7 Evaluation

Software Engineering Rationale (SER) can play several roles in supporting system evaluation. One is to support the evaluation of decision alternatives by providing the means to capture the arguments for and against each alternative. The rationale can be used to automatically calculate support for alternatives and present it to the developer to assist them in making, or revising, their decisions. Rationale also supports usability evaluation by providing a process for analyzing use scenarios via Scenario-Claims Analysis (SCA) (Carroll and Rosson 1992; Carroll 2002). In this chapter, we discuss a number of approaches for using rationale to evaluate the alternatives to assist with decision-making and also how SCA supports usability evaluation.

7.1 Introduction

7.1.1 Argumentation-Based Rationale

7.1.1.1 Decision-Making in SE

Developing a software system requires making many different types of decisions. Decision-making consists of generating alternative solutions, or approaches, identifying the reasons for and against these alternatives with respect to evaluation criteria, and selecting the "best" alternative based on these reasons and criteria.

Decisions made during software development affect many aspects of the development process and the developed product:

- *Product decisions* – What is being developed? Who should it be marketed to? Who is the customer/user? What are the requirements? Where does the system need to run?

- *Process decisions* – How should the system be developed? What process model should be followed? When should versions be released? What level of documentation needs to be produced? What is the testing strategy?
- *Management decisions* – How should the development team be structured? Who should be on it? What resources should be made available to the project?
- *Development decisions* – What development tools should be used? What components can be integrated? What is the system architecture? What are the data structures?

These are only a few examples of the many different decisions and decision types that need to be made. The results of each decision may be important to a different collection of stakeholders. For example, a system user would be interested in decisions regarding functionality but not as concerned with process models or data structures.

Each decision also has several different types of criteria that influence alternative selection. These criteria include functional requirements, non-functional requirements, assumptions, dependencies, risk, and constraints. The degree to which an alternative meets or fails to meet criteria may vary as well as the certainty in that evaluation. The decision-making task is further complicated by criteria differing in importance.

7.1.1.2 Rationale and Decision Support

The information generated and used during decision-making consists of decisions required, alternatives considered, reasons for and against the alternatives, and the criteria used for evaluation. This information forms the rationale for the choices made as a software system is developed and maintained. The rationale can be used to evaluate these choices and support the human decision-maker by advising them if their decisions are inconsistent with the rationale that they recorded.

The rationale can both be evaluated itself and used to support evaluation of the decisions made. Evaluating the rationale itself involves syntactic checks on the structure of the rationale and semantic checks that analyze its content (Conklin and Burgess-Yakemovic 1996). An example of a syntactic check would be to look for missing information, such as decisions where alternatives were not chosen, while semantic checks would look for contradictions in reasoning, such as arguments that are used to both support and refute an alternative.

Evaluating the decisions made involves using the rationale to indicate which alternatives are preferable over other alternatives and why. The

method of evaluation and the inputs to each method vary depending on the complexity of the problem and the types of information available. Decisions may involve looking at different types of criteria (functional and nonfunctional requirements, assumptions, constraints, etc.), conflicting opinions from multiple decision-makers, uncertainty, shifting priorities, and missing or incomplete data. The evaluation of an alternative may change over time as well so there also needs to be a way to determine when re-evaluation is necessary.

Selecting an evaluation method requires tradeoffs between the amount of information required to use a method, the computational requirements (if evaluation is computer assisted), and the required rigor. The value of the evaluation is directly dependent on the ability to capture the rationale in sufficient detail to support the method chosen. This chapter will describe several alternative methods for computer-assisted evaluation of argumentation-based rationale in order to augment human decision-making.

7.1.2 Scenario-Based Rationale

Scenario-based design (Carroll and Rosson 1992) uses scenarios as the starting point of design. Scenarios describe how the user goes about performing a task using the artifact that is being designed. Scenarios are valuable because they are a way to take knowledge about system use that is tacit, such as assumptions, and make it concrete (Carroll 2000). Scenario- Claims Analysis (SCA) is the process of analyzing scenarios to extract "claims"—implicit causal relations that describe the desirable and undesirable consequences of design features described in the scenario (Carroll 2000). These claims describe the rationale behind the scenario—why the scenario operates the way that it does. Later in this chapter we will describe how SCA can be used in evaluation.

7.1.3 Objectives of This Chapter

This chapter discusses the evaluation of and using argumentation rationale as well as using rationale generated during scenarios-claims analysis for system evaluation. For the argumentation evaluation, this chapter looks at two types of evaluation: evaluation of the rationale itself for completeness and correctness and using the rationale to evaluate decision alternatives. For alternative evaluation, it concentrates on three issues: comparing the alternatives, combining inputs from multiple developers, and handling uncertainty. The focus is primarily on computational evaluation using argumentation. The scenarios-claims analysis section describes how

analyzing scenarios to extract claims is a form of evaluation that can be fed into the development of testing scenarios to gather evaluation data.

7.2 Evaluating the Rationale

Many rationale representations take the form of semiformal argumentation. This format is a natural way to express the decisions, alternatives, and arguments and can be read easily by people and interpreted by computers. There are many argumentation formats which date back to Toulmin's warrants, claims, datums, backings, and rebuttals (Toulmin 1958). These include the Issue-Based Information System (IBIS) notation (Kunz and Rittel 1970), Questions, Options, and Criteria (QOC) (MacLean et al. 1989), the Decision Representation Language (DRL) (Lee 1991), WinWin (Boehm and Ross 1989), the Design Recommendation and Intent Model (DRIM) (Peña-Mora et al. 1995), and numerous notations that extend these representations and Rationale Management Systems that use them.

In this section we describe two types of evaluation of the rationale: checking the rationale for completeness and checking the rationale for correctness.

7.2.1 Completeness

Completeness checking over the rationale looks primarily at the syntax checks, or what Conklin and Burgess-Yakemovic referred to as "well-formedness checks" on the syntax and structure (Conklin and Burgess-Yakemovic 1995). Completeness checking typically does not ensure that all the rationale for the system has been collected but instead checks to see if there are any holes in the rationale that is present.

There are many possible checks, or inference, that can be performed on the rationale. The availability of these checks depends on the richness of the representation format. There are some checks, however, that can be made over most argumentation-based formats. These include: checks to ensure that there are alternatives proposed for each issue/decision, checks to see if an alternative has been selected for each issue/decision, checks to see if alternatives are selected that do not have any arguments (in either direction), and checks to see if alternatives are selected that only have arguments objecting to them with none in support.

7.2.2 Correctness

While syntactic inference looks at the structure of the rationale, semantic inference looks at the contents. The ability to do this is limited—comparing information within the rationale requires that a common vocabulary be used. The Knowledge-Based Design System (KBDS) (Bañares-Alcántara et al. 1995; King and Bañares-Alcántara 1997), which extends IBIS, used keywords to check argument consistency. Inferencing over Rationale (InfoRat) (Burge and Brown 2000) created a common vocabulary of arguments. SEURAT's RATSpeak (Burge 2005), an extension of DRL, extended this vocabulary into an argument ontology that described a hierarchy of reasons for making software decisions at different levels of abstraction. Using a common vocabulary within arguments allows for inferences that look for contradictions such as using the same argument for and against an alternative.

Some rationale representations, such as RATSpeak, capture dependencies between alternatives. These relationships can be used to check if there is a dependency violation where an alternative is chosen that conflicts with another selected alternative or requires an alternative that has not been selected. If the requirements are explicitly captured in the rationale, the rationale can also be used to detect if an alternative has been selected that has an argument indicating that it violates a requirement. Some representations, such as RATSpeak and REMAP (Ramesh and Dhar 1992) represent requirements as explicit types of rationale entities. QOC and DRL can do this less directly by having QOC's critieria and DRL's goals contain requirements.

Another type of semantic inference is to detect if there have been any tradeoff violations. Many arguments captured in rationale describe qualities that are "traded off" when making decisions. Known tradeoffs that apply at a system-wide level can be captured as "background knowledge" in InfoRat (Burge and Brown 2000) and SEURAT (Burge and Brown 2004). An example of a software tradeoff would be the ease of coding an alternative versus its flexibility. In most cases, the more flexible design is likely to be more difficult to implement initially. The rationale can be evaluated to check to see if there were alternatives with arguments that claim flexibility where there were no opposing arguments warning of the potentially longer development time. The rationale can also be checked to ensure that alternatives do not claim to be flexible and easy to implement. The developer can override the results of these inferences in cases where there are exceptions to the general rule.

7.3 Evaluating the Decisions

Software development decisions are often multidimensional, i.e., decision outcomes involve multiple dimensions. Vetschera (2006) states four contributors to multidimensionality: alternatives impact multiple criteria, uncertainty of alternative outcomes, multiple stakeholders, and alternative outcomes that vary over time. The rationale can serve as inputs to many different evaluation methods. In this section we will describe some of the methods and issues and how rationale has been, or can be, used to support them.

7.3.1 Comparing Alternatives

There are many possible methods that can be used to compare alternatives. The choice of method depends on the information available as input (i.e., the richness of the rationale representation and the fidelity of the data) and the results of tradeoffs between computational complexity and semantic justification of the results. Methods require extensive calculation, evaluations for each criteria, multiple pairwise comparisons (which do not scale well if the number of alternatives is large), or quantitative measurements (which may not be available).

The simplest evaluation involves arguments that are either for or against an alternative. The support for the alternative consists of the difference in the pro and con arguments divided by the total number of arguments (Fox and Das 2000). This method assumes that all arguments are equally important.

Many evaluation methods fall into the category of Additive Sum Methods (Vetschera 2006) where the alternative utility is calculated using a weighted value for each argument. The simplest form, Weighted Sum Method (WSM), is used by several rationale-based systems including HERMES (Karacapilidis and Papadias 2001), InfoRat (Burge and Brown 2000), and SEURAT (Burge and Brown 2004; Burge and Brown 2006). In these systems, each argument is given a weight to indicate its relative importance. Assigning these importance values is not a simple task—the values could be given relative to the specific decision or could apply system wide. In HERMES, the evaluation involves the sum of the weights in favor minus the sum of the weights against. In InfoRat and SEURAT, the weight is applied to (multiplied by) a numerical amount indicating the degree to which the alternative affects the criteria. Additive Sum Methods can be evaluated for sensitivity to any of the weight values by plotting the result when expressed as a function of that weight (Vetschera 2006).

Determining the appropriate weights can be difficult and the results of the summations do not always accurately reflect the utility. Vetschera (2006) demonstrates that a summation of weights may result in avoiding compromise alternatives. He suggests correcting this by adding an additional partial utility function to each argument in addition to the weight. This would be especially valuable when different types of arguments are involved. A violation to a functional requirement, for example, should have a significantly higher impact on the decision than other types of arguments.

The Analytic Hierarchy Process (AHP) (Saaty 1980) is another method for comparing alternatives. In this method, pairwise comparisons are performed between all alternatives examined against all relative criteria. As with the other weighted methods, criteria are given different weights. AHP has been applied to software engineering decision problems such as prioritizing software requirements (Karlsson and Ryan 1997) and choosing software products (Lai et al. 2002). This method requires that the same criteria be used to weigh each alternative. The significant disadvantage to this method is that it does not scale well when comparing large numbers of alternatives.

7.3.2 Combining Inputs from Multiple Developers

Rationale can be a valuable tool for collaboration and negotiation. This was demonstrated with gIBIS (Conklin and Burgess-Yakemovic 1995), Compendium (Buckingham Shum et al. 2006), and SHARED-DRIM (Peña-Mora et al. 1995). The argumentation can serve as a natural medium for the different contributors, or stakeholders, in a project to state their views on alternatives under consideration. This does pose an interesting challenge for evaluation: how can conflicting beliefs and opinions be aggregated? Factors that contribute to the difficulty include the differing expertise of developers and differing degrees of confidence in evaluations. There could potentially be arguments refuting and supporting other arguments as developers debate each other's arguments. The developers may not disagree with the arguments themselves but may not agree with information such as the importance of the argument criteria, the degree to which the alternative meets the criteria, or the plausibility of the argument.

Combining conflicting beliefs has been an important topic of research in economics, statistics, and artificial intelligence. How can conflicting beliefs be combined to reach some version of Pareto optimality? There are numerous impossibility theories (Arrow 1963; Mongin 1998; Blackorby et al. 2000) but also many approaches that avoid impossibility by methods

that include restricting the Pareto condition (Gilboa et al. 2004) and understanding that not all expert opinions should carry the same weight (Maynard-Zhang and Lehman 2003).

As with other evaluation methods, the belief combination method used will depend on the type of information available and the amount of computation that needs to be performed.

The field of economics has studied this issue when looking at preference aggregation (Andreka et al. 2002; Hild et al. 1998; Harsanyi 1955). Lexicographic ordering is another method used to combine preference operations (Andreka et al. 2002). Clemen and Winkler (1999) describe many different methods for combining probability distributions from multiple experts when performing risk analysis/assessment. These methods include the linear opinion pool (Stone 1961), which uses a weighted sum incorporating the "quality" of each expert and Bayesian updating (Winkler 1968). In AI, combining beliefs is necessary when performing ensemble learning (Pennock et al. 2000) and when merging information from multiple data sources (Booth 2002; Meyer et al. 2001).

The most promising methods are those that take advantage of information about the experts—their level of expertise, their experience, their reliability, and potentially even their influence. When experts disagree and their negotiation is captured in the rationale, they are unlikely to be given equal weight in the decision-making process and it is important to utilize this information when proposing decisions. Knowledge about the expert providing the information can be used to provide a "pedigree" for the information. This pedigree information is used in belief fusion (Maynard-Ried II and Shoham 2001) to combine beliefs from different experts.

7.3.3 Handling Uncertainty

Software decision-making needs to address the uncertainty surrounding the development process. Uncertainty can refer to many things: vagueness, imprecision, inconsistency, incompleteness, or ambiguity (Parsons 2001). Ziv et al. (1996) describe four domains where uncertainty is an issue: requirements analysis, transitioning from requirements to design and code, uncertainty in re-engineering, and uncertainty in reuse. This uncertainty can come from many sources. Three examples are the problem domain ("real world"), the solution domain, and the humans participating in the development process (Ziv et al. 1996). Lehman and Fernández-Ramil (2006) are concerned with the impact of assumptions which may change over time. When assumptions that were the basis of software decisions no longer hold they can result in system failure. A high-profile example of

this is the loss of the Ariane 5 rocket (Nuseibeh 1997; Lehman and Fernández-Ramil 2006). Decisions must also be made in the presence of incomplete information and may require revisitation later in the process when more is known about the problem.

The presence and role of uncertainty in making software decisions can be captured in the rationale. Systems such as REMAP (Ramaesh and Dhar 1994) and SEURAT (Burge and Brown 2006) explicitly represent assumptions in the rationale. SEURAT supports the ability to disable an assumption and re-evaluate the support level for any alternatives referring to it. If the assumption refers to an event that is expected to be true at some point in time, it should be given a time stamp to remind the designer that the decision should be re-examined (Burge et al. 2006).

The need to gather additional information can be captured in the form of questions as is done in DRL/SIBYL (Lee and Lai 1996) and SEURAT. These systems use questions to describe what information is required to make a decision or evaluate an argument and to indicate, if known, the likely sources of that information. SEURAT will report all unanswered questions as errors until they are resolved.

Uncertainty in arguments is captured in DRL, SEURAT, and the Knowledge-Based Decision System (KBDS) (King and Beñares-Alcántara 1997) using plausibility, or uncertainty, values for each position. SEURAT and KBDS use these values, along with weights applied to each criteria, to rank the alternatives.

Using a plausibility value as a weighting factor in a weighted sum evaluation is one approach to incorporating the effect of uncertainty in evaluation. There are numerous other approaches that can also be used. Parsons and Hunter (1998) divide formalisms for uncertainty handling into two "camps"—the "numerical camp" that uses quantitative methods and the "symbolic camp" that uses logical, or qualitative, methods.

Numerical, or quantitative, measures include those based on probability theory, evidence theory, such as Dempster–Shaefer (Shafer 1976), and possibility theory (Zadeh 1978), based on fuzzy sets (Zadeh 1965). These methods share several drawbacks: the potential difficulty in obtaining the "numbers" (probabilities, possibilities, and distributions), the risk of comparing different types of beliefs, and the possibly significant computational expense (Parsons and Hunter 1998).

Two quantitative methods frequently used in decision-making are influence diagrams and decision trees (Clemen and Reilly 2001). Influence diagrams capture the decision structure as decisions, change events, the desired outcome (payoff node), and intermediate consequences/calculation nodes. The different alternatives, outcomes, and consequences are present as tables within the nodes. Decision trees express this information more

explicitly in the structure where decisions branch to choices and "chance events" branch to outcomes. Decision trees are often used to compute the "Expected Value" of a decision. Decision trees have been used to support Value Based Software Engineering by calculating the value of a software project (Erdogmus et al. 2006).

Qualitative methods are those that work either without numeric information or with only some numeric information (Parsons 2001). In some cases, these methods are variants on quantitative methods. Qualitative Probabalistic Networks (Wellman 1990; Parsons 2001) are a variant on influence diagrams where the influence of one node on another is expressed qualitatively as being positive or negative.

Defeasible reasoning is a form or reasoning that accounts for the need to retract initial conclusions when new information is obtained (Parsons 2001). Parsons describes three forms of defeasible reasoning: logic, probability, and argumentation. Argumentation can support reasoning under uncertainty either by calculating the "safety" of arguments based on the presence of counterarguments or by adding a confidence factor indicating the degree to which the argument is believed to be true (Parsons and Hunter 1998).

The ability to re-evaluate beliefs (in our case, in the form of alternative evaluations) in the face of changing assumptions is similar to work done using Truth Maintenance Systems (TMSs) (Doyle 1979; de Kleer 1986). In rationale-based systems, changing assumptions and NFR priorities can be used to re-evaluate alternatives to indicate where changes might be advisable. This process would probably stop short of actually retracting the selection of alternatives but would instead inform the developer of the potential problems.

7.4 Scenario-Based Evaluation

As described earlier, scenario-based design uses interaction scenarios as an informal and holistic working representation in requirements analysis and design. The scenarios depict user interactions observed, predicted, and proscribed, and provide a medium for exploring first-order consequences and interactions of envisioned design features. For example, one obstacle to code reuse is that it is often difficult for programmers to find examples of how a given object or module is to be reused; thus, they must work directly from code definitions, which is a strong deterrent to reuse (Rosson and Carroll 1996). In designing support for code reuse, one might envision

and analyze a scenario in which part of the documentation for software objects and modules is pointers to commented example uses of that code.

The scenario might be the starting point for a design solution (e.g., part of the programming environment), but it also helps to evoke and evaluate rationale. For every design feature in an envisioned scenario, one can identify desirable and undesirable consequences. Thus, providing example-based usage documentation is indeed a resource to programmers: they quickly learn to borrow usage protocol directly from example uses (Rosson and Carroll 1996). This is an upside consequence of the design solution. However, there are also downsides, risks, or costs entailed by the design solution: positing new documentation raises the question of who will create and maintain the documentation, and of how and where it will be stored and accessed.

Evaluating a design solution and its rationale by analyzing interaction scenarios is an example of what Scriven (1967) called *intrinsic evaluation*. Intrinsic evaluation assesses solution properties analytically, instead of empirically measuring performance characteristics. Intrinsic evaluation is often more illuminating than empirical evaluation, since it constructs an arbitrarily rich decision space of implicit tradeoffs. Intrinsic evaluation can also be less expensive, but it is always less definitive in that it cannot determine the exact cost parameters in the tradeoffs. In the example of reuse documentation, the analysis identified valid desirable and undesirable consequences of the design solution, but only a large-scale implementation could show whether the benefits outweigh the costs.

7.5 Summary and Conclusions

Here we have described two ways that SER can be used to support software evaluation: supporting decision-making by evaluating decision alternatives and supporting usability evaluation through scenarios claims analysis. There are many different types of decisions made during software development for which rationale can be captured. This rationale can then be used to evaluate these decisions to ensure that choices made do not contain flaws that can be detected via computation. This evaluation is not necessarily used to make the final decision but can be used as a verification step. Evaluation is also an important aspect of change analysis that provides a means for accessing the impact of changing criteria on the recommended decisions. Scenarios and SCA evaluate how the system supports its goals in operation by providing a framework for evaluating usability based on the scenarios and the accompanying usability rationale.