uncertainty about the market demands in the future by means of probabilities, in a similar manner to, for example, options theory. Market experts, in accordance with software engineers and project managers, typically define a number of outlooks on the market. These outlooks describe market demand situations that potentially can occur in the future. Each of the outlooks is attributed with a probability function, which describes the probability of the demand state's occurrence. The demand states are considered at each point where resources are allocated to components of the application framework. To represents all possible sequences of demand states that can be derived from a given set of outlooks, a graph-based structure called the scenario graph is defined.

The controllable inputs for scheduling optimization lie in the choices that can be made on the allocation of resources. The decision graph, in a manner comparable to the scenario graph, is a compact representation of the resource allocation decisions that can be taken at given points during the development. This graph can be determined based on the estimated implementation effort, dependency structure of the components and products and the available resources. The model is capable of considering restrictions that apply on the choices that can be made, such as the implementation order of components, or a limited availability of resources. In accordance with the time period set by the scenario graph, the decision graph contains the possible productions plans over the same time period. The final step in the resource allocation approach is the evaluation of this information by computing a combined graph of the decision graph and the scenario graph. In this combined graph each production plan is evaluated with each scenario by calculating the expected profit for each decision from the current demand state. The resulting scheduling advice indicates the best action given a demand state and the work completed on the components and products. In addition, it is possible to explore the complete combined graph, to get additional insights into "what-if" scenarios and the consequence of alternative allocations of resources.

The imperfection that is addressed in this approach can not be solved by means of iterative design, since the imperfection is not part of the software designs. Therefore, this approach does not directly resolve one of the requirements for successful iterative design, as defined in chapter 1. Nonetheless, the resource allocation optimization facilitates the optimal usage of resources, which increases the available resources for incremental design steps.

7.8 Discussion

In this thesis, we have introduced three models for the support of imperfect information in software design processes. Each of these models captures the imperfection by means of numerical representations based on probability theory and fuzzy set theory. With these models we have enhanced the capabilities of the software design process and therefore the impact imperfection information can have is reduced. Nonetheless, for each approach we have raised a number of discussion points, to identify the benefits and weaknesses of our approaches. In this section, we examine the overall benefit by discussing the proposed approaches with respect to the following concerns:

Can imperfection in software design processes be avoided?

In each of the three approaches proposed in this thesis, we have defined models that can describe imperfection in the available information. With this type of approach, it is possible to consider the imperfection in the decision making process. If it would be possible to avoid the introduction of imperfection in the software development process, there would be no need for this type of model. However, in chapter 1 we have identified that imperfection is an inherent property of the information that is used in design processes. With a rigorous approach to collecting the information, the amount of imperfection can be minimized, but only in a very limited number of cases can it be avoided. The failure of the waterfall-model, and the general acceptance of iterative and agile approaches is a clear illustration of this fact. We conclude that imperfection can not be avoided during software design. Nonetheless, the iterative correction does not offer a complete solution, due to the All-at-Once, All-Always, All-Isolation and All-Infinite-Requirement, which were identified in chapter 1. Explicit support for imperfection is therefore required to complement iterative design at the inevitable moment that imperfection is encountered.

Do the proposed models describe the relevant properties of imperfect information accordingly?

The imperfection models that have been proposed in this thesis are based on probability theory and fuzzy set theory. It can be questioned how well these models are able to capture the nature of the imperfection that can occur in the information used during software development. The imperfection in the design information, however, does not necessarily always correspond to what can be described by probability and fuzzy set theory. But while it is true that these models do not always reflect the actual nature of imperfection, not acknowledging imperfection or trying to avoid imperfection altogether is worse still. In this case, the consequences of imperfection can solely be corrected by additional iterations, which are only capable of this up to a certain degree. Therefore, while the proposed models certainly do not facilitate all types of imperfect information to be included, they offer a good starting point from which more intricate approaches can be derived in the future.

How do we acquire the applicable definitions for fuzzy sets and probability distributions?

For the proposed approaches, it is very important to define the “right” fuzzy sets and probability distributions to represent the imperfection. The use of imperfection models adds an extra level of detail for expressing information, however the correctness of the results very much depends on the correctness of these inputs. In addition, it should be noted that probability distributions have long been used to model for instance performance of computer systems with a probabilistic nature. Additionally, fuzzy set theory offers the possibility to use linguistic variables [Zadeh1975] to refer to standard definitions of fuzzy sets within a particular area. This can be used to model domain specific information. In this way generic information can be captured and processed by abstracting away from the mathematical definition of the fuzzy sets. Also, from our early experimentations it became clear that the usage of fuzzy sets and probability distributions became quite natural. In particular imperfection like variance and tolerance was easily captured with, for example, triangular fuzzy numbers. The implementation of the toolset also assists greatly in the usage of the models. In addition to facilitating experimentation with specific fuzzy sets and probability distributions, it enables the software engineers to refine the imperfection specifications at later stages. Nonetheless, the usefulness of the results heavily depends on the accuracy and relevance of the inputs. Therefore it is very important to assess the sensitivity of the results to variations and changes in the inputs. While at this time, the sensitivity of the models has not been charted, an empirical evaluation is planned for the future to assess the sensitivity and adjust the models accordingly, if needed.

Will the proposed approaches scale up to industrial settings?

A discussion point that has been raised for all the approaches in the respective chapter is the scalability. The imperfection approaches require additional effort to be performed during the development process. This warrants the question on whether these approaches will scale to industry-size applications. However, the increased effort that is needed for the application of the models is compensated, since it reduces the need for corrective actions that a result from imperfect information. Additionally, the approaches support the systematic exploration of design alternatives, which can reduce the need for expensive optimization activities for the resulting designs. The support of the approaches by set of tools further reduces the required effort. As a result of these considerations, we believe that the extra activities required for the application of this approach will not create overhead that is larger than the gains. The initial results from the pilot study support the supposition that our approach will scale to industrial settings. Nonetheless, in order to properly assess the usability of our approach in an industrial context, an empirical evaluation is required. We have planned an empirical experiment based on the results of the pilot study in this chapter, which should give a more detailed insight into the applicability of our approach in an industrial context.

7.9 Reflection and Future Work

The models that have been proposed in this thesis are the starting point for a new approach to dealing with imperfect information in software development processes. In contrast to modern design processes, we propose to accept imperfect information as an integral part of software design, since it will be inherently present even when it is not always visible. Rather than stabilizing design solely based on incremental design and iterations, our approaches model imperfection and use these models to optimize decision making and minimize the amount of iterations needed. With these models we have taken the first step, but to handle imperfect information in design processes accordingly there are many unanswered question and continuation points. In the following we define the research outline for the continuation and extension of imperfection support during software development.

The most important next step in the research is the validation of the approach by means of empirical experiments. While the results of the conducted pilot study show promising results, it is important that the usability and scalability of our approach is tested in a representative environment. To achieve this, future research should define a uniform manner to compare the results from development processes with and without explicit support for imperfect specifications. In chapter 7, we have defined an outline for empirical validation, but at this point usable metrics for expressing the quality of design processes still need to be defined. An empirical validation provides valuable insights and feedback on the fine tuning of the approach and the toolset implementation.

The second continuation point is the integration of the individual approaches into a large scale decision support model. The Artifact Trace Model and the Design Tree Model intuitively have a clear connection, which at this moment is not defined. Each refinement step in the Artifact Trace Model can be the result of a design issue where multiple alternatives have been considered. In a combination of these approaches it would be possible not only to indicate the design state to roll back to, but also the impact on the design and the implemented alternatives. This should enable the possibility of a trade-off between quality attributes on one side and, for instance, relevance on the other. The integration of the models can further be achieved by exploring how the resource allocation approach can steer the decision making process with respect to probabilistic inputs and complex limited inputs. Since the resource allocation model has a generic definition, it is possible to adjust the approach to other quantifiable decision and reward states, such as refactoring or overall project strategies.

One of the areas where imperfection models in the future can proof their added value is in the modeling of domain information and design heuristics. In recent years, software development methods have been proposed that consider domain information during the design process. While this information generally gives good indications on how to design systems within a particular domain of expertise, it is typically a general steering direction. The information can therefore be considered as reusable imperfect design advice, which makes it a natural candidate to be modeled and considered as such. In a similar manner the design heuristics that exist in a variety of development methods are candidates to be modeled using imperfection models. In particular linguistic variables, as introduced by [Zadeh1975], offer excellent opportunities to capture the imperfect design knowledge in a usable and reusable form, without neglecting the inherent danger of relying on such rules of thumb.

In an ideal picture, the presence of imperfect information in any of the inputs of a software development process do not limit the design capabilities of software engineers. Rather, imperfect information enhances the design possibilities that are available and therefore should be included in the design process up to the point where it is no longer applicable or manageable. With the models proposed in this thesis, we have taken a first step to achieve this. In the long term, research in this area should result in a full scale support of imperfect information from the early conception of requirements until the delivery of the final system. At any time it should also be possible to refine imperfection models that are used in subsequent states, as well as that it should be possible to remove imperfection models in a consistent and usable manner. When this is achieved, the activity of designing software will have come a step closer to maturity, since robust and defensive design starts with a proper understanding and usage of all inputs in the development process, as imperfect as they may be.

REFERENCES

The quotes that are included at the beginning of each chapter originate from the book *Dune*, by *Frank Herbert* [Herbert2005]. I was first introduced to this work at the age of 14 and I have been fascinated with the amazingly rich and intricate universe created in the books ever since. It feels only right therefore, that I pay tribute at this point to Frank Herbert and the Dune universe.

One of the main topics of Dune is related to the topics that have been covered in this thesis. In the Dune universe, a sisterhood called the Bene Gesserit aims to produce the perfect human specimen by means of bloodline manipulation, which has been ongoing for thousands of generations. The main character in the first books of Dune, turns out to be this perfect human and in the end becomes the emperor of the known universe. But even while this perfect being becomes the leader of the universe, society itself becomes far from perfect. The masses start to worship the new emperor and holy wars are started to bring this "faith" to the unknowing.

When this is interpreted in the context of this thesis, it is obvious that searching for perfection is not an answer. It can be very difficult to attain and might not bring what was expected of it initially. Rather, imperfection should be accepted as a natural state and something that one must always be aware of. Ignoring imperfection can result in bad situations, even when it is in the presence of perfection.

[Aksit2001] Aksit, M. & Marcelloni, F. (2001), 'Deferring Elimination of Design Alternatives in Object-Oriented Methods', in *Concurrency and Computation: Practice and Experience*, Wiley, pp. 1247-1279.

[Aksit2001a] Aksit, M. & Marcelloni, F. (2001), 'Leaving Inconsistency Using Fuzzy Logic', in *Information and Software Technology* **43**(10), pp. 725-741.

[Antonsson1996] Antonsson, E. & Otto, K. (1996), 'Improving Engineering Design with Fuzzy Sets', in *Fuzzy Information Engineering*, pp. 633-654.

[Bellman1961] Bellman, R. (1961), *Adaptive Control Processes: A Guided Tour*, Princeton University Press, Princeton, NJ.

[Barbacci1998] Barbacci, M.; Carriere, S.; Feiler, P.; Kazman, R.; Klein, M.; Lipson, H.; Longstaff, T. & Weinstock, C. (1998),'Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis' (ESC-TR-97-029), Technical report, Carnegie Mellon University/ Software Engineering Institute.

[Besnard1989] Besnard, P. (1989), *Introduction to default logic*, Springer-Verlag, Berlin.

[Boehm1986] Boehm, B., (1986), 'A spiral model of software development and enhancement', in *ACM SIGSOFT Software Engineering Notes*, 11, issue 4, pp. 14-24.

[Bonissone1985] Bonissone, P.P. & Tong, R. M. (1985), 'Reasoning with uncertainties in expert systems', in *International Journal of Man Machine Studies*, 22, pp. 241-250.

[Bosc1993] Bosc, P. & Prade, H. (1993), 'An introduction to fuzzy set and possibility theory based approaches to treatment of uncertainty and imprecision in database management systems', in *Proceedings of the 2nd Workshop on Uncertainty Management in Information Systems: From Needs to Solutions*, Catalina, CA.

[Bowen1990] Bowen, J.; O'Grady, P. & Smith, L. (1990), 'A Constraint Programming Language for Life-Cycle Engineering', in *Artificial Intelligence in Engineering*, Computational Mechanics Publications, pp. 206-220.

[Brown1991] Brown, P.G. (1991). 'QFD: Echoing the Voice of the Customer', in *AT&T Technical Journal*, March/April, pp. 21-31.

[Bubenko1994] Bubenko, J.; Rolland, C.; Loucopoulos, P. & Antonellis, V.D. (1994), 'Facilitating "Fuzzy to Formal" Requirements Modelling', in *Proceedings IEEE International Conference on Requirements Engineering*, IEEE Publishing.

[Buckley2003] Buckley, J.J. (2003), *Fuzzy Probabilities New approach and applications*, Springer Verlag.

[Carver2003] Carver, J.; Jaccheri, L.; Morasca, S. & Shull, F. (2003) 'Using Empirical Studies during Software Courses', in (eds.) Reidar Conradi and Alf Inge Wang *Lecture Notes on Computer Science*, Springer-Verlag Heidelberg.

[Chung1992] Chung, S.; Hanson, F. & Xu, H. (1992), 'Parallel stochastic dynamic programming: finite elements methods', in *Linear Algebra and Applications*, 172, pp. 197-218.

[Clements2004] Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R. & Stafford, J. (2004), in *Documenting Software Architectures*, Addison Wesley.

[Cmitile1992] Cimitile, A.; Lanubile, F. & Visaggio, G. (1992), 'Traceability Based on Design Decisions', in *Proceedings of Conference on Software Maintenance*, IEEE Press, pp. 309-317.

[Davis1990] Davis, A.M. (1990). 'Software Requirements: Analysis and Specification', Prentice-Hall, Inc.

[Egyed2006] Egyed, A. & Wile, D.S. (2006), 'Support for Managing Design-Time Decisions', in *IEEE Transactions on Software Engineering* **32**(5), pp. 299-314.

[Fayad1999] Fayad, M. E.; Schmidt, D. C. & Johnson, R. E. (1999), *Building Application Frameworks*, Wiley Computer Publishing, ISBN: 0-471-24875-4.

[Finkelstein1994] Finkelstein, A.; Kramer, J. & Nuseibeh, B.Finkelstein, A.; Kramer, J. & Nuseibeh, B. (ed.) (1994), *Software process modelling and technology*, Research Studies Press Ltd.

[Forman2001] Forman, E. & Sally, M. (2001), *Decision by Objectives*, World Scientific, ISBN: 981-02-4142-9.

[Foulds1984] Foulds, R. L. (1984), *Combinatorial Optimization for Undergraduates*, Springer, New York, ISBN: 0-387-90977-X.

[Fudenberg1991] Fudenberg, D. & Tirole, J. (1991), *Game Theory*, MIT Press, 1991, ISBN 0-262-06141-4.

[Gray1998] Gray, A. R. & MacDonnel, S. G.. (1998), 'Fuzzy Logic Techniques for Software Metrics Models of Development Effort', in Pedrycz, W. & Peters, J. F. (eds) *Computational Intelligence in Software Engineering*, World Scientific Publishing, 981-02-3503-8.

[Gregoriades2005] Gregoriades, A. & Sutcliffe, A. (2005), 'Scenario-Based Assessment of Nonfunctional Requirements', in *IEEE Transactions on Software Engineering*, IEEE Computer Society, pp. 392-409.

[Haykin1998] Haykin, S. (1998), *Neural Networks: A comprehensive foundation*, Prentice Hall.

[Herbert2005] Herbert, F. (2005), *The Great Dune Trilogy*, Gollancz, ISBN-10: 0575070706, ISBN-13: 978-0575070707

[Jacobson1999] Jacobson, I.; Booch, G. & Rumbough, J. (1999), *The Unified Software Development Process*, Addison Wesley.

[Kaindl1993] Kaindl, H. (1993), 'The missing link in requirements engineering', in *SIGSOFT Softw. Eng. Notes* **18**(2), pp. 30-39.

[Kaiser1994] Kaiser, G.E.; Popovich, S. & Ben-Shaul, I.Z. (1994), 'A Bi-Level Language for Software Process Modeling', in Walter Tichy (ed.), *Configuration Management*, John Wiley and Sons, Ltd.Baffins Lane, Chichester, West Sussex PO19 1UD, England, pp. 39-72.

[Kang1990] Kang, K.; Cohen, S.; Hess, J.; Novak, W. & Peterson, A. (1990), 'A Feature Oriented Domain Analisys (FODA) Feasability Study', CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

[Karlsson1997] Karlsson, J. & Ryan, K. (1997), 'Prioritizing Requirements Using a Cost-Value Approach', in *IEEE Software*, September/October issue, pp. 67-74.

[Karolak1995] Karolak, D. W. (1995), *Software Engineering Risk Management*, Wiley, ISBN: 978-0-8186-7194-4.

[Kazman1994] Kazman, R.; Bass, L.; Abowd, G. & Webb, M. (1994), 'SAAM: a method for analyzing the properties of software architectures', in *Proceedings of 16th International Conference on Software Engineering*, IEEE, pp. 81-90.

[Kazman1998] Kazman, R.; Klein, M.; Barbacci, M.; Longstaff, T.; Lipson, H. & Carriere, J. (1998), 'The Architecture Tradeoff Analysis Method', in *4th Int'l Conference on Engineering of Complex Computer Systems*, IEEE Computer Society Press, pp. 68-78.

[Klir1995] Klir, G.J. & Yuan, B. (1995), *Fuzzy Sets and Fuzzy Logic*, Prentice Hall.

[Kniesel2002] Kniesel, G.; Noppen, J.; Mens, T. & Buckley, J. (2002), 'The First International Workshop on Unanticipated Software Evolution', in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 92-106.

[Kroll2003] Kroll, P. & Kruchten, P. (2003), *The Rational Unified Process Made Easy*, Addison Wesley, ISBN: 0-321-16609-4.

[Kruchten1999] Kruchten, P. (1999), *The Rational Unified Process - An Introduction*, Addison Wesley, ISBN: 0-201-60459-0.

[Larson1965] Larson, R. (1965), 'Dynamic programming with reduced computational requirements', in *IEEE Transactions on Automatic Control*, AC-14, pp. 135-143.

[Larson1968] Larson, R. (1968), *State Increment Dynamic Programming*, American Elsevier, New York, NY.

[Law1995] Law, W.S. & Antonsson, E.K. (1995), 'Optimization Methods for Calculating Design Imprecision', in *Advances in Design Automation*, ASME, pp. 471-476.

[Lee1991] Lee, J. & Lai, K. (1991), 'What's in design rationale'. in *Human-Computer Interaction*, 6(3–4), pp. 251-280

[Lee2003] Lee, J.; Kuo, J.; Hsueh, N. & Fanjiang, Y. (2003), 'Trade-off requirement engineering', in Jonathan Lee (ed.) *Software Engineering with Computational Intelligence*, Springer, pp. 51-72.

[Lethbridge2005] Lethbridge, T.C. & Laganière, R. (2005), *Object-Oriented Software Engineering Practical Software Development using UML and Java*, McGraw Hill.

[Liu2005] Liu, X. & Da, Q. (2005), 'A Decision Tree Solution Considering the Decision Maker's Attitude', in *Fuzzy Sets and Systems*, Elsevier, pp. 437-454.

[Mamdami1981] Mamdami, E. H. & Gaines, B. R. (eds.), (1981), *Fuzzy reasoning and its applications*, Academic Press, New York.

[Marcelloni1999] Marcelloni, F. & Aksit, M. (1999), 'Reducing Quantization Error and Contextual Bias Problems in Software Development Processes by Applying Fuzzy Logic', in *Proceedings 18th Int. Conference of NAFIPS*, IEEE, ISBN 0-7803-5211-4.

[Martin2003] Martin, R.C. (2003), *Agile Software Development*, Prentice Hall, ISBN: 0-13-597444-5.

[Mayrhauser1998] Von MayrHauser, A., Anderson, C. W., Chen, T., Mraz, R. & Gideon, C. A. (1998), 'On the promise of neural networks to support software testing', in Pedrycz, W. & Peters, J. F. (eds) *Computational Intelligence in Software Engineering*, World Scientific Publishing, 981-02-3503-8.

[McCall1991] McCall, R. J. (1991), 'PHI: A conceptual foundation for design hypermedia', in *Design Studies*, 12(1), pp. 30-41.

[McCarthy1986] McCarthy, J. (1986), 'Applications of circumscription to formalising commonsense knowledge', in *Artificial Intelligence*, 28, pp 89-116.

[McCarthy1980] McCarthy, J. (1980), 'Circumscription - a form of non-monotonic reasoning', in *Artificial Intelligence*, 13, pp 27-39.

[Molter1999] Molter, G. (1999), 'Integrating SAAM in Domain-Centric and Reuse-Based Development Processes', in *Proceedings of the Second Nordic Workshop on Software Architecture*, pp. 1103-1581.

[Moore1985] Moore, R. C. (1985), 'Semantical considerations in nonmonotonic logic', in *Artificial Intelligence*, 25, pp 75-94.

[Nilsson1986] Nilsson, N. (1986), 'Probabilistic logic', in *Artificial Intelligence*, 28, pp. 71-87.

[Noppen2004] Noppen, J.; Broek, P. van den & Aksit, M. (2004), 'Dealing with Fuzzy Information in Software Design Methods', in Scott Dick; Marek Reformat; Lukasz Kurgan; Petr Musilek & Witold Perdycz

(ed.) *Proceedings of the 2004 Annual Meeting of the North American Fuzzy Information Processing Society*, IEEE, IEEE Operations Center 445 Hoes Lane Piscataway, NJ, 08854-4150 USA, pp. 22-27.

[Noppen2004a] Noppen, J.; Aksit, M.; Nicola, V. & Tekinerdogan, B. (2004), 'Market-Driven Approach Based on Markov Decision Theory for Optimal Use of Resources in Software Development', in *IEE Proceedings Software* **151**(2), pp. 85-94.

[Noppen2005] Noppen, J.; Broek, P. van den & Aksit, M. (2005), 'Dealing with Imprecise Quality Factors in Software Design', in Barry Boehm; Sunita Chulani; June Verner & Bernard Wong (ed.) *Proceedings of the Third Workshop on Software Quality*, pp. 40-45.

[Noppen2005a] Noppen, J.; Broek, P.van den & Aksit, M. (2005), 'A Model for Quality Optimization in Software Design Processes', in *Proceedings of Net.Objectdays 2005, 6th International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, TranSIT GmbH, pp. 529-541.

[Noppen2007] Noppen, J.; Broek, P. van den & Aksit, M. (2007), 'Imperfect Requirements in Software Development', in *Proceedings of Requirements Engineering: Foundations for Software Quality 2007*, LNCS 4542, Springer, pp. 247-261.

[Noppen2007a] Noppen, J.; Broek, P. van den & Aksit, M. (2007), 'Software Development with Imperfect Information', in *Soft Computing: Special Issue on Software Engineering and Soft Computing*, Springer, to appear.

[Osterweil1997] Osterweil, L. (1997), 'Software Processes are software too, revisited: An invited talk on the most influential paper of ICSE 9', in *Proceedings of 19th International Conference on Software Engineering*, IEEE Press, pp. 2-13, Monterey, CA.

[Osterweil1998] Osterweil, L. (1998), 'Architecting Processes are key to software quality', International Workshop on the Role of Software Architecture in Testing and Analysis, Marsala, Italy.

[Parnas1976] Parnas, D. (1976), 'On the Design and Development of Program Families', in *IEEE Transactions on Software Engineering*, Vol. SE-2, 1, pp. 1-9

[Parson1996] Parsons, S. (1996), 'Current Approaches to Handling Imperfect Information in Data and Knowledge Bases', in *IEEE Transactions on Knowledge and Data Engineering*, Vol. 8, 3, pp. 353-72.

[Pedrycz1998] Pedrycz, W. & Sosnowski (1998), Z. A., 'FOOD: Towards Fuzzy Object-Oriented Systems Design', in Pedrycz, W. & Peters, J. F. (eds) *Computational Intelligence in Software Engineering*, World Scientific Publishing, 981-02-3503-8.

[Pedrycz1999] Pedrycz, W., Peters, J. F., Ramanna, S. (1999), 'A Fuzzy Set Approach to Cost Estimation of Software Projects', in *Proceedings of the 1999 Canadian Conference on Electrical and Computer Engineering*, IEEE Press, pp. 1068-1073.

[Pedrycz2002] Pedrycz, W. (2002), 'Computational Intelligence as an Emerging Paradigm of Software Engineering', in *Proceedings of the 14th international conference on Software engineering and knowledge engineer ing*, ACM Press, New York, pp. 7-14.

[Pfleeger1998] Pfleeger, S. L. (1998), *Software Engineering, Theory and Practice*, Prentice Hall, New Jersey, ISBN: 0-13-624842-X

[Podorozhny1999] Podorozhny, R. Staudt Lerner, B. Osterweil, L. (1999), 'A rigorous approach to resource management in activity coordination', University of Massachusetts, Amherst MA, 01003, USA.

[Pohl2005] Pohl, K.; Böckle, G. & van der Linden, F.J. (2005), *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag New York, Inc.

[Poppendieck2003] Poppendieck, M. & Poppendieck, T. (2003), *Lean Software Development*, Addison Wesley, ISBN: 0-321-15078-3.

[Potts1988] Potts, C. & Bruns, G. (1988), 'Recording the reasons for design decisions', in *ICSE '88: Proceedings of the 10th international conference on Software engineering*, IEEE Computer Society PressLos Alamitos, CA, USA, pp. 418-427.

[Poundstone1993] Poundstone, W. (1993), *Prisoner's Dilemma*, Anchor Books, Double day, New York, ISBN: 038541580X.

[Power2002] Power, D. J. (2002), *Decision support systems: concepts and resources for managers*, Westport, Conn., Quorum Books.

[Pressman1997] Pressman, R. S. (1997), *Software Engineering, A practicioner's Approach*, McGraw-Hill, New York, ISBN: 0-07-709411-5.

[Ramesh2001] Ramesh, B. & Jarke, M. (2001), 'Toward Reference Models of Requirements Traceability', in *Software Engineering* **27**(1), pp. 58-93.

[Ran1996] Ran, A. & Kuusela, J. (1996), 'Design Decision Trees', in *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, IEEE Computer SocietyWashington, DC, USA, pp. 172-175.

[Reformat2002] Reformat, M., Pedrycz, W., Pizzi, N. J. (2002), 'Software Quality Analysis with the use of Computational Intelligence', in *Proceedings of the 2002 IEEE International Conference on Fuzzy Systems*, Vol. 2, IEEE Press, Washington, pp. 1156-1161.

[Regli2000] Regli, W.C., Hu, X., Atwood, M. & Sun, W. (2000), 'A Survey of Design Rationale Systems: Approaches, Representation, Capture and Retrieval', in *Engineering with Computers* 16, pp. 209-235, Springer-Verlag London Limited.

[Regnell1992] Regnell, B.; Karlsson, L. & Höst, M. (2002), 'An Analytical Model of Requirements Selection Quality in Software Product Development', in *Second Conference on Software Engineering Research and Practise*, Blekinge Institute of Technology, Karlskrona, Sweden.

[Royce1970] Royce, W. (1970), 'Managing the Development of Large Software Systems', in *Proceedings of IEEE WESCON*, 26, pp. 1-9.

[Ruhe2004] Ruhe, G. (2004), 'Software Engineering Decision Support', in *iCORE Research Report*, 3.

[Russel1995] Russel, S. & Norvig, P. (1995), *Artifical Intelligence A Modern Approach*, Prentice Hall, ISBN: 0131038052.

[Salo1997] Salo, A. & Hämäläinen, R. (1997), 'On the measurement of preferences in the analytic hierarchy process', in *Journal of multi-criteria decision analysis*, vol 6, pp. 309-319.

[Shaw1996] Shaw, M. (1996),'Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does', in *Proc. 8th International Workshop on Software Specification and Design*.

[Smithers1991] Smithers, T., Tang, M.X. & Tomes, N. (1991). 'The Maintenance of Design History in AI-Based Design', in *Proceedings of the Colloquium by the Institution of Electrical Engineers Professional Group C1 (Software Engineers)*, London, pp. 8/1- 8/3.

[SPOT2007] Web address for the SPOT Toolset: http://www.cs.utwente.nl/~noppen/public/spot.zip

[Stoelinga2002] Stoelinga, M. (2002), 'An introduction to probabilistic automata', in G. Rozenberg (ed.) *EATCS bulletin*, pp. 1-23.

[Tekinerdogan2000] Tekinerdogan, B. (2000), *Synthesis-Based Software Architecture Design*, Ph. D. Thesis, Print Partners Ipskamp, Enschede, ISBN 90-365-1430-4, Also available through http://www.cs.bilkent.edu.tr/~bedir/PhDThesis/index.htm.

[Tekinerdogan2002] Tekinerdogan, B. & Aksit, M. (2002), 'Classifying and evaluating architecture design methods', in Mehmet Aksit (ed.) *Software Architecture and Component Technology*, Kluwer Academic Publishers, pp. 3-28.

[Tretmans2001] Tretmans, J.; Wijbrans, K. & Chaudron, M. (2001), 'Software Engineering with Formal Methods: The Development of a Storm Surge Barrier Control System Revisiting Seven Myths of Formal Methods', in *Form. Methods Syst. Des.* **19**(2), pp. 195-215.

[Voß2003] Voß, S. & Woodruff, D.L. (2003), *Introduction to computational optimization models for production planning in a supply chain*, Springer Verlag.

[Yen1993] Yen, J. & Lee, J. (1993), 'Fuzzy Logic as a Basis for Specifying Imprecise Requirements', in *Proceedings of 2nd IEEE International Conference on Fuzzy Systems (FUZZ-IEEE'93)*, IEEE Computer Society, pp. 745-749.

[Yourdon1979] Yourdon, E. & Constantine, L.L. (1979), *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall.

[Zadeh1965] Zadeh, L.A. (1965), 'Fuzzy Sets', in *Information and Control*, 8(3), pp. 338-353

[Zadeh1975] Zadeh, L.A. (1975), 'The concept of a linguistic variable and its application to approximate reasoning - II.', in *Inf. Sci.* **8**(4), pp. 301-357.

[Zadeh1978] Zadeh, L. A., (1978), 'Fuzzy sets as the basis for a theory of possibility', in *Fuzzy Sets and Systems*, 1, pp. 1-27.

[Zadeh1983] Zadeh, L. A., (1983), 'Commonsense knowledge representation based on fuzzy logic', in *IEEE Computer*, October, pp. 61-65.

[Zadeh1983a] Zadeh, L. A., (1983), 'The role of fuzzy logic in the management of uncertainty in expert systems', in *Fuzzy Sets and Systems*, 11, pp. 199-227.

APPENDIX A - REFINEMENT STEPS OF THE TMS

This appendix describes the refinement steps for the Traffic Management System example, used in chapter 3 to demonstrate the Artifact Trace Model. The tables in this appendix indicate the refinement from the initial requirements until the final components in three steps, for both the crisp requirement and the fuzzy requirement case. The initial requirement specification and the alternative interpretations have been included in the respective sections.

A.1 Artifact Refinement for Crisp Requirements

In the Traffic Management System example, six functional requirements have been specified, based on the description provided by the stakeholders. In chapter 3, the six requirements are described as follows:

1 *The TMS must support displaying relevant information to the users of the TMS*
2 *There should be an explicit, convenient model of tasks and scenarios*
3 *The system must support action coordination for optimal normalization of traffic flow*
4 *The system should support task allocation*
5 *Contextual Information should be accessible*
6 *The TMS should be able to communicate with the roadside system*

As has been identified in chapter 3, the requirement specification contains considerable imperfection. To compare the design that result from using crisp requirements and fuzzy requirements, the Artifact Trace Model is used for both cases. For the example, the requirements are refined to components in three steps, which are described in this appendix. In Table A.1 the refinement step from requirement to problems is given, while using the initial requirement specification as crisp requirements.

Table A.1 From Requirements to Problems

Requirement	Problems to be solved
1	P1 How do we display information P6.1
2	P2.1 How do we express Tasks and Scenarios in an extensible manner? P2.2 How do we capture Task and Scenarios in a portable and exportable manner?
3	P3.1 How do we normalize traffic flow with actions? P3.2 How do we rate normalizations with respect to each other?
4	P2.1 P4.1 How do we support a generic Task Allocation Support Model? P4.2 How do we offer this information?
5	P5.1 How do we support interaction with the system? P5.2 How do we define a generic model that captures contextual information for external usage?
6	P6.1 How do we make the internal data available? P6.2 How do we realize a constant and stable communication stream?

In the first refinement step, the requirements are refined to a set of problems that need to be solved to implement the requirements. In the table it can be seen that requirement 1 and 4 reuse problems from the other requirements to define their problem set. While the identified problems in this table do not describe a complete set of problems, the overlap and amount is satisfactory for illustrational purposes.

Table A.2 From Problems to Solutions

Problem	Solution
P1	S1 Displaying by interpretation and formatting for the affected user
P2.1	S2.1 XML Schema for Tasks and Scenarios
P2.2	S2.2.1 State and Scenario Models based on Language Constructs S2.2.2 XML based Language Parser
P3.1	S3.1 Determine and execute traffic relocation strategies
P3.2	S3.2 Compare strategies based on completion time and congestion reduction
P4.1	S4.1 Task Allocation based on XML models
P4.2	S4.2.1 Open Source XML Parser S4.2.2 XML Communication Component
P5.1	S5.1.1 Corba based MiddleWare S5.1.2 SQL Query Component
P5.2	S5.2 Database + Standardized Database Content Output
P6.1	S6.1 Uniform Communication Interface
P6.2	S6.2.1 Video streaming support S6.2.2 Corba based Communication S6.1

In Table A.2 the identified problems for the crisp requirements are refined to a set of solutions that should be provided to solve the respective problems. Again, there is reuse of solutions among the problems, which is used in the Artifact Trace Model optimization to minimize the implementation effort. The final step of the design process is the refinement of solutions to the components that implement them. This step is given in Table A.3.

Table A.3 From Solutions to Components

Solution	Components	Cost
S1	I Definable Views on Traffic Data Component	3
S2.1	II XML Schema for Tasks and Scenarios III Common File Format Definition	3 0.5
S2.2.1	IV State and Scenario Models in specific language	1
S2.2.2	V Custom Language Parser Component III	3.5
S3.1	VI Relocation Strategy Component	2
S3.2	VII Strategies Comparison and Selection Component	2
S4.1	VIII XML Schema for Task Allocation	1
S4.2.1	IX Open Source XML Parser Component	4
S4.2.2	X XML Communication Component	0.5
S5.1.1	XI Corba Communication Components	1
S5.1.2	XII SQL Query Component	0.1
S5.2	XIII Database + Database Serializer Component	0.5
S6.1	XIV Uniform Communication Interface	3
S6.2.1	XV Dynamic Protocol Support Component XVI Video Streaming Support Component	2 2
S6.2.2	XVII Corba based Communication Component	4

In this table, for each solution the components are given that are needed for their implementation. Each of the components is numbered with a roman numeral, and in the right column the time needed for their implementation is given. With this step, the design process of the Traffic Management System is completed. The analysis and optimization results for this design can be found in chapter 3.

A.2 Artifact Refinement for Fuzzy Requirements

The second step is chapter 3 consists of the design of the Traffic Management System by using a fuzzy requirement specification rather than a crisp specification. For this purpose, in chapter 3 a number of interpretations are identified for four of the requirements from the initial specification. In Table A.4 the fuzzy requirement specification that is the starting point for the design process is given.

Table A.4 Interpretations of Imperfect Requirements

Requirement		Membership
1	The TMS must support displaying relevant information to the users of the TMS	1
2	There should be an explicit, convenient model of tasks and scenarios	
2.1	There should be an easily extensible model of tasks and scenarios	0.8
2.2	There should be an easily understandable model of tasks and scenarios	0.9
2.3	There should be an easily exportable and portable model of tasks and scenarios	0.6
3	The system must support action coordination for optimal normalization of traffic flow	1
4	The system should support task allocation	
4.1	The system should support user extensible task allocation profiles	0.6
4.2	The system should support task allocation as individual task blocks	0.2
4.3	The system should support task allocation with automated decision support	0.9
5	Contextual Information should be accessible	
5.1	Contextual Information should be accessible internally in a generic format	0.7
5.2	Contextual Information should be accessible externally at an interface in a generic format	0.5
5.3	Contextual Information should be accessible both internally and externally at an inter-face in a generic format	0.3
6	The TMS should be able to communicate with the roadside system	
6.1	The Traffic Management System should be able to communicate with the roadside system unidirectionally	0.3
6.2	The Traffic Management System should be able to communicate with the roadside system with flexible support for separate data formats	0.6
6.3	The Traffic Management System should be able to communicate with the roadside system for realtime video	0.8

In this table the requirement 2, 4, 5 and 6 have been identified to contain imperfection, and for each of these requirements three alternative interpretations have been identified. In the rightmost column the membership value for each interpretation is given.

Table A.5 From Requirements to Problems

Req.	Problems to be solved
1	P1.1 How do we display information? P6.1.2
2.1	P2.1.1 How do we support a generic model that captures tasks and scenarios? P2.1.2 How do we express Tasks and Scenarios in an extensible manner?
2.2	P2.2.1 How do we capture tasks and scenarios in an easily understandable manner? P2.2.2 How do we support Tasks and Scenarios while maintaining system performance?
2.3	P2.3.1 How do we capture Tasks and Scenarios in a portable and exportable manner? P2.1.2
3	P3.1 How do we normalize traffic flow with actions? P3.2 How do we rate normalizations with respect to each other?
4.1	P4.1.1 How do we support a generic Task Allocation Support Model? P4.1.2 How do we offer this information? P2.1.2
4.2	P4.2.1 How do we offer a highly composable Task Allocation Support Model? P4.2.2 How do we extract the information from the model P4.1.2
4.3	P4.3.1 How do we provide reasoning support for Task Allocation? P4.3.2 How do we extract this information from the Reasoning System? P4.1.2
5.1	P5.1.1 How do we define a generic model that captures contextual information for internal usage? P5.1.2 How do we make this generic model available inside the system?
5.2	P5.2.1 How do we support interaction with the system? P5.2.2 How do we define a generic model that captures contextual information for external usage?
5.3	P5.1.2, P5.2.1 P5.3.1 How do we define a generic model that captures contextual information for internal and external usage?
6.1	P6.1.1 How do we realize the unidirectional communication? P6.1.2 How do we make the internal data available?
6.2	P6.2.1 How do we achieve dynamic switching of communication protocols? P6.1.2
6.3	P6.3.1 How do we realize a constant and stable communication stream? P6.1.2

In Table A.5 the refinement of the fuzzy requirement specification to a set of problems is given. It can be seen that at this point the interpretations of fuzzy requirements are interpreted as crisp requirements, without considering their membership values. This will only become applicable in the optimization process (see chapter 3).

Table A.6 From Problems to Solutions

Problem	Solution
P1.1	S1.1 Displaying by interpretation and formatting for the affected user
P2.1.1	S5.1.1
P2.1.2	S2.1.2 XML Schema for Tasks and Scenarios
P2.2.1	S2.2.1 State and Scenario Models based on StateMachines
P2.2.2	S2.2.2 State Machine Interpreter
P2.3.1	$S2.3.1_1$ State and Scenario Models based on Language Constructs $S2.3.1_2$ XML based Language Parser
P3.1	S3.1 Determine and execute traffic relocation strategies
P3.2	S3.2 Compare strategies based on completion time and congestion reduction
P4.1.1	S4.1.1 Task Allocation based on XML models
P4.1.2	S5.1.2, $S5.1.1_2$
P4.2.1	S4.2.1 Task Allocation based on Object Oriented models
P4.2.2	S4.2.2 COM+ Component S5.2.1
P4.3.1	S4.3.1 Task Allocation based Expert Systems
P4.3.2	S4.3.2 Text based Allocation report
P5.1.1	$S5.1.1_1$ XML-based Model for capturing contextual information $S5.1.1_2$ Open Source XML Parser
P5.1.2	S5.1.2 XML Communication Component
P5.2.1	$S5.2.1_1$ Corba based MiddleWare $S5.2.1_2$ SQL Query Component
P5.2.2	S5.2.2 Database + Standardized Database Content Output
P5.3.1	S5.3.1 XML Model + Database Representation S5.1.1, S5.2.1, S5.2.2
P6.1.1	S6.1.1 Corba based Communication
P6.1.2	S6.1.2 Uniform Communication Interface
P6.2.1	S6.2.1 Dynamic Protocol Support S6.1.1, S6.1.2
P6.3.1	S6.3.1 Video streaming support S6.1.1, S6.1.2

Table A.6 describes the refinement of the problems in the previous table to a set of solution techniques. Since the amount of problems and solutions is considerably higher than the crisp case, the possibilities for reuse also increases, which is reflected in the table.

Table A.7 From Solutions to Components

Solution	Components	Cost
S1.1	I Definable Views on Traffic Data Component	3
S2.1.2	II XML Schema for Tasks and Scenarios XV	3
S2.2.1	III State and Scenario Models based on StateMachines XV	1
S2.2.2	IV State Machine Interpreter Component	4
$S2.3.1_1$	V State and Scenario Models in specific language	1
$S2.3.1_2$	VI Custom Language Parser Component XV	1
S3.1	VII Relocation Strategy Component	2
S3.2	VIII Strategies Comparison and Selection Component	2
S4.1.1	IX XML Schema for Task Allocation	3.5
S4.2.1	X Object Oriented Task Allocation Model	2
S4.2.2	XI COM+ Component	3
S4.3.1	XII Task Allocation Expert System	2
S4.3.2	XIII Text based Allocation report extractor and interface XV	3
$S5.1.1_1$	XIV XML Model Schema XV Common File Format Definition	2 0.5
$S5.1.1_2$	XVI Open Source XML Parser Component	4
S5.1.2	XVII XML Communication Component	0.5
$S5.2.1_1$	XVIII Corba Communication Components	1
$S5.2.1_2$	XIX SQL Query Component	0.1
S5.2.2	XX Database + Database Serializer Component	0.5
S5.3.1	XXI XML Schema and ER Diagram	1
S6.1.1	XXII Corba based Communication Component	4
S6.1.2	XXIII Uniform Communication Interface	3
S6.2.1	XXIV Dynamic Protocol Support Component	2
S6.3.1	XXV Video Streaming Support Component XXIV	2

In Table A.7 the final design step is described, in which the solutions from Table A.6 are refined to sets of components that implement them. Again the components are numbered with roman numerals, and the relevance value is indicated in the rightmost column.

APPENDIX B - DERIVATION OF THE COMPARISON OPERATORS

This appendix describes the derivation of the comparison operators for fuzzy quality requirements and fuzzy quality estimations presented in chapter 4. The derivation is done for requirements represented by semi-trapezoidal fuzzy intervals and estimations represented by triangular fuzzy numbers. The use of these comparison operators is described in chapter 4.

B.1 Derivation of the Comparison Operator for Fuzzy Sets

The operation on fuzzy sets which is needed in this thesis is the comparison of a fuzzy estimation *C* with a fuzzy requirement *A*. The result of the comparison should be a number between zero and one, indicating the degree to which A is smaller than C. The comparison operator for this operation is based on comparing the α–cuts of C to the α–cuts of A.

For 0<α<=1, the α-cut of a fuzzy set *F* with membership function μ is defined by $\{x \mid \mu(x) \geq \alpha \}$ and denoted by F[α]. If F is a fuzzy number or a fuzzy interval, then F[α] is an interval. The degree of acceptance for C with respect A is defined to be:

$$\int_0^1 S(\alpha)\,d\alpha$$

where S(α) is the degree to which A[α] is smaller than C[α]. S(α) is defined to be the fraction of C[α] which belongs to A[α]. With C[α] = [q(α), r(α)] and A[α] =]–∞ , p(α)] we find:

$$S(\alpha) = 0 \qquad , \text{if } r(\alpha) \leq p(\alpha)$$

$$S(\alpha) = 1 \qquad , \text{if } p(\alpha) \leq q(\alpha)$$

$$S(\alpha) = \frac{p(\alpha) - q(\alpha)}{r(\alpha) - q(\alpha)} \qquad , \text{if } q(\alpha) < p(\alpha) < r(\alpha)$$

In this thesis fuzzy estimations are assumed to be triangular fuzzy numbers (c_1, c_2, c_3), and requirements are assumed to be semi-trapezoidal fuzzy numbers ($-\infty$, a_1, a_2), as can be seen in Figure B.1.

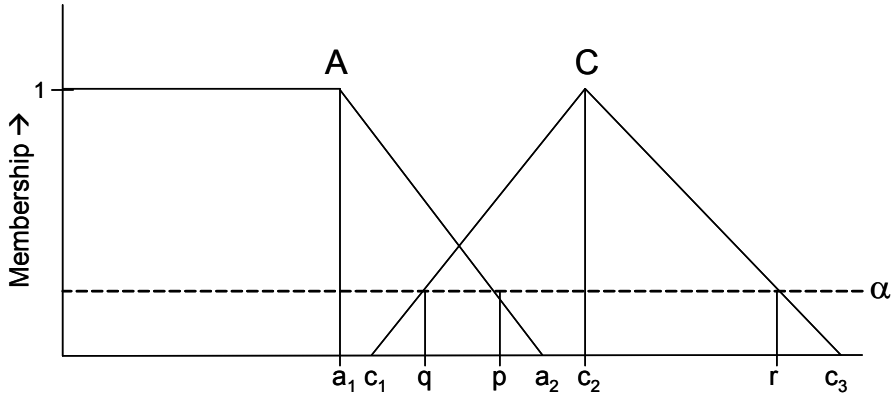


Figure B.1 Fuzzy Numbers

In this case, p(α), q(α) and r(α) are given by:

$$p(\alpha) = a_2 - \alpha(a_2 - a_1)$$
$$q(\alpha) = c_1 - \alpha(c_2 - c_1)$$
$$r(\alpha) = c_3 - \alpha(c_3 - c_2)$$

Let the fuzzy estimation be the triangular fuzzy number (c_1, c_2, c_3) and the fuzzy requirement interval be the semi-trapezoidal fuzzy number (a_1, a_2). We define q_1 and q_2 as follows:

$$q_1 = \frac{a_2 - c_1}{a_2 - a_1 + c_2 - c_1} \qquad q_2 = \frac{a_2 - c_3}{a_2 - a_1 + c_3 - c_2}$$

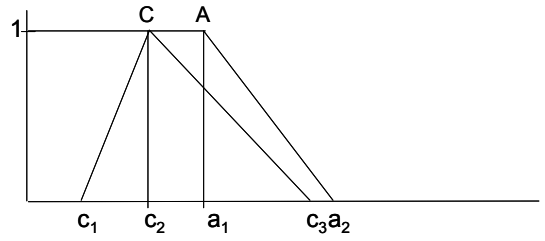
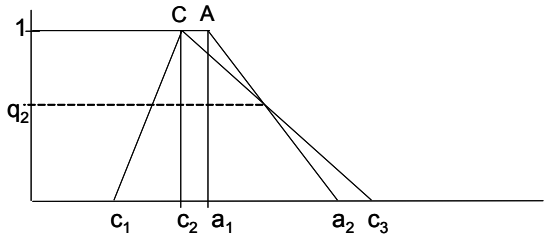
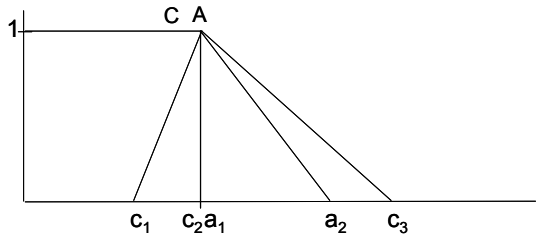
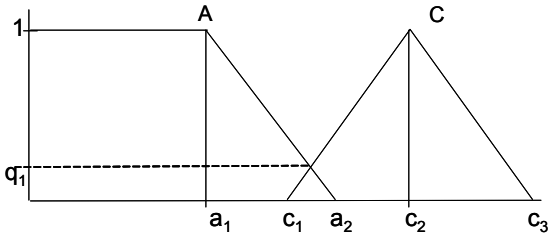
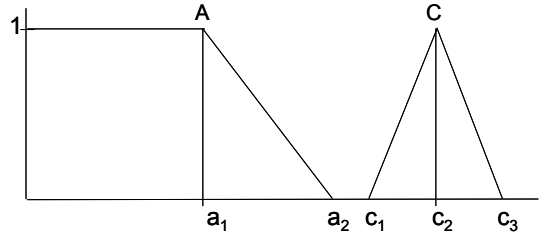
Then q_1 is the α-height of the intersection of the lefthand side of the estimation with the requirement, and q_2 is the α-height of the intersection of the righthand side of the estimation with the requirement.

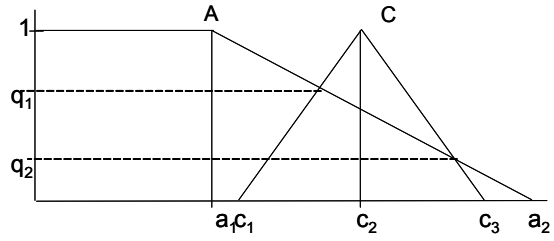
S(α) can now be defined for six cases of comparing a requirement (a_1, a_2) with an estimation (c_1, c_2, c_3). The value of *Comp((a_1, a_2), (c_1, c_2, c_3))* is given in Table B.1, which corresponds to the degree of acceptance of the fuzzy estimation and the fuzzy requirement. In the table for each of the six cases a graphical depiction of the actual situation is given in the left column. In the right column, for each situation S is defined for height α above the line, and

$$Comp((a1, a2), (c1, c2, c3)) = \int_0^1 S(\alpha)\,d\alpha$$

is given below the line.

Table B.1 Comparing fuzzy requirements with fuzzy estimations

$c_2 \le a_1$ & $c_3 \le a_2$	
	$S(\alpha) = 1$, for $0 \le \alpha \le 1$
	$Comp((a_1, a_2), (c_1, c_2, c_3)) = 1$
$c_2 < a_1$ & $c_3 > a_2$	
	$\frac{a_2 - c_1 - \alpha(a_2 - a_1 + c_2 - c_1)}{c_3 - c_1 - \alpha(c_3 - c_1)}$, for $0 \le \alpha \le q_2$ 1 , for $q_2 \le \alpha \le 1$
	$\frac{a_2 - c_1}{c_3 - c_1} + \frac{a_1 - c_2}{c_3 - c_1} \ln\left(1 + \frac{c_3 - a_2}{a_1 - c_2}\right)$
$c_2 = a_1$ & $c_3 > a_2$	
	$\frac{a_2 - c_1}{c_3 - c_1}$, for $0 \le \alpha \le 1$
	$\frac{a_2 - c_1}{c_3 - c_1}$
$c_2 > a_1$ & $c_1 < a_2$ & $c_3 \ge a_2$	
	$\frac{a_2 - c_1 - \alpha(a_2 - a_1 + c_2 - c_1)}{c_3 - c_1 - \alpha(c_3 - c_1)}$, for $0 \le \alpha \le q_1$ 0 , for $q_1 \le \alpha \le 1$
	$\frac{a_2 - c_1}{c_3 - c_1} - \frac{c_2 - a_1}{c_3 - c_1} \ln\left(\frac{a_2 - c_1}{c_2 - a_1} + 1\right)$
$c_2 > a_1$ & $c_1 \ge a_2$	
	0 , for $0 \le \alpha \le 1$
	0

$c_2 > a_1$ & $c_3 < a_2$	
	1 ,for $0 \le \alpha \le q_2$ $\frac{a_2 - c_1 - \alpha(a_2 - a_1 + c_2 - c_1)}{c_3 - c_1 - \alpha(c_3 - c_1)}$,for $q_2 \le \alpha \le q_1$ 0 ,for $q_1 \le \alpha \le 1$
	$1 + \frac{c_2 - a_1}{c_3 - c_1} \ln\left(1 - \frac{c_3 - c_1}{c_2 - a_1 + a_2 - c_1}\right)$

B.2 Comparison Operators for Crisp Requirements

In the previous section we have derived the comparison operator for fuzzy requirements and fuzzy estimations. Logically, for the evaluation of crisp requirements and fuzzy estimations a similar comparison operator is needed. We can derive this comparison operator from the result in Table B.1 by viewing crisp requirements as a special case of fuzzy requirements.

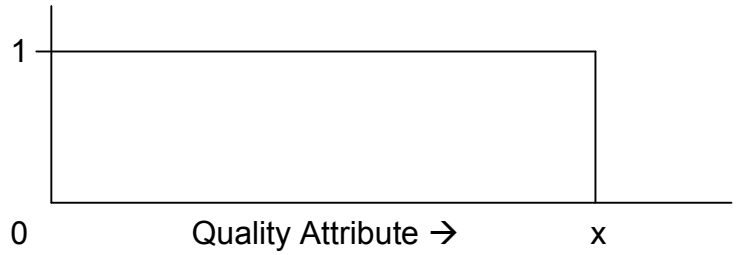
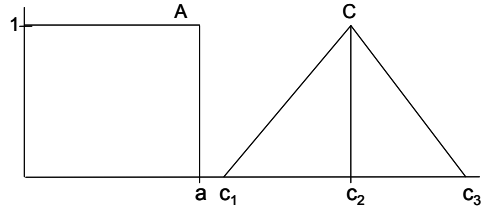
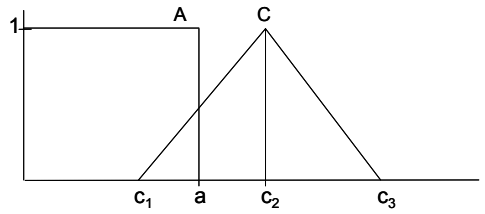
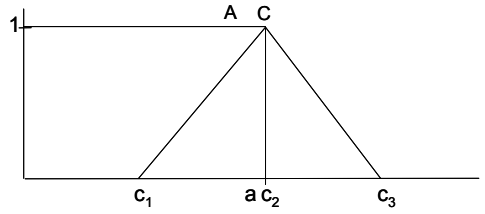
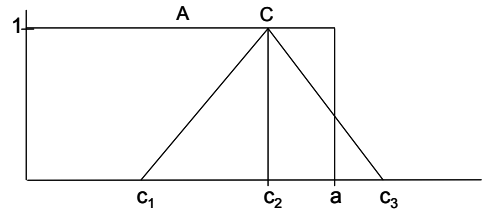
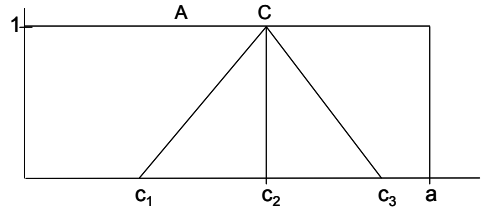


Figure B.2 A crisp requirement

In Figure B.2 the crisp requirement interval is described by a piecewise linear function. The sharp boundary of the requirement can be seen by the abrupt change from 1 to 0 at the value x. Analogous to fuzzy set we call this function the *membership function* of the requirement interval. It can be seen that in this representation crisp requirements are modeled by a special case of fuzzy intervals. Let *A* be a crisp requirement with the crisp interval $]-\infty, a]$, and let the estimation *C* be given by the triangular fuzzy number (c_1, c_2, c_3). The degree of acceptance for C by A is now given in Table B.2.

Table B.2 Comparing crisp requirements with fuzzy estimations

$a \le c_1$	
	0
$c_1 < a \le c_2$	
	$\frac{a-c_1}{c_3-c_1} - \frac{c_2-a}{c_3-c_1}\ln\left(\frac{a-c_1}{c_2-a}+1\right)$
$c_2 = a \ \& \ a \le c_3$	
	$\frac{a-c_1}{c_3-c_1}$
$c_2 < a \le c_3$	
	$\frac{a-c_1}{c_3-c_1} - \frac{a-c_2}{c_3-c_1}\ln\left(1+\frac{c_3-a}{a-c_2}\right)$
$a > c_3$	
	1

The formulas in this table are special cases of the formulas in Table B.1. By substituting *a* for a_1 and a_2, the expressions for the degree of acceptance between crisp requirements and fuzzy estimations is attained.

SAMENVATTING

Het ontwerpen van softwaresystemen van hoge kwaliteit is één van de meest belangrijke onderzoeksproblemen op het gebied van de software engineering. Het onderzoek op dit gebied heeft over de jaren geresulteerd in een veelvoud van ontwerpmethodieken, elk met zijn eigen specifieke plus- en minpunten. Waar bijvoorbeeld het Rational Unified Process een alomvattend ontwerpproces is dat de verschillende fasen van softwareontwerp uitgebreid ondersteunt, richten zogenaamde "agile" processen, zoals "Extreme Programming", zich meer op een flexibele aanpak. Hoewel de moderne ontwerpprocessen zich genoegzaam hebben bewezen en er een veelvoud aan ontwerpprocessen is, zijn deze allen gevoelig voor de gevolgen van imperfecte informatie.

Imperfecte informatie tijdens het ontwerpen van software is informatie, die tot een bepaalde graad onzeker danwel incompleet is. Deze imperfectie kan het gevolg zijn van verschillende oorzaken, zoals incomplete informatiebronnen of een onduidelijk beeld over wat het systeem moet gaan doen. Vanwege het feit dergelijke informatie typisch voor meerdere interpretaties vatbaar is, wordt de uitvoering van het softwareontwerpproces ernstig belemmerd. Met de keuze voor een enkele interpretatie is er het risico dat deze interpretatie uiteindelijk niet de juiste blijkt te zijn. Als gevolg hiervan kan het noodzakelijk zijn (een deel van) het systeem opnieuw te ontwerpen, hetgeen kan leiden tot hoge kosten.

Echter, moderne ontwerpmethoden negeren veelal de aanwezigheid van imperfecte informatie. Over het algemeen vereisen de ontwerpmethoden eenduidige en, bij voorkeur, complete specificaties van eisen, niettegenstaande het feit dat er consensus is over het feit dat dit in het algemeen zeer moeilijk is te realiseren. Ook wordt imperfecte informatie als gevolg van ontwerp activiteiten over het algemeen genegeerd. Echter, imperfecte informatie is een inherent probleem van vrijwel elk ontwerpproces. Toch wordt imperfectie genegeerd door het maken van expliciete veronderstellingen welke op dit moment nog niet kunnen worden verantwoord en kunnen leiden tot foutieve ontwerpbeslissingen. De gevolgen van het negeren van imperfecte informatie zijn zwaarder gedurende de vroege fases van het ontwerpproces. Gedurende het voortschrijdende ontwerpproces kunnen zowel de softwareontwerpers als de klanten tot nieuwe inzichten komen over het te ontwerpen systeem. Omdat deze inzichten slechts druppelsgewijs beschikbaar worden, is het niet mogelijk gedurende de vroege ontwerpfasen de imperfecte informatie efficiënt te corrigeren.

In dit proefschrift identificeren we de twee gebieden waarin imperfecte informatie zich kan manifesteren, namelijk in informatie komend uit de context van het ontwerpproces (met name specificaties van eisen) en tijdens ontwerpactiviteiten. We analyseren de verschillende typen imperfecte informatie and de manier waarop de imperfectie geïnterpreteerd dient te worden. Op basis van deze analyse stellen wij extensies voor softwareontwerpprocessen voor, die op een generieke manier imperfecte informatie kunnen modelleren. Hierdoor wordt het mogelijk verschillende interpretaties voor ontwerpbeslissingen te beschrijven en overwegen met betrekking tot hun toepasselijkheid zonder hier te vroeg daadwerkelijk over te hoeven beslissen. Door het modelleren van alternatieve interpretaties voor een ontwerpbeslissing wordt het algehele ontwerp flexibeler. Nieuwe inzichten gedurende het ontwerpproces kunnen efficiënter worden overwogen, zonder een directe noodzaak to herontwerpen.

In de hier voorgestelde methode zijn technieken gecombineerd uit verschillende disciplines zoals het ontwerp van softwarearchitecturen, kansrekening and de theorie van vage verzamelingen om de relevante eigenschappen van het softwareontwerpproces en imperfecte informatie te kunnen beschrijven. De voorgestelde methodieken verhogen de werklast voor de ontwerper waardoor we een verzameling van gereedschappen hebben geïmplementeerd die de rekenkundige aspecten van de benadering afhandelen. De effectiviteit en het gebruikersgemak

van de benadering en de gereedschappen worden geanalyseerd met behulp van een experiment.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Realtime and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The _ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent- Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima – A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java - Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault- Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005- 10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005- 21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *B¨ohm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther – Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural

Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trcka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

Stellingen

(Propositions)
belonging to the dissertation

IMPERFECT INFORMATION IN SOFTWARE DESIGN PROCESSES

by

Joost Noppen

I.

While it is acknowledged by most software development processes that providing precise requirement specifications is vital for the delivery of high quality software, only few offer explicit support for the definition of such requirements.

II.

Imperfection is an inherent property of the information used during software development, even when it is not always recognized as such.

III.

The requirements for successful iterative design of software systems can not be fulfilled in a realistic setting, which means design processes can not rely only on iteration to ensure the timely delivery of systems with acceptable quality.

IV.

Development methods need to understand the nature of imperfection which occurs during software design, such as conflict, ambiguity or tolerance, since this directly influences the way in which the information should be used.

V.

For the successful application of imperfection models during software design, at any time it should be possible to refine imperfection models that are used. Additionally, it should be possible to remove imperfection models in a consistent and usable manner.

VI.

Similar to the proof by intimidation in formal methods, software engineering has the concept of proof by irritation. By using a constant repetition and variation of drawings and acronyms, the hearer is inclined to believe the speaker, if only to make him stop.

VII.

The chance of one in a million will occur nine times out of ten.

VIII.

On the northern hemisphere, a thesis ideally is written in the period from Autumn to Spring, since then the changes in daylight perfectly match the mental state of the writer.

IX.

The most efficient way to chart differences in eating habits between people, is to organize a barbecue for a large group of people.

X.

Het verrichten van wetenschappelijk onderzoek kost veel hoofdbrekens. Dit is meteen de validatie van het doen van wetenschappelijk onderzoek, want een bekend Twents gezegde geeft aan: *ai d'r met ‘n kop tegn an könt knap’n, dan is 't wat*.