Dealing with Fuzzy Information in Software Design Methods

Joost Noppen (SA)
TRESE Group
Dept of Computer Science
University of Twente
P.O. Box 217, 7500 AE, Enschede
The Netherlands
noppen@cs.utwente.nl

Pim van den Broek
TRESE Group
Dept of Computer Science
University of Twente
P.O. Box 217, 7500 AE, Enschede
The Netherlands
pimvdb@cs.utwente.nl

Mehmet Aksit
TRESE Group
Dept of Computer Science
University of Twente
P.O. Box 217, 7500 AE, Enschede
The Netherlands
aksit@cs.utwente.nl

***Abstract* - Software design methods incorporate a large set of heuristic rules that should result in stable software architecture of high quality. In general, clearly defined inputs are required to deliver the desired results. Unfortunately, especially in the early phases of software development, it is very difficult or even impossible to provide precisely defined information. Since methods cannot deal with imprecision, the designers need to make approximations which are generally not justifiable. In this paper, we will advocate an approach where the inputs for software design methods are modeled by using fuzzy sets. This approach renders the need for introduction of extra information for removal of inexactness obsolete.**

I. Introduction

During the last decades a considerable amount of software design methods have been introduced [1]. Although there are differences among methods, the general structure of methods is quite similar. They all require a well-defined requirement specification, which are transformed into a system design in a number of design steps. Each of these steps can be seen as transformations from one intermediate design model to another.

As has been identified in [2], one major problem with software design methods is that the initial requirements can only be transformed into a design model when they are crisp, unambiguous and stable. However, requirements generally contain conflicts, are vague and change over time. As a result additional steps are taken to make the requirements conform to this initial demand. Vagueness and conflicts are removed by making explicit assumptions on the requirements, even if these assumptions cannot be justified at the current point in time. This results in loss of information. The problem that is being solved by the software design process no longer corresponds to the initial requirements provided by the customer. Therefore there is a high risk that a mismatch will occur between the system that will be delivered, and the system that was demanded by the customer.

To reduce these problems we propose to explicitly model the approximate and inexact nature of requirements as they are provided by the customer. Instead of forcing requirements to become crisp, the vague and conflicting information can be modeled using fuzzy sets. These fuzzy annotations in the requirements cause the software design process to result in a fuzzy design, rather than a crisp and directly implementable design. For the fuzzy design to become implementable, a defuzzification step is needed. By postponing the removal of vagueness and conflict to the point of a fuzzy design, unjustifiable assumptions can be avoided and the timeframe for the required information to become available is extended. This may help in improving the quality of design because in the later phases of the design process, it is likely that more precise information is available.

The remainder of this paper will consist of the following parts: section II describes the forces that influence the effectiveness of software design processes. Section III describes a formal model for representing design methods and in section IV the approach for modeling fuzzy requirements is described. In section V a case study will be described demonstrating the approach. Section VI describes the conclusions and finally the future work is described in section VII.

II. Disruptive forces in Software Design

The inexactness in software design processes can be caused by many different sources and manifest itself in many different ways. In general, three “disruptive forces” can be identified that influence the effectiveness of software design processes. In the worst case, this may lead to repeating the process from the beginning. These three forces are:

1. **Vagueness & Uncertainty:** The occurrence of vagueness and uncertainty may hinder the progress of the process, especially if the follow-up phases require precise input.
2. **Conflict:** The occurrence of conflict during the design of software can delay the process. Whenever a conflict arises, the software engineers have to resolve the conflicts. In some cases the conflict may be obvious, but in most cases conflicts may be identifiable only after several iterations and refinement steps.
3. **Change:** The occurrence of a change during software design processes may invalidate the trajectory of the software design process.

These three disruptive forces clearly hinder the software design process. Vagueness will lead to difficulties in either

properly defining the relevant models for one or more of the stages or properly defining the relations between the model elements (artefacts). Conflicts can lead to difficulties in defining relationships because the conflicting elements per definition cannot be integrated. A change can even have a multitude of influences, since a change can mean addition or removal of requirements. This may result in addition of new artefacts or deleting existing ones. In this paper we will focus on missing or incomplete information in the requirements.

III. Representation of Design Methods

To analyse how these forces influence software design processes, a formal model of software design methods is needed. In this paper we present a model that represents a general design methodology, commonly referred to as analysis and synthesis, as for example exemplified in the Synthesis-based Software Architecture Design method [3], known as Synbad*. In an analysis and 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 (commonly named solution domains). From these domains the solution concepts are extracted that make up the system design. Schematically an analysis and synthesis process (Synbad) can be characterized as follows:

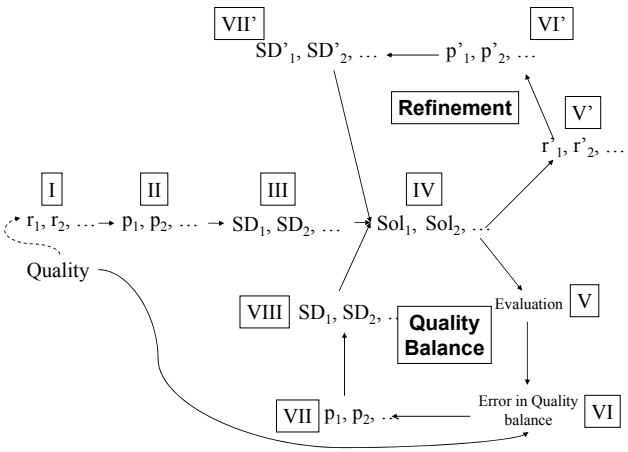

Fig. 1 A schematic overview of Synbad

In figure 1 Synbad is depicted as a sequence of phases numbered I, II, etc.. These phases represent intermediate design models during the software design process. The first phase leads to the *requirements definition*. The requirements are denoted by $r_1, r_2, \ldots$ in the figure. In the second phase decomposition is made into the relevant *problems* in order to implement the requirements that were found in phase I. The problems are denoted $p_1, p_2, \ldots$ in the figure. The problems are mapped to *solution domains* in the third phase, denoted by $SD_1, SD_2, \ldots$. In the fourth phase the *solution concepts,* denoted $SC_1, SC_2, \ldots$ are selected that will make up the final system.

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. This is done by defining *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 found.

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, etc.. These characteristics are compared to the 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 identified.

Each step in Synbad from one phase to the next requires input from a software engineer. In the model we propose, a distinction is made between two types of building blocks, *artefact blocks* and *process blocks*. Artefact blocks represent parts of intermediate design models that are the result of steps taken in the design process, such as the requirements or problems. Process blocks represent activities of transforming one design model to another. We represent intermediate design models by circles and the activity of transforming by rectangles. A typical step will look like this:

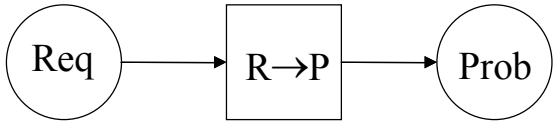

Fig 2 Transforming a requirement to a problem

In figure 2 a requirement is transformed into a relevant problem, via a process that associates problems with requirements.

Since Synbad consists of a sequence of such steps, the process of designing is essentially a classification activity. From a set of initial requirements a sequence of transformations needs to be made, until an implementable solution is found. A formal model that represents design processes should contain the essential building blocks that can occur. The following building blocks can be identified:

* Note that Synbad represents a method that has the same basis as many different design methods based on domain specific input

TABLE I
BUILDING BLOCKS FOR SOFTWARE DESIGN PROCESSES

Building Block		Description
Artefact Blocks	Requirement	Requirements are the initial input for the design process, and each instance of this building block represents a single requirement on the eventual result.
	Composition	The Composition building block is the explicit representation of the way requirements should be composed to implement the overall required system.
	Quality	Quality building blocks represent the quality constraints that are imposed on the overall problem. These can represent architectural aspects such as adaptability, but also more abstract issues such as usability.
	Problem	Problem building blocks represent actual technical problems as they are encountered by designers. These problems can reside on several different levels of abstraction, with the most abstract levels closer to the requirements and the most concrete problems closer to the implementation.
	Solution	Solution building blocks represent actual solutions to problems on a particular level of abstraction. This means that like problems also solutions can reside on multiple levels of abstraction, with the most concrete solutions being the actual implementation.
Process Blocks	Sequencer	The sequencer building block is a block that represents a prioritisation mechanism for problems to be solved on a single level of abstraction. Due to complexity it might be impossible to solve all problems on a certain abstraction level in parallel. Therefore prioritisation is needed, which is provided by the sequencer.
	Relation Process	The Relation Process building block represents the activity of stepping from one artefact to another, such as going from a problem to a solution. This building block models the actual engineering input into the design process.

By connecting these building blocks and defining a sequence in time, a formal representation of design processes can made. The connections between the blocks indicate the order in which the activities are executed. For connecting the blocks the following rules apply:

Acyclic Graph
The graph representing the software design process is a directed, acyclic graph. Cycles are not allowed since the model represents a process that consists of sequential steps. Therefore it is not possible to return from a node to a node that already has been visited.

Artefact Block Connection
Each artefact block can only be connected directly to a process block. This is because a mapping should be done from one artefact block to the next, which can't be done automatically. As a completion of this rule, process blocks always connect two or more artefact blocks.

Quality Block Connection
The only exception to the Artefact Block Connection Rule is for Quality Blocks. The blocks can be directly connected to requirements and/or composition operators for requirements. Other than that also Quality Blocks should be directly connected to process blocks.

Composition Process Block
A process block that marks the activity of composing two or more individual solutions to a bigger solution part should always contain at least one input from a composition operator or Composition Artefact Block in addition to at least two other artefact blocks. Also Composition Process Blocks can only occur at the lowest level of abstraction (i.e. when an actual implementable solution has been found).

IV. TRANSFORMING FUZZY REQUIREMENTS TO ACCOMMODATE DISRUPTIVE FORCES

As was identified earlier, generally software design methods do not explicitly identify disruptive forces that influence the design process. However, the disruptive forces specifically cause the inputs of the software design methods to be vague, conflicting and subject to change. Therefore Software Design Processes need to be extended with mechanisms that can accommodate these forces.

Vague or missing information is a prominent force to be dealt with in software design methods. This is mostly resolved by adding additional information, even if this is not known at the current point in time. A more fitting solution would be to capture the actual input including vagueness, and design the software system based on this input. This information in turn should be processed throughout the entire design process, to come to a fuzzy design. A fuzzy design then represents the exact solution to the requirements that were given, but contains fuzzy information. The output should always be implemented in the final system.

We assume that a crisp requirement consists of the specification of a set S on a universe U. For instance, specification of the set $\{A, B, C\}$ corresponds to the requirement: I need property A, B and C and no other from the universe U. Furthermore we assume that a fuzzy requirement consists of the specification of a fuzzy set FS on U. For instance, the fuzzy set $\{A/0.5, B/0.7, C/0.3\}$ corresponds to the vague requirement that property A is requirement to degree 0.5, property B to degree 0.7 and property C to degree 0.3.

Instead of assuming only one alternative in case of a vague requirement, we propose to solve this problem by including a range of alternatives that can clarify the requirement. Each of the alternatives will be member of a fuzzy set on the universe of requirements. The value of membership should be determined in accordance with the customer, and can be interpreted as the relevance of the requirement to the customer.

For the design process all alternatives should be considered, and therefore they fuzzy set is transformed to a set of individual requirements with numbers denoting their respective relevance. Now all requirements can be used as input for the software design process. When we assume the exact execution of the design process for n requirements is a black box, we get the following picture:

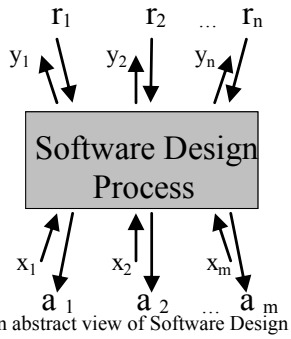

Fig 3 An abstract view of Software Design Processes

In figure 3 $r_1, r_2, \ldots, r_n$ represent the relevance for each of the n requirements. $A_1, A_2, \ldots, A_m$ represent the resulting artifacts from the software design process. At this point we have the choice to continue with any subset of the m artifacts and build the final system. For each of the selected artifacts a 1 is passed into the design process model, and a zero is passed into the design process model for the others. The formal model enables us the force these values back up through the black box (using the values $x_1, x_2, \ldots, x_m$). This results in a value of zero or one for each of the requirements indicating whether or not it is fulfilled by choosing this particular subset of artifacts. These values are indicated by $y_1, y_2, \ldots, y_n$ in the picture.

Passing the values down is done by passing the relevance values from one artifact block to each other artifact block to which it is connected through a process block. For instance, the relevance value of each requirement is passed to each problem that should be solved as a result of this requirement. Whenever an artifact block is the result of two or more process blocks, the highest value is taken. Passing values back up is done similarly. However, now it is possible to encounter multiple artifacts that lead to one artifact block (for instance multiple solutions that one particular problem). In this case the lowest of all values is taken.

By modeling additional properties of the artifacts the selection of a subset of artifacts can be defined as an optimization problem. For example, when the additional property of the artifacts is cost of implementation (indicated by c_i) the following optimization strategies can be defined:

Maximize User Satisfaction

Assuming user satisfaction will be higher whenever more of the given requirements are satisfied, the most simple optimization problem is to find the subset of artifacts that maximizes the following expression: $\sum_{i=1}^{n} y_i r_i$. In the ideal case all the requirements should be satisfied since the costs are not constrained. If the optimum does not return the sum of all r_i, this means some requirements could not be fulfilled, which indicates a conflict in the requirements. This optimization strategy could therefore be used as a conflict detection method.

Minimize Costs

Another approach is to minimize cost of implementation. Especially for software production companies this can provide interesting information. The minimal cost of implementation can be found by minimization of the following expression: $\sum_{i=1}^{m} c_i x_i$. The constraints provided by the customer will ensure that a minimal set of requirements will be implemented.

Trade-off between User Satisfaction and Costs

A third strategy is to enforce a trade-off in the optimization between the user satisfaction and the costs. For this strategy three different approaches can be identified:

The first approach is to express a minimum user satisfaction and add this to the set of constraints for the optimization problem and then minimize the costs using the expression that was defined earlier.

The second approach is to express a maximum allowed cost as a set of constraints for the optimization problem and to maximize the expression for user satisfaction that was given earlier.

The third approach is to define a cost-function for the optimization problem that makes a trade-off between costs of implementation and user satisfaction directly.

V. Case Study: PDA Input & Storage System

To illustrate how this works, we will present and analyse a very small example. This example is part of the implementation of the software for a Personal Digital Assistant (PDA) operating system. In this particular example we will focus on the means of inputting information into the PDA, and storing this information. The customer asks for a system that is able to take inputs and store them in text file format. At this point it is not exactly known in which ways it should be possible to give these inputs. However, the customer is capable of indicating several possible alternatives and their expected relevance. For simplicity this example will focus only on two alternatives: Textual Input and Audio Input. In accordance with the customer the relevance rating of Textual Input is set to 0.6 and the relevance rating of Audio Input is set to 0.4. However, the customer wants to have at least one type of input implemented.

The fuzzy set representing the requirements can now be characterized as follows: *{Textual Input/0.6, Audio Input/0.4, Write Text to Disk/1.0}*. The first step is transforming the fuzzy set to crisp requirements with a property containing the membership value:

FR_1	*The system should be able to accept textual input from the user (0.6)*
FR_2	*The system should be able to accept spoken input (Audio) (0.4)*

FR_3 *The system should be able to store the given input in text format on a local disk (1.0)*

To these requirements we can attach the values y_1, y_2 and y_3, in the same way as figure 3. We attach y_1 to FR_1, y_2 to FR_2 and y_3 to FR_3. To fulfill the requirements, these values are bound to two constraints. First of all, at least one of the input types should be implemented. This is guaranteed by the constraint: $Max(y_1, y_2) = 1$. The second constraint is that the output requirement FR_3 should always be implemented. This is guaranteed by the constraint: $y_3 = 1$. Next the software design process is modelled using the model presented in section III. This results in the following model:

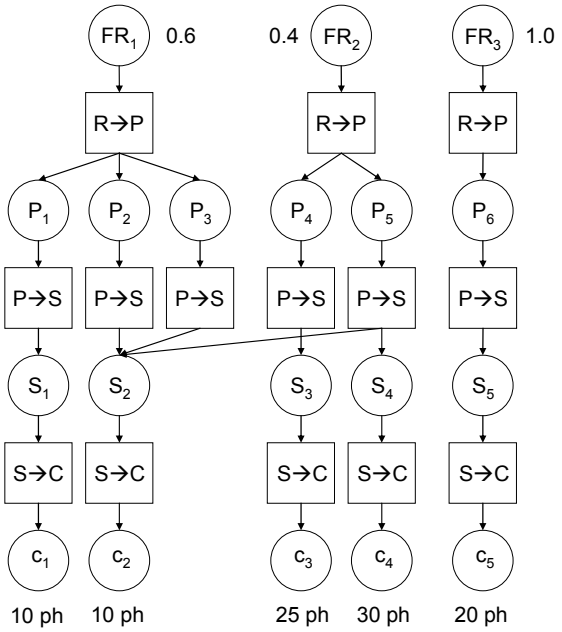

Fig 4 Formal Model of the Software Design Process

TABLE II
LEGEND FOR THE DESIGN PROCESS MODEL

Artefact Block	Description
P_1	How do we get the textual input from the user?
P_2	How do we treat special characters?
P_3	How do we convert to a predefined format?
P_4	How do we record audio input from the user?
P_5	How do we convert to a predefined format?
P_6	How do we store textual information?
S_1	Keyboard reader
S_2	Input Formatter
S_3	Audio Recorder
S_4	Voice Recognizer
S_5	File Writer
C_1	KeyBoard Reader Class
C_2	Input Formatter Class
C_3	Audio Recorder Class
C_4	Voice Recognizer Class
C_5	File Writer Class

As can be seen in figure 4, this particular design method has resulted in a design that consists of a set of classes. The classes in this diagram (c_1, c_2, c_3, c_4, c_5) are denoted with a number indicating the implementation time in person hours.

Note that the model that has been presented is not the only possible model. Other problem decompositions and solution techniques could have been chosen that would have been equally applicable. In addition, note that the presented design model does not completely cover all aspects of design, such as communication between classes and quality aspects. These have been left out, in order not to make the example overly complex.

Based on the formulas we can now formulate the optimization problem for finding the best defuzzified design. The first step is to determine for each requirement the minimal set of classes which should be implemented to fulfill the requirement. For the example this results in the following:

FR_1 *$\{c_1, c_2\}$*
FR_2 *$\{c_2, c_3, c_4\}$*
FR_3 *$\{c_5\}$*

Using the formulas for calculating user satisfaction and the cost estimations in person hours, the following table of implementation possibilities can be compiled:

TABLE III
IMPLEMENTATION OPTIONS PROPERTIES

	Fulfilled Requirement	$\{y_1, y_2, y_3\}$	Person hours	User Satisfaction
I	∅	{0, 0, 0}	0	0
II	$\{FR_1\}$	{1, 0, 0}	20	0.6
III	$\{FR_2\}$	{0, 1, 0}	65	0.4
IV	$\{FR_3\}$	{0, 0, 1}	20	1.0
V	$\{FR_1, FR_2\}$	{1, 1, 0}	75	1.0
VI	$\{FR_1, FR_3\}$	{1, 0, 1}	40	1.6
VII	$\{FR_2, FR_3\}$	{0, 1, 1}	85	1.4
VIII	$\{FR_1, FR_2, FR_3\}$	{1, 1, 1}	95	2.0

The constraints specified that $Max(y_1, y_2) = 1$. This invalidates option IV of the table. The constraints also specified that $y_3 = 1$. This invalidates options I, II, III and V. Therefore for our optimization problem only options VI, VII and VIII are proper solutions.

Assume there is a total of 75 person hours to divide on the implementation of the classes. How should we distribute the available resources? According to the table it is not possible to implement options VII and VIII. Therefore the optimal decision for assigning 75 person hours is to implement requirement *FR_1* and *FR_3*. Note that only 40 person hours are assigned, since assigning the remaining 35 person hours cannot lead to a better result.

From the table it is also obvious that maximal user satisfaction is achieved when all requirements are implemented. From this it can be concluded that the requirements have not lead to conflicts during the software design process.

Another approach that was identified is to demand a minimal user satisfaction, and then minimize cost. Assume a minimum user satisfaction of 1.4 is demanded. From the available solutions, option VI is clearly minimal with respect to cost.

Finally, the strategy of defining a trade-off function between costs and user satisfaction will be applied. This strategy is more complex than others since the definition of trade-off function is not a straight-forward activity. For this paper we will therefore define a simple trade-off function. Assume that user satisfaction will lead an extra increase in revenue, due to personal marketing. This increase is estimated at € 2000,- per User Satisfaction point. Assume also that each person hour costs € 30,-. The trade-off function can now be defined as follows: The result of the trade-off function is the increase in revenue (User Satisfaction x 2000) minus the cost (Person Hours x 30). Now the following table can be compiled:

TABLE IV
TRADE-OFF FUNCTION RESULTS

	Fulfilled Requirement	Trade-off function
VI	$FR_1 \wedge FR_3$	2000
VII	$FR_2 \wedge FR_3$	250
VIII	$FR_1 \wedge FR_2 \wedge FR_3$	1150

For this specific trade-off function, option VI is the best choice. This means, according to this trade-off, the crisp design to be implemented with the current vague inputs should be $\{c_1, c_2,, c_5\}$.

VI. CONCLUSION

This paper identifies the need of software design processes for clearly defined inputs to result in software architectures of high quality. However, it can be very difficult or even impossible to provide precisely defined information. In this paper three disruptive forces, vagueness, conflict and change, have been identified that influence the quality of the result of software design processes. The inability of software design processes to accurately accommodate these forces force software engineers to add information to initial requirements, even if this information is not known at the current moment in time. Therefore there is high risk that a mismatch will occur between the desired system and the system that is eventually delivered to the customer.

We have proposed an approach for modeling and tracing vagueness in software design methods by using fuzzy sets. This enables the designer to represent vagueness in requirements, and consider explicitly how this should influence the system that is implemented. The approach consists of two individual parts, a formal model for representing software design processes and an extension to requirements modeling using fuzzy sets. The formal model for software design processes enables the software engineer to trace the dependencies between the initial requirements and the resulting implementation model artifacts of the software design process. The fuzzy set extension for modeling requirements enables the software engineer to include all relevant alternatives for the requirements, and also an indication of its relevance to the customer. By relating the fuzzy requirements and system artifacts, identifying the system design that satisfies the customer best can be expressed as an optimization problem. In addition the approach can also be used as a detection mechanism for conflicts in the requirements.

VII. FUTURE WORK

The current approach addresses a number of problems that have been identified in this paper, but it is still limited in its capabilities. At this time the formal model for software design processes can represent typical executions of software design processes, but for a more complete analysis the model needs to be refined. Especially in the area of quality requirements the model should be extended.

Capturing disruptive forces using fuzzy set theory also should be extended in the field of quality requirements. The quality of software architectures is mostly a trade-off between individual quality aspects such as performance, adaptability, etc. The optimizations that can be done should also include these particular constraints and inputs.

In addition, also the definition of a tool set will be future work. To work with the approach that has been proposed in this paper comfortably, tooling support will be very important. Most steps of the approach can be automated (partially), and with tooling support the application of this method can be done more effectively.

REFERENCES

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

[2] Marcelloni, F. & Aksit, M., *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, 1999

[3] Tekinerdogan, B., *Synthesis-Based Software Architecture Design*, Ph. D. Thesis, Print Partners Ipskamp, Enschede, 2000. ISBN 90-365-1430-4, Also available through http://www.cs.bilkent.edu.tr/~bedir/PhDThesis/index.htm.

[4] Broek, P. M. van den & Noppen, J. A. R., *Approximate Reasoning with Fuzzy Booleans*, International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, 2004, Perugia, Italy, in press