Fuzzy requirements

Xiaoqing Frank Liu
Missouri University of Science and Technology, fliu@mst.edu

Follow this and additional works at: http://scholarsmine.mst.edu/faculty_work
Part of the Computer Sciences Commons

Recommended Citation
http://scholarsmine.mst.edu/faculty_work/713

This Article is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. For more information, please contact weaverjr@mst.edu.
Requirement analysis and specification is the first major step in software development. The goal is to develop a requirements' specification that contains all the customers' true needs. The analysis describes quality requirements and their constraints, such as cost and resources. Functional requirements are analyzed in terms of inputs, outputs and their relationships.

Hence, requirements analysis enables software engineers to specify software function and establish software design and implementation constraints. Frederick P. Brooks, Jr. has pointed out that no other part of the work so cripples the resulting system if done wrong. What's more, no other part is harder to rectify later. Defects in the requirements specification can be propagated and amplified in the design and the coding phase. However, many challenges hinder the wide application of requirement engineering methodologies and techniques in software development organizations.

Requirements are sometimes not specified and documented in detail in many software development projects. This makes software validation and software maintenance very difficult. One challenge is that many product requirements are fuzzy in nature. In fact, customers usually describe their requirements in fuzzy terms such as good, high, low, very important. Translating these general terms into design specifications that will accurately create the desired product is difficult. Another challenge is that requirements often conflict with each other, especially those from different perspectives. For example, the efficiency requirement from customers often conflicts with the maintainability requirement from maintenance personnel. Moreover, many conflicts between requirements are implicit and difficult to identify. For requirements with complex relationships between them is an iterative process. Negotiations among conflicting requirements can be very difficult and time-consuming. A formal trade-off analysis can help. A requirement engineering process for analysis is shown in Fig. 1.

Individual requirements from different perspectives are first acquired and represented formally. The relationships between them are then analyzed. Implicit conflicts can be detected based on fuzzy inference techniques. Multiple requirements are collated to formulate an overall system requirement based on fuzzy, multi-criteria, decision making techniques; in which, an objective is often formulated by combining multiple criteria using fuzzy operators, such as fuzzy conjunctives, in fuzzy logic.

Defining the individual and the overall system requirements permits verifying and validating that product requirements have been met. If requirements are evaluated to be unsatisfactory to customers or infeasible to developers, they need to be reformulated. If thought to be unacceptable to the customer, they should be strengthened while maintaining their feasibility. If a problem for the developer, they should be relaxed while maintaining customer satisfaction as much as possible. The resulting overall impact should be assessed fully before requirements are relaxed or strengthened.

**Specifying individual fuzzy requirements using fuzzy sets**

The most effort is placed on representing system requirements. Fuzzy requirements often represent elastic criteria against which the acceptability of a realized target system is judged. Some impose constraints on the system's development process; others describe...
desired properties, such as cost. An example would be:

\[ R_c: \text{The software development cost should be LOW.} \]

Some fuzzy requirements impose constraints on the final system; others describe desired system features. An example would be:

\[ R_e: \text{The software system should be EASY to learn.} \]

The constraint imposed by fuzzy requirement \( R \) can be represented as a satisfaction function, denoted as \( \text{Sat}_R \).

This maps an element of \( R \)’s domain \( D \) to a number in \([0,1]\), which represents how well the requirement is satisfied:

\[ \text{Sat}_R: D \rightarrow [0,1]. \]

In essence, the satisfaction function characterizes a fuzzy subset of \( D \) that satisfies the fuzzy requirement. For example, the elasticity of the requirement \( R_i \) can be captured using the satisfaction function. This corresponds to the membership function of the fuzzy set \( EASY \) in the requirement (see Fig. 2). On the graph, on the vertical axis, 1 represents that the requirement is completely satisfied, while 0 means not meeting the requirement at all.

**Classifying relationships**

There are four types of significant relationships between requirements: 1) conflicting, 2) cooperative, 3) mutually exclusive and 4) irrelevant. The classification is determined by how satisfying one requirement impacts on the satisfaction degree of another requirement.

Two requirements are **conflicting** if raising satisfaction in one requirement “often” decreases the other’s level of satisfaction. If it “always” decreases the satisfaction degree of the other, they are said to be **completely conflicting**.

On the other hand, two requirements are **cooperative** if increasing satisfaction in one often increases the degree the other is satisfied. If the rise in satisfaction of one always increases satisfaction in the other, they are **completely cooperative**.

So how do we characterize partially conflicting and cooperative requirements? A conflicting degree \( 0 \leq \text{conf}(R_i, R_j) \leq 1 \) or cooperative degree \( 0 \leq \text{coop}(R_i, R_j) \leq 1 \) can be attached to a relationship between requirement \( R_i \) and \( R_j \). If \( \text{conf}(R_i, R_j) = 1 \), \( R_i \) completely conflicts with \( R_j \). If \( \text{conf}(R_i, R_j) = 0 \), they do not conflict with each other at all. If \( \text{coop}(R_i, R_j) = 1 \), \( R_i \) is completely cooperative with \( R_j \). If \( \text{coop}(R_i, R_j) = 0 \), they are not cooperative at all. Thus, one can define terms such as “highly conflicting” or “somewhat conflicting” using membership functions in Fig. 3.

Sometimes two requirements cannot be satisfied all at the same time. That is, to satisfy one fuzzy requirement somewhat, the other cannot be satisfied at all, and vice versa. We refer to them as **mutually exclusive requirements**.

Of course, two requirements may be irrelevant. Satisfying one requirement to any degree does not have any impact on the degree another requirement is satisfied.

**Detecting implicit relationships**

Many conflicts are implicit and difficult to identify, especially in large scale software systems. Therefore, it helps to have techniques to identify implicit conflicting and cooperative relationships between requirements. Several heuristics can be used to infer relationships between requirements based on the identified relationships.

• **Heuristic rule 1** (infer relationships from cooperative requirements):
  
  Let \( D \) be either a domain shared by three requirements \( R_1, R_2 \) and \( R_3 \). If requirement \( R_1 \) is cooperative with \( R_2 \) in \( D \), \( R_2 \) is cooperative with \( R_3 \) in \( D \), and \( R_1 \) and \( R_3 \) are not irrelevant, then \( R_1 \) is cooperative with \( R_3 \) in \( D \).

• **Heuristic rule 2** (infer relationships from conflicting and cooperative requirements):
  
  Let \( D \) be either a domain shared by three requirements \( R_1, R_2 \) and \( R_3 \). If requirement \( R_1 \) is cooperative with \( R_2 \) in \( D \), \( R_2 \) conflicts with \( R_3 \) in \( D \), and \( R_1 \) and \( R_3 \) are not irrelevant, then \( R_1 \) is conflicting with \( R_3 \) in \( D \).

• **Heuristic rule 3** (infer relationships from conflicting requirements):
  
  Let \( D \) be either a domain shared by three requirements \( R_1, R_2 \) and \( R_3 \). If requirement \( R_1 \) conflicts with \( R_2 \) in \( D \), \( R_2 \) conflicts with \( R_3 \) in \( D \), and they are not irrelevant, then \( R_1 \) is conflicting with \( R_3 \) in \( D \).

To illustrate the usage of these heuristics, let us consider some typical requirements for software systems. Suppose we plan to develop a billing information system, BILLSYS, with the following customer requirements:

• \( R_1 \): high reliability;
• \( R_2 \): low development cost;
• \( R_3 \): short time to deliver software;
• \( R_4 \): short time to diagnose and repair software;
• \( R_5 \): short system response time, and
• \( R_6 \): low maintenance costs.
High, low and short are fuzzy terms that can be characterized by fuzzy sets. There are conflicts among these requirements. Therefore, it is impossible to satisfy all requirements fully. As a result, trade-offs among them are desirable.

The initial qualitative specification of trade-offs among requirements are given by an experienced requirement analyst and customer (Fig. 4). Here "+" denotes a conflicting relationship between requirements and "-" denotes a cooperative relationship between requirements.

More relationships can be identified between requirements using the heuristic rules for reasoning. For example, using Heuristic rule 2, it has been inferred that $R_1$ and $R_2$ conflict with each other. This is because $R_1$ conflicts with $R_3$ and $R_2$ is cooperative with $R_3$. The refined description of the relationships between requirements is shown in Fig 5. The inferred relationships are represented by dashed lines and the original relationships are represented by solid lines.

## Aggregating multiple fuzzy requirements

The overall system requirement for a system usually has multiple requirements. Fusing them into one overall requirement involves partitioning the requirements into groups and choosing suitable fuzzy aggregation operators. These operators combine requirements in each group and combine different groups. A set of cooperative requirements can usually be satisfied at the same time. Thus, a fuzzy conjunctive operator may be preferred for combining them.

Examples of fuzzy conjunctive operators include MIN and the algebraic sum. Mutually exclusive requirements cannot be satisfied all at the same time. Only one can be satisfied. Therefore, fuzzy disjunctive operators can be used to combine them.

Examples of fuzzy disjunctive operators include MAX and the algebraic sum. Fuzzy compromise operators are usually used to aggregate conflicting requirements to achieve trade-offs. They do this by allowing compensation between requirements. The arithmetic mean is an example of a compromise operator.

Figure 5 shows that the requirements in the billing information system, BILL SYS, can be grouped as $G_1 : (R_1, R_3, R_6)$, $G_2 : (R_2, R_5)$ and $G_3 : (R_4)$. Requirements in each group are cooperative. Thus, they can be aggregated with a fuzzy conjunctive operator. We can use a compromise operator to combine all groups since trade-offs among them are desirable. Therefore, how well the overall customer requirement is satisfied by the billing system can be computed. This is done using the satisfaction degrees of individual requirements and fuzzy conjunctive and compromise operators.

## Summary

Software requirement analysis and specification is an important activity in software development and maintenance. Without explicitly specifying customer's requirements, engineers do not know what to build and how to validate the software. This approach captures the elasticity of requirements using fuzzy sets. It helps to perform a trade-off analysis between requirements. It also facilitates formulating the overall requirement using the results of the trade-off analysis among conflicting requirements. Consequently, it can help create a better system objective that is satisfactory to customers and feasible to developers.

## Read more about it


## About the author

Xiaoqing Frank Liu is currently an assistant professor at the Computer Science Department in the University of Missouri-Rolla. He graduated with a PhD in computer science from the Texas A&M University at College Station in August, 1995. He has conducted research in software engineering, database systems, fuzzy logic and knowledge based systems since 1985. He has published about twenty referred journal and conference papers.