Imperfect Requirements in Software Development

Joost Noppen
TRESE Software Engineering
Dept. of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
noppen@cs.utwente.nl
Tel: +31-(0)53 – 489 5676
Fax: +31-(0)53 - 489 3247

Pim van den Broek
TRESE Software Engineering
Dept. of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
pimvdb@cs.utwente.nl
Tel: +31-(0)53 – 489 3762
Fax: +31-(0)53 - 489 3247

Mehmet Akşit
TRESE Software Engineering
Dept. of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
aksit@cs.utwente.nl
Tel: +31-(0)53 – 489 2638
Fax: +31-(0)53 - 489 3247

Abstract. Requirement Specifications are very difficult to define concisely and unambiguously. Due to lack of information and differences in interpretation, software engineers are frequently faced with the necessity to redesign and iterate. The imperfection in software requirement specifications that causes these problems has lead to for example iterative approaches and incremental design. In this paper, we advocate an approach where the imperfect requirements in requirement specifications are modeled by fuzzy sets. By supporting this approach with a requirement tracing and optimization approach, the necessity for design iteration can be reduced.

Keywords: imperfect requirements, design optimization, decision support, fuzzy requirements

1. Introduction

During the last decades, a considerable amount of software design methods have been introduced, such as Structural design [17] and the Rational Unified Process [6]. Although there are differences among the methods, the general structure of 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 [11], one major problem with software design methods is the existence 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 a later stage the requirements change or are refined, additional iteration 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 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 imperfection in requirement specifications can not be avoided.

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.

The remainder of this paper consists of the following parts: in the next section an example case will be presented and the problems will be identified. Section 3 describes the approach for tracing intermediate design artifacts and the approach for dealing with imperfection in software requirements. In Section 4 we analyze the example case using the results of section three. Related work is described in Section 5. In Section 6 we conclude the paper.

2. Problem Statement

2.1 An Example: Traffic Management System

We consider the possible problems that can occur, when the development of a software system is started with an imperfect set of requirements. To demonstrate the problems, we first present an example. In Section 3 we define the improved approach.

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 the 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 must support the action coordination for traffic flow normalization. The normalization is done by allocating tasks and scenarios to system operators. The Task Allocation part must gather and store information about traffic in its direct and indirect geographical vicinity. To communicate the tasks and actions, the TMS must be able to access its connected roadside systems. In addition, the TMS must 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. After a first evaluation, the software designers have devised the overall architecture depicted in Figure 1.

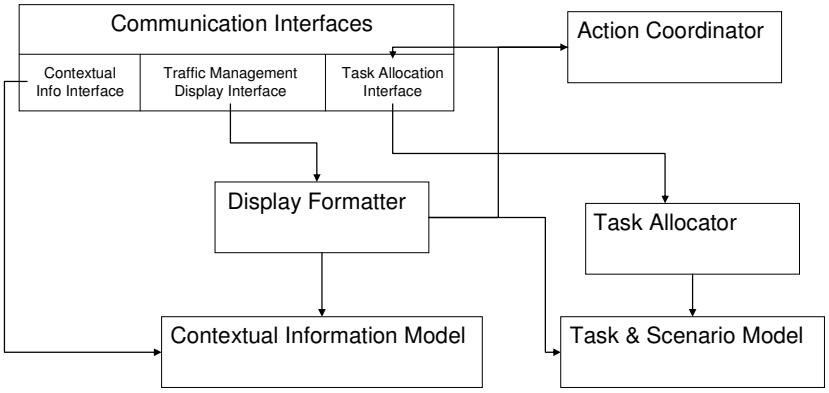

Fig. 1. Overall Architecture of the Traffic Management System

In this figure, six elements are identified for the TMS. The tasks and scenarios are modeled with the *Task & Scenario Model*. Contextual information 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 available information to displayable forms for the *Traffic Display System*. Finally, the *Action Coordinator* determines the actions to be taken to normalize traffic flow. The *Communication Interfaces* enable the interaction with the system from both inside and outside. In the case that traffic flow is hindered, the TMS will initiate a traffic flow normalization. To achieve this, a number of tasks and scenarios are defined 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 step, 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.

2.2 Imperfect Software Requirements as Input of the Software Design Process

The requirements of the TMS, at first glance, quite accurately describe what is expected from the system. However, upon closer inspection, the requirements contain ambiguity in several different definitions. For example, in the second requirement prescribes that there should be an explicit and convenient model of tasks and scenarios. However, the term convenient can imply completely different solutions from the operator point-of-view and the software designer point-of-view. Also, in for instance the fourth requirement, it is described that the system should be able to communicate with the roadside system. However, it is not described what kind of communication is needed, and whether it is single- or bidirectional. The imperfection in the requirements is actually caused by the imperfection that already existed in this particular specification. The cause of the imperfection in these requirements, and functional requirements, in general, is two-fold.

The cause of the imperfection in requirement specifications is two-fold. Firstly, the initial requirements are defined at 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 partial view 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 requirements. 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, for example, formal languages does not resolve the situation, since defining a formal specification requires a very clear understanding of the statements you are trying to formalize. Formal methods can only be used if the information you are using is perfect, which makes it impossible to resolve imperfect information in this manner.

2.3 Systematic Trade-off of System Functionality and Process Parameters

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 re-use existing system parts to minimize costs and development time. In the case of a crisp and concise requirements specification, it is already a very challenging task, but it becomes even more difficult when the software engineer is faced with imperfect requirements. The added difficulty 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. What is needed is an explicit relationship between the requirement and the components that implement this particular requirement. In case of imperfect requirement specifications, the lack of a formal trace becomes even more apparent. Due to the fact that imperfect requirements can become

perfect at the latter stages of the design process, it becomes imperative to be able to determine which components are no longer needed.

3. Software Design with Imperfect Information

3.1 Introduction

To resolve the problems identified in Section 2, we extend the software design such that it is possible to capture the imperfection in the requirements accurately. The first part of our approach extends the tracing capacities of modern design processes, such that it becomes possible to assess individual system designs. To properly evaluate alternative system designs, our approach captures these traits and assesses the system designs accordingly. The second part of our approach addresses imperfection in software requirements in such a manner, that multiple alternative interpretations can be included in the design process. In addition, the stakeholder is able to indicate the relative preference for each alternative, according to several different attributes such as relevance and urgency. The imperfection model is linked to the tracing model, which makes it possible to determine the best system design from the alternatives based on costs and stakeholder preference. However, please note that we do not aim to achieve automated design. Rather, we aim to provide a set of analysis approaches that can support the designer during the design process.

3.2 A Trace Model for Artifacts and Relations

In this paper, we present the Artifact Trace Model (ATM) that captures the relationships between 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 [14], known as Synbad. In an analysis-synthesis based approach, user requirements lead to the definition of a relevant set of interrelated problems that should be solved. Based on this problem decomposition, the relevant domains of expertise are identified, which are commonly named solution domains. From these domains the solution concepts are extracted that make up the system design

In each step in Synbad, an intermediate design artifact, such as a requirement, is transformed into new intermediate design artifacts like the problems that should be solved to implement this requirement. In the Artifact Trace Model (ATM) we propose, we represent intermediate design artifacts by circles and the activity of transforming by arrows. From a set of initial requirements a sequence of transformations needs to be made, until an implementable solution is found. In order to make a complete trace model that represents design processes, it should contain the essential building blocks that can occur. The following building blocks can be identified:

- Requirement
- Problem
- Solution Domain
- Solution
- Component/Class

By connecting these building blocks, a trace of the design process can made. Generally, in a software design process it takes several of these sequences to completely solve a particular problem. By transforming components/classes into new lower-level requirements, and continuing the design process in the same manner the requirements are fulfilled. The structure of the artifact trace model allows the designer to determine which requirements are implemented with a particular selection of components. From a set of requirements, the components can be traced down in the ATM. When we examine the Artifact Trace Model in Figure 3, without going into too much detail at this point, we can trace for instance requirement *R3* to the components *C3.1* and *C3.2*. Complementary we can also see that the set *C2.1.2, C2.3.1* and *C2.3.2* implement requirement *R2*.

3.3 The Fuzzy Requirement Concept

As opposed to assuming 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 imperfect information in the design process, and taking its potential consequences into account, the software design is less vulnerable for its alternative interpretations. Therefore, instead of intuitively assuming one interpretation that hopefully corresponds to the stakeholder's intentions, we propose to include a range of possible interpretations. To accommodate the interpretations, we define the concept of a fuzzy requirement.

We assume that a crisp or 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 requirement 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 consist of the specification of a fuzzy set *FS* on U. The *degree of membership* for each 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.

For example, suppose a stakeholder asks for *I. a convenient model* in the requirement specification. The requirement set representing this specification then is *{ I }*.Suppose this requirement is considered an imperfect requirement, since it is not clear what convenient exactly means. 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 the fuzzy requirement. The requirement specification thus becomes *{ {1/0.4, 2/0.6, 3/0.8} }*. While the definition of the membership values for requirements interpretations is far from trivial, their definition can be facilitated by offering standardized ratings or variations and values. This part is still subject to future research, however.

3.4 Fuzzy Requirements in the Artifact Trace Model

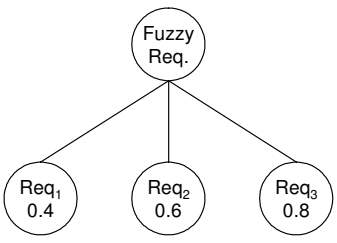

Fig. 2. Alternative Interpretations

When imperfect requirements are replaced with fuzzy requirements, the design process can be continued since the alternative interpretations are treated as normal requirements. However, the resulting software system will likely exceed to stakeholder requirements, since the fuzzy requirements introduce interpretations that, at later stages, can turn out to be irrelevant. When the superfluous interpretations are included in the design process for too long, they can lead to added workload and overcomplete systems. To analyze the correlation that exists between the interpretations and implementation effort, the optimization capabilities of the Artifact Trace Model can be used.

In the Artifact Trace Model, requirements are modeled by rootnodes in a graph. Logically, a fuzzy requirement, like a perfect requirement, is represented by such a root node. To accommodate the identified interpretations of the fuzzy requirement, each interpretation is attached to the fuzzy requirement node as a child node. To each interpretation the degree of membership is attributed. A typical picture for a fuzzy requirement and its alternative interpretations is displayed in Figure 2.

In Figure 2 the three requirements, Req_1, Req_2 and Req_3 as defined in the previous paragraph, are depicted as interpretations of the fuzzy requirement. By treating the interpretations of this fuzzy requirement as perfect requirements, the software engineer can design the software system as he normally would. However, since not every interpretations is necessary to fulfill the fuzzy requirement, a multitude of possible system configurations can be derived from the included interpretations.

For example, in Figure 2 the fuzzy requirement has three interpretations Req_1, Req_2 and Req_3. This fuzzy requirement can now be partially implemented by implementing any subset of these interpretations. As a result, eight 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}. Obviously, implementing all interpretations takes more time and therefore will be more expensive, while implementing a limited set of interpretations will result in a system with lower relevance but also lower costs.

To compare the possible options for a particular fuzzy requirement, we use the membership values that are given to its interpretations. For the *relevancy value of fuzzy requirements* we can choose any function that reflects the combination of interpretations. Here, we define the relevancy to be the algebraic sum of the membership values of all implemented childnodes. The algebraic sum of two numbers A and B is defined as A+B-AB. Since we have the membership value of the interpretation in the requirement set are values between zero and one, the algebraic sum ensures that fuzzy requirements does not have a relevancy larger than one. In addition, the relevancy is always larger than or equal to the largest implemented membership value. For example, if we would implement the components for Req_1 and Req_2 of the fuzzy requirement in Figure 2, the relevancy of this fuzzy requirement would become 0.4+0.6-(0.4*0.6) = 0.76. For perfect requirements we define the relevancy to be one if they are implemented, and zero if they are not.

With the approach described above, we can now calculate the relevancy value of each individual requirement, both perfect and fuzzy. We define the value of the overall relevance of the system to be equal to the product of all requirement values. In the example above, we have focused on the relevancy evaluation that each interpretation has received. Obviously, it is possible to attribute multiple membership degrees to one interpretation, such as one for relevance, one for urgency, etcetera. When we want to use multiple attributes, we need a mechanism to combine these attribute values into an overall value. We define the overall value of the system as a result of these multiple attribute values to be the weighted average of these values.

3.5 Optimization of the System Functionality Trade-off

In the previous section, we have defined how we can calculate system properties, which enables us to compare systems based on these system properties. We can now define optimization goals and systematically search for systems that adhere to these goals. We can distinguish between two configurations of optimization goals:

Maximization of Attribute Values

The first configuration is aimed at the maximization of one or more attribute values. Typically, while looking for an optimum value, a number of constraints must be fulfilled for the other attributes. For costs, typically an upper boundary is defined, and other system attributes mostly restricted by means of a lower bound.

Minimization of Costs

The second configuration is aimed at minimization of costs for the system that is being developed. While in general, the best solution is to do nothing, in practice a number of constraints should be fulfilled, all of the form that a system attribute should not drop below a certain value.

Both configurations search for a particular optimal system among all possible systems that can be derived from the Artifact Trace Model. The amount of systems that needs to be evaluated grows exponentially with the amount of interpretations for fuzzy requirements. The amount of systems with *n* fuzzy requirements equals:

$$\prod_{i=1}^{n} 2^{\#_i} \text{ , where } \#_i \text{ is the amount of interpretations for fuzzy requirement } i$$

To reduce this complexity, we propose the use of a heuristic approach when optimizing the system design. The starting point for the heuristic approach is the system for which all interpretations are implemented. The systems we consider from this point are all the systems that can be achieved by removing one interpretation. For each system, we determine the attribute values and the optimization criterion value, and

we calculate the value *Δ criterion value / Δ attribute values*. We then choose the system for which this value is the smallest, and repeat this process for this new system. The stopping criterion for the minimization of costs is when none of the new systems that can be derived from the current system adhere to all the restrictions on the attribute values. When this is the case, the current system is heuristically optimal. For the maximization of attribute values, the stopping criterion is the system for which the costs restriction fulfilled. Since the attribute values are maximal when all interpretations are included, the only restriction that can be violated is the cost restriction. Therefore the heuristic optimizer should search for a system that adheres to the costs constraint and also does not violate other constraints. In a worst case scenario this heuristic approach will be faced with a quadratic complexity.

4. Analysis of the approach using the example case

To demonstrate our approach, we apply it to an example called the Traffic Management System. To demonstrate and analyze our approach, we first trace the design process while assuming that the requirements are perfect. We evaluate these results, by tracing and optimizing the design process while modeling the imperfection in the requirements.

4.2 Analysis with Crisp Requirements

We will first design the TMS while not considering imperfection that could be present in the requirement specification. First, the requirements are transformed into a set of problems that need to be solved. Second, for each of these problems a solution domain and a solution is identified. Finally, from these solutions an overall architecture is defined. In Table 1, the first step is described, where for each requirement the relevant problems are identified.

Table 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 Tasks 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 this table, a number of problems are identified for each requirement, which need to be resolved in order to fulfill the respective requirement. For example, for requirement 1 the first problem P1 is to decide on the interaction mechanism, and the second problem P6 is how this interaction will be supported by the model. Note that a number of problems are reused for multiple requirements. For example P2.1.2 is a problem that must be solved for both requirement 2 and requirement 3. This reuse means that when P2.1.2 is solved, a part of requirement 2 as well as requirement 3 is resolved.

The next step in the design process, is to identify solutions for the problems that have been found. In order to solve the problems that have been identified, available knowledge sources on the specific areas are used, 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. For example, in the problem set communication of data is an important issue. Typically, a uniform communication

interface is a useful solution, which is used to solve P1.2.1 and P4.1.2. For problem P2.1.2, there is emphasis on the extensibility of the task and scenario model, and for P1.2.2 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 Table 3.

As the final step, the selected solutions should be mapped to a component model, which localizes the functionality that is needed to implement the system. Since the decomposition of the system into solution parts, the structure is largely known. However, since a number of solutions are too large to fit into one component and other functionality can be provided by commercial components, the component form a more refined model of the TMS system. The way in which the components are related to the solutions is described in Appendix Table 4. In addition, in this table the time is estimated that is expected for the implementation or adaptation of these components for the TMS. These estimations are expressed in person-months.

The implementation of the components that are needed for the TMS sums up to 33.1 person-months. We can make a graphical depiction of the design steps that are described in this paragraph. This depiction is achieved by explicitly linking the artifacts, such as for instance requirement 1, which is decomposed into P1.1 and P1.2. In Figure 4 the Artifact Trace Model for the TMS is depicted.

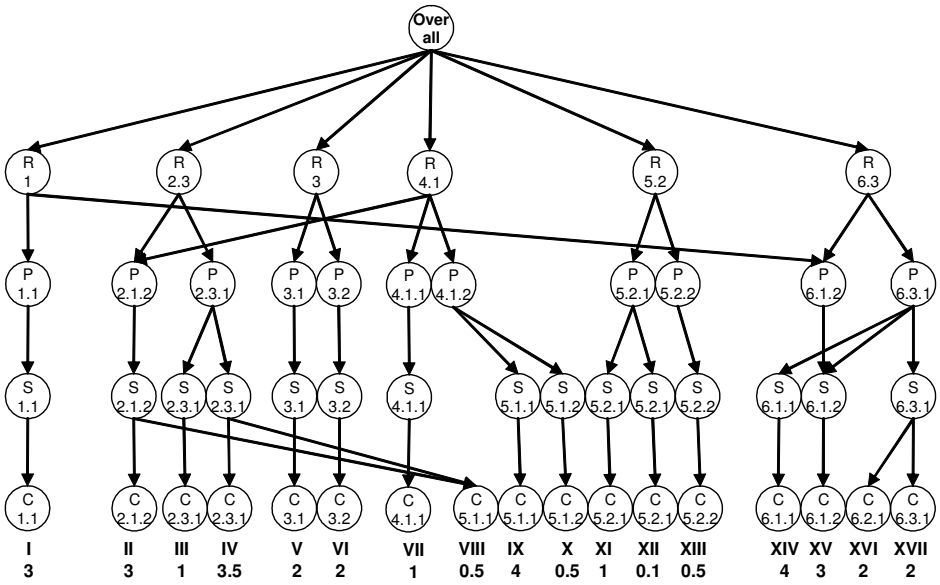

Fig. 3. Artifact Trace Model with Crisp Requirements

In this picture, 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. When we place these components in the reference architecture, this results in the following picture:

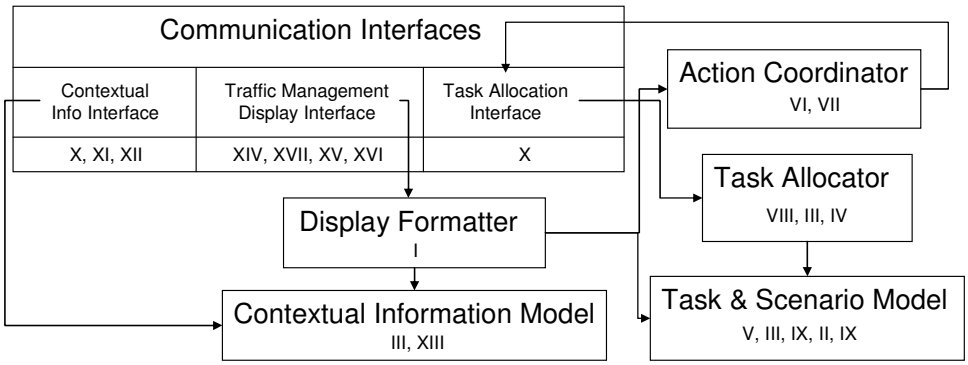

Fig. 4. Architecture of the TMS with Crisp Requirements

In Figure 4, the components are placed in parts of the reference architecture. The resulting architecture is an implementation of the requirements specification at the beginning of the paragraph. However, it is only acceptable if the chosen interpretations of the requirements, either chosen implicitly or not, reflect the stakeholder desires. In the next paragraph we use fuzzy requirements to see whether architecture in Figure 5.5 is the best solution. Note that some components appear multiple times in the picture to indicate that these components are used at multiple places in the system.

4.3 Analysis with Fuzzy Requirements

For our example, let us consider the design of a system where requirements *2, 4, 5* and *6* are identified as imperfect requirements. These four requirements are replaced by fuzzy requirements, and for each of these requirements three possible interpretations are identified. In addition, in accordance with the stakeholders, a number between 0 and 1 is attached to each interpretation, indicating the degree to which this interpretation is applicable, which is its membership value. In the following requirement specification, the interpretations are described as follows:

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

In the same manner as before, the software engineers identify the problems for these requirements.

Table 2. From Requirements to Problems

Requirement	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	P5.3.1 How do we define a generic model that captures contextual information for internal and external usage?, P5.1.2, P5.2.1
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 2, 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 use during the optimization of the Artifact Trace Model. The subsequent steps where the problems are refined to solutions, and the solutions to components can be found in Appendix Table 5 and 6 respectively. When we depict this design process in an Artifact Trace Model, this results in the following picture:

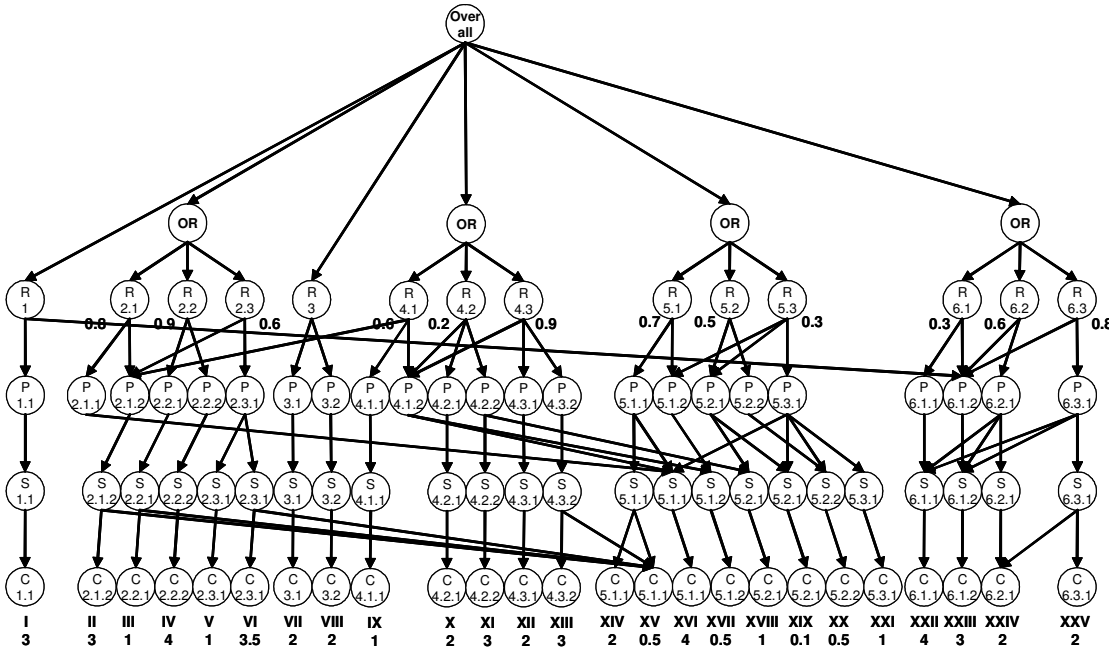

Fig. 5. Artifact Trace Model of the TMS with Fuzzy Requirements

In Figure 5 the Artifact Trace Model is depicted for the design of the TMS with imperfect requirements. The nodes labeled *OR* depict the imperfect requirements, and the fact that at least one of their respective child nodes should be implemented. For all the components the implementation time is estimated in person-months. As indicated in Section 3.4, only one 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 architecture in Figure 4 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.

4.3.2 Optimized Systems for Relevance and Costs

When we take as a reference point the system from Section 4.2, we see that the requirements that are implemented by these components is *{ R1.1, R2.3, R3.1, R4.3, R5, R6 }*. When we determine the overall rele-

vance according to our method, 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, which either offer lower costs or higher relevance. First, we identify the system with minimal costs, while having a relevance of at least 0.114 , which is equal to the relevance of the system resulting from the crisp requirements. The system that is the result of this optimization consists of the following components: *{ I, II, VII, VIII, IX, XIV, XV, XVI, XVII, XVIII, XIX, XX, XXII, XXIII }*. With these components, the following requirements are implemented: *{ R1, R2.1, R3, R4.1, R5.1, R5.2, R6.1 }*. The resulting architecture for this optimization is depicted in Figure 6.

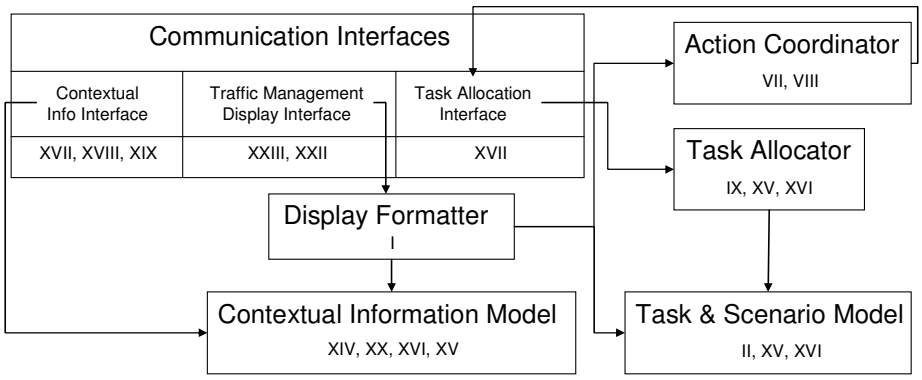

Fig. 6. Cost Minimized TMS Architecture

From this architecture, it can be seen that the cost minimization is achieved by choosing different interpretations for both requirement 2 and 4. Additionally, the relevance is raised by including two 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. We can conclude that the optimal system that can be found using the Artifact Trace Model, not only exhibits lower costs than the crisp system, but also has a better relevance.

Second, we maximize relevance, while not exceeding the amount of 33.1 person-months. Our approach comes up with a system consisting of the following components: *{ I, II, VII, VIII, XII, XIII, XIV, XV, XVI, XVII, XXII, XXIII, XXIV, XXV }*. With these components the following requirements are implemented *{ R1, R2.1, R3, R4.3, R5.1, R6.1, R6.2, R6.3 }*. This architecture differs considerably from the system that was designed based on perfect requirements. Especially for requirement 4 multiple interpretations have been included, which considerably boasts the relevance of this system. This system has a relevancy of 0.476 and the cost of implementing the components is 33.0. For this optimization we can conclude that the resulting system has a considerably higher relevance, and still the costs are lower than for the perfect requirements system. Due to space limitations a graphical depiction of the resulting architecture has been omitted.

5. Related Work

5.1 Decision Models of Software Processes

During the last 20 years, a considerable number of design methods have been introduced, such as Structural design [17] and Rational Unified Process [6]. These approaches generally differ from each other with respect to the adopted models, such as functional, data-oriented, object-oriented, etcetera. These 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 [15], 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 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 [8][5], 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. As a result, most approaches can not deal with imperfect information in the design process. In [11], 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 paper.

5.2 Imperfect Information in Design Processes

Modeling imperfection in the inputs of design processes is not new. However it is seldomly applied in the field of software design. In [1] fuzzy logic is applied to support the partial applicability of design heuristics in the OMT 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 [16], a fuzzy logic framework is defined that can be used to model imprecise functional requirements. After each design step the proposed solution can be compared with the requirement, similar to proving an invariant over a piece of code. The resulting value then indicates to which degree the requirement holds.

In [9], an extension to decision trees (see next paragraph) is proposed. The imprecise attitude of the decision maker with respect to risks is modeled using techniques from fuzzy logic, and combined with the decision optimization algorithms of probabilistic decision trees. In [10], an approach is proposed to model imprecision in design inputs. This imprecision is captured using fuzzy set theory, and the imprecision 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 modeled imprecision using fuzzy set theory. In [12], the uncertainty of market demands for software products is captured using probabilistic models. These models are then used by a Markov decision model to determine the implementation order of the components of the system, in order to optimize the expected profit.

5.3 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 this purpose. For instance, a tracing approach based on hypertext [7] is primarily aimed at easily browsing to 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 [4] and matrix sequences [3]. Another use of trace models is to analyze the fulfillment of requirements based on the structure of the requirement trace. Examples of such approaches are assumption-based truth maintenance networks [13] and constraint networks [2]. While all these approaches have specific uses, it is not possible to apply these approaches to work with imperfect inputs and optimize system designs. This limitation is caused by in the need for specific attributes that are needed in the trace model, which are mostly only in part captured by these tracing models.

6. Conclusions

In Section 2, imperfect information in software requirements and trading off system functionality systematically are identified as two important problems in the design of software systems. The first problem can lead to the development of software systems that do not reflect the stakeholder's intentions, since the imperfect requirements can be interpreted differently by software engineers. The second problem is caused by

the lack of a tracing model that explicitly models the relationships between requirements and the components that implement them. This lack makes it impossible to analyze alternative systems based on the components that are implemented, while simultaneously considering cost or implementation time.

We have shown that imperfect information can be managed by capturing the nature of the imperfection. To support imperfection, we describe the imperfect information with fuzzy sets and treat the extended requirements in the same way as normal requirements. As a result, the design becomes more flexible and resilient to the occurrence of new insights alongside the development process. By adding annotations to the imperfect requirements, we can model particular interests of stakeholders, such as desirability or applicability. In addition, we have shown that the design process can be supported by tracing the transformation steps that are taken from the initial requirements to the final components. For each design step the subsequent intermediate artifacts are captured, and explicitly linked to the originating artifact. The relationship between these elements is captured by a tree structure, which can be used to trade off system functionality and process parameters, such as cost or implementation time, systematically.

Our approach was demonstrated by applying the approach to an example case, where four out of six requirements contained imperfect information. In the traditional evaluation method, one interpretation for each requirement was used. When this system was compared to the results of our approach, it turned out to be considerably more expensive and less adequate according to the stakeholder's desires. The approach that has been presented in this introduces new activities and computations to be performed. Therefore, to support the software engineer in the application of this approach, a prototype tool has been implemented. This prototype is currently being evaluated in experimental settings to validate our approach.

7. References

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

[2] Bowen, J., O'Grady, P. & Smith, L. (1990). 'A Constraint Programming Language for Life-Cycle Engineering', *Artificial Intelligence in Engineering*, Vol. 5, No. 4, pp. 206-220.

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

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

[5] Finkelstein, A, Kramer, J. & Nuseibeh, B. (1994). 'Software process modelling and technology', Research Studies Press Ltd..

[6] Jacobson, I., Booch, G. & Rumbaugh, J. (1999), 'The Unified Software Development Process', Addison Wesley, ISBN 0-201-57169-2

[7] Kaindl, H. (1993). 'The Missing Link in Requirements Engineering', *ACM SIGSOFT Software Engineering Notes*, Vol. 18, No. 2, pp. 30-39.

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

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

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

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

[12] 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', *IEE Proceedings Software* **151**(2), pp. 85-94.

[13] 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.

[14] 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.

[15] Tekinerdogan, B. & Aksit, M. (2002). 'Classifying and evaluating architecture design methods' In: Mehmet Aksit (Eds.) *Software Architecture and Component Technology*, Kluwer Academic Publishers, pp. 3-28.

[16] 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

[17] Yourdon, E. & Constantine, L.L. (1979). 'Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design', Prentice-Hall.

8. Appendix

Table 3. 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

Table 4. 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

Table 5. 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 System
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 Databse 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 6. 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 State Machines, 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