

# Supporting Requirements Traceability with Rationale

Janet E. Burge  
Miami University  
230H Kreger Hall  
Oxford, OH 45056 USA  
+01 513-529-9760  
burgeje@muohio.edu

David C. Brown  
Worcester Polytechnic Institute  
100 Institute Road  
Worcester, Massachusetts USA  
+1 508-831-5618  
dcb@cs.wpi.edu

## ABSTRACT

Requirements, both functional and non-functional, are the driving force behind the many decisions required to develop a software system. These decisions, alternative solutions, and the reasons behind them, can be captured in the *rationale* for the software system. The rationale contains the arguments for and against each alternative solution, which in turn relate to the functional and non-functional requirements. When this information is captured in the rationale for a system, it provides an alternative mechanism for tracing the impact of functional and non-functional requirements on the software system. In this paper we describe how the Software Engineering Using RATIONale system (SEURAT) supports requirements traceability by incorporating functional and non-functional requirements into the argumentation.

## Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specification – *methodologies, tools*. D.2.6 [Software Engineering]: Programming Environments – *integrated environments*. I.2.4 [Artificial Intelligence]: Knowledge Representation – *representation languages*.

## General Terms

Documentation, Design.

## Keywords

Design Rationale, Requirements Traceability, Inference.

## 1. INTRODUCTION

Requirements traceability refers to the ability to trace from the system requirements through the rest of the development process. Traceability is necessary to both ensure that requirements have been met and to assess the impact and consequences of requirements changes [33]. Traceability can be viewed as going in two directions: Pre-RS (Requirements Specification) traceability that traces a requirement back to its origins and Post-RS traceability that traces a requirement forward from its specification through implementation and test [21].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

TEFSE/GCT'07, March 22-23, 2007, Lexington, KY, USA.  
Copyright 2007 ACM ISBN 1-59593-6017/03/07...\$5.00

Requirements are typically broken into two categories: functional requirements (FRs) that describe what the system does (its functionality) and non-functional requirements (NFRs) that describe how the system should accomplish those functions given “the constraints of a non-ideal world” [38]. Non functional requirements often refer to software quality and are related to Boehm et al.’s “Quality Characteristics.” [4]. Roman [37] describes NFRs as restricting the types of solutions under consideration. NFRs are not directly related to specific system components and often involve aggregate system behavior [31]. Despite their importance in overall software quality, NFR traceability is typically not captured [11].

The requirements, both functional and non-functional, are the driving force behind the many decisions required to develop a software system. These decisions, alternative solutions, and the reasons behind them, can be captured in the *rationale* for the software system. The rationale contains the arguments for and against each alternative solution, which in turn relate to the functional and non-functional requirements. When this information is captured in the rationale for a system, it provides an alternative mechanism for tracing the impact of the FRs and NFRs on the software system.

The Software Engineering Using RATIONale system (SEURAT) [7][8] was originally developed to support software maintenance by allowing the software maintainer to inference over the captured rationale to detect the impact of changing development priorities on existing software. In this paper, we will describe how this system also supports requirements traceability by incorporating functional and non-functional requirements into the argumentation. Section 2 of this paper describes SEURAT and how it supports software development. Section 3 describes how SEURAT supports traceability of functional requirements to their origin, supports NFR traceability by providing an Argument Ontology, and supports traceability of both requirement types to the software, via the rationale. Section 4 describes related work and Section 5 presents the summary and conclusions.

## 2. USING RATIONALE IN SOFTWARE DEVELOPMENT

The rationale behind a software system describes what decisions were made during its development, what decision alternatives were considered and the reasons for their selection or rejection. Rationale differs from other forms of documentation by describing what *was considered but not* done as well as what *was* done.

The ability to investigate and evaluate decision alternatives is a key element in Decision Analysis and Resolution (DAR), one of

the process areas in the Capability Maturity Model Integration (CMMI) [13]. This is especially critical for high risk decisions where the consequences of making an incorrect choice could be catastrophic.

Unfortunately, despite agreement that rationale is valuable, it is typically not captured in practice. One obstacle to this is a lack of integrated tool support. Another obstacle is motivation: rationale capture is viewed as time consuming so in order to make the capture worth while there needs to be compelling uses for the rationale once it has been obtained.

To address these issues, the primary goal of the SEURAT system was to provide the means to investigate uses of rationale that went

beyond simply presenting it to the developer. A secondary goal was to provide a system that integrated with a standard development environment.

SEURAT is implemented as an Eclipse ([www.eclipse.org](http://www.eclipse.org)) plugin. Integration with a development environment is the key to a successful rationale system. It makes the capture and use less disruptive by not forcing the developers to use an additional tool, and provides the means to inform developers that rationale is available when a development artifact is edited. Figure 1 shows SEURAT integrated with the Eclipse IDE.

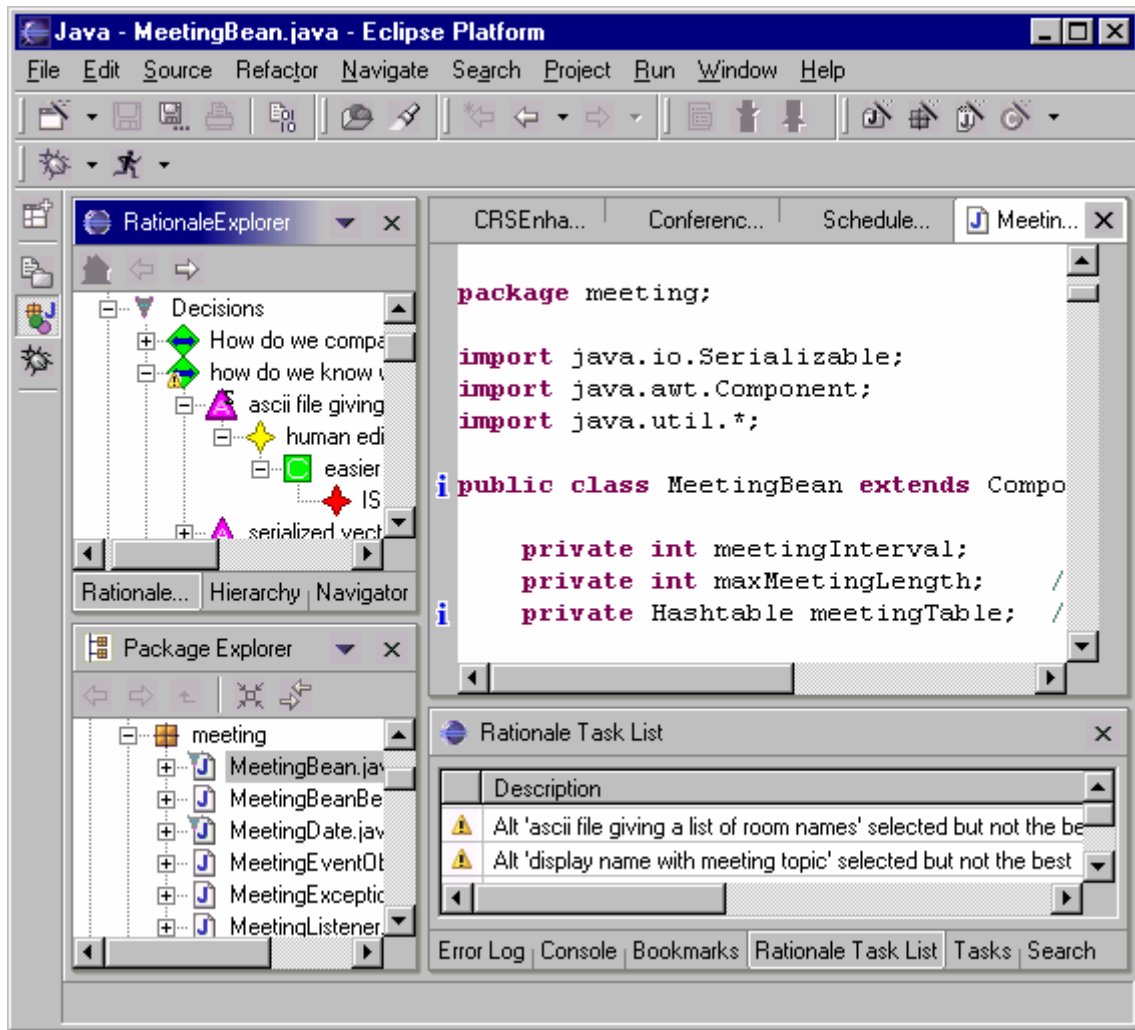


Figure 1. SEURAT User Interface

SEURAT participates in the development environment in three ways: a Rationale Explorer (upper left pane) that shows a hierarchical view of the rationale and allows display and editing of the rationale; a Rationale Task List (lower right pane), that shows a list of errors and warnings about the rationale; and Rationale Indicators that appear on the Java Package Explorer (lower left pane) and in the Java Editor (upper right pane) to show

whether rationale is available for a specific Java element. The examples in this paper come from a meeting scheduling system. Note that the screenshots are in color. This makes the icons much easier to distinguish than when reproduced in black and white.

The software developer enters the rationale to be stored in SEURAT while the system the rationale describes is being

developed. SEURAT supports this by providing rationale entry screens for each type of rationale element.

The rationale representation used in SEURAT is a semi-formal argumentation format. This structure was chosen because it can be easily read (and written) by people as well as by the computer. The representation is described in detail in [7][8]. Key elements are the decisions that need to be made, the alternative solutions for those decisions, and the arguments for and against them. Requirements are also included in the representation in two ways. Functional requirements appear both on their own, with associated argumentation justifying the requirement, and in arguments for and against decision alternatives. Non-functional requirements, stored in an Argument Ontology that provides NFRs at different levels of abstraction, are associated with Claims that appear in arguments for and against Alternatives.

As the developer enters, or revises, the rationale, SEURAT inferences over that information to check for structural errors in the rationale, such as missing arguments, and possible errors in the reasoning, such as selecting an alternative that is not well supported or selecting an alternative that violates a functional requirement.

The key behind many of the inferences supported by SEURAT is the representation chosen for the arguments. Five types of arguments are supported by SEURAT:

- arguments that an alternative supports or denies an assumption;
- arguments that an alternative is dependent on, or in opposition to, another alternative;
- arguments that an alternative satisfies, addresses or violates a requirement;
- arguments that an alternative supports or denies a claim; where claims relate directly to non-functional requirements, and;
- arguments that support or deny (refute) other arguments.

Figure 2 lists the contents of an argument. The plausibility factor provides the certainty of the evaluation, the amount provides a

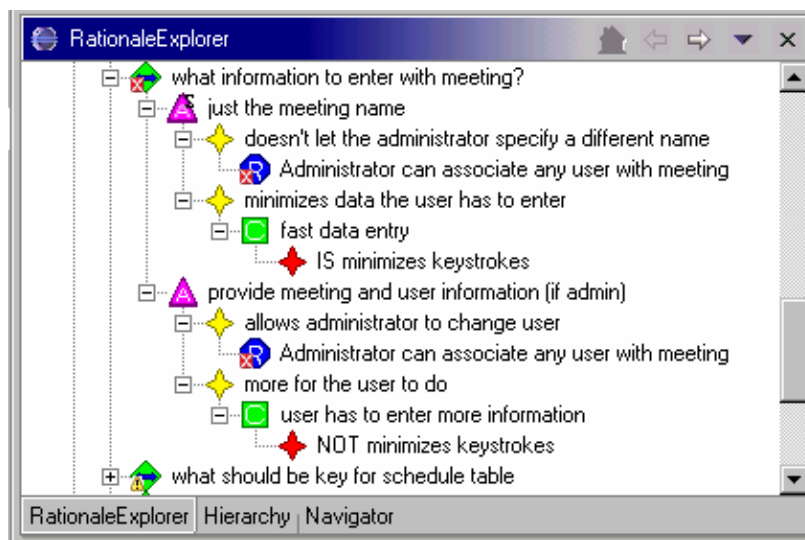
numeric score for how much the argument applies to the alternative, and the importance describes how important the argument is to this decision.

name:	<name of the argument>
description:	<more detailed description of the argument>
category:	For alternatives <supports, denies> For arguments <supports, denies> For assumptions <supports, denies> For claims <supports, denies> For requirements <addresses, satisfies, violates>
plausibility:	<certain, high, medium, low, not>
amount:	<numeric: 1-10>
importance:	<essential, high, moderate, low, not, default>
type:	<alternative, argument, assumption, claim, requirement>

**Figure 2. Argument Contents**

Arguments are evaluated using the Weighted Sum Measurement (WSM) technique, where a score is calculated for each argument by multiplying the different factors together and the alternative score is the difference between the sum of the arguments for and against the alternative. This technique is the simplest form in the category of Additive Sum Methods [39].

Figure 3 shows an example of rationale displayed in the SEURAT Rationale Explorer. In this example, which contains rationale for a conference room scheduling system, the alternative chosen violates a requirement. This is indicated with an error icon overlaid on the requirement icons and on the decision where the incorrect alternative was selected. The error icon is a small square with an "X" inside it and is shown on the requirement "Administrator can associate any user with meeting." The alternative "just the meeting name" violates that requirement and has been selected, resulting in an error icon appearing on the corresponding decision, "what information to enter with meeting?"



**Figure 3. Rationale Explorer with Requirement Violation**

The ability to inference over the rationale and report any problems, or potential problems, found supports the ability to make changes in the requirements, both functional and non-functional, and trace their impact on the software system. SEURAT will report an error when a requirement is violated by an alternative selection, as shown above, and will report warnings if an alternative is selected that is not as well supported (by arguments referring to FRs and NFRs) as other available options.

The following section describes in more detail how requirements are used in SEURAT.

### 3. SUPPORT FOR TRACEABILITY

#### 3.1 Functional Requirements

Functional requirements play an important role in the rationale captured by SEURAT. Requirements appear in arguments for and against different decision alternatives. An alternative can address, satisfy, or violate a functional requirement. If an alternative violates a functional requirement, and that alternative is selected, SEURAT will report the violation to the user as an error, both via the Rationale Explorer as shown in Figure 3 and on a Rationale Task List that lists all problems (errors and warnings) discovered in the rationale.

SEURAT also allows the capture of the rationale behind the requirements. This serves several functions. One is to capture the relationship between functional and non-functional requirements. For example, there may be functionality that is necessary in order to address a non-functional requirement such as security or safety. Arguments for requirements describing that functionality would refer to the appropriate NFRs as justifications.

Another function of requirements rationale is to relate the functional requirements with the original customer requirements from which they are derived. This can help to support “rich traceability” [17] while providing Pre-RS traceability to the source of the requirements.

One of the inferencing capabilities provided by SEURAT is to allow requirements to be enabled and disabled. Either of these actions will re-evaluate all alternatives referring to that requirement. SEURAT will report a warning for any selected alternative that is not given the highest evaluation score for the corresponding decision. This could indicate where some of these decisions should be reconsidered taking into account the different requirement status.

SEURAT does not currently output a traceability matrix mapping requirements to alternatives. It does, however, allow queries that provide similar information. Adding the ability to generate the matrix would be a straightforward enhancement to the tool.

#### 3.2 Non-Functional Requirements

Non-functional requirements can have a significant impact on the system, yet are often not documented and are even more difficult to trace than functional requirements. SEURAT supports the traceability from NFRs to code by using them as arguments for and against alternatives. This is done through arguments about Claims which then map to elements in an Argument Ontology containing a hierarchy of NFRs. This allows several interesting

inferences to be performed that assist in determining the impact of different NFRs on the code.

The following sections describe the Argument Ontology and how it can be used to support traceability of NFRs to alternatives.

##### 3.2.1 Argument Ontology

The SEURAT argument ontology was created to provide a common vocabulary to describe the NFRs that may appear as reasons for or against decision alternatives. There are several ways that NFRs have been organized. For example, CMU’s Quality Measures Taxonomy [14] organizes quality measures into Needs Satisfaction Measures, Performance Measures, Maintenance Measures, Adaptive Measures, and Organizational Measures. Bruegge and Dutoit [6] break a set of design goals into five groups: performance, dependability, cost, maintenance, and end user criteria. Chung, et al. [9] provides an unordered list of NFRs, which provided many elements for the Argument Ontology, as well as specific criteria for performance and auditing NFRs. The ISO/IEC 9126 software product quality standards [25] give six characteristics (functionality, reliability, usability, efficiency, maintainability, and portability) as well as 27 sub-characteristics.

The Argument Ontology starts with the more abstract desirable “ilities” [20] and breaks them down into how these properties can eventually be achieved. The ability to capture NFRs at different levels of abstraction is useful because more abstract reasons can be applied to early decisions and more specific reasons applied when the alternatives become more concrete. Figure 4 gives the top levels of the Argument Ontology while Figure 5 shows the sub-criteria for “End User Criteria.”

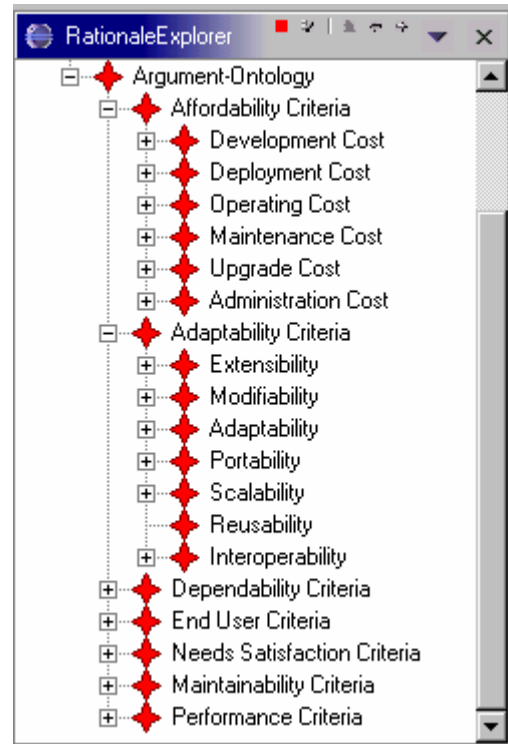


Figure 4. Top levels of the Argument Ontology

The current ontology contains 277 terms and is described in more detail in [7]. To develop the ontology, we took a bottom-up

approach by looking at what characteristics a system could have that would support the different types of software qualities. This involved reviewing literature (see [7] for details) on the various quality categories to break the general “ility”-level qualities into more specific software characteristics.

For example, one quality attribute that is a factor in design decisions is scalability. We looked to see what might contribute toward scalability in a software design and added these attributes to the ontology. One way to increase scalability is to minimize the number of connections a system must set up, another is to avoid using fixed data sizes that may limit the capacity of the system. Our aim was to go beyond the idea of design goals or quality measures to look at how these qualities might be achieved by a software system.

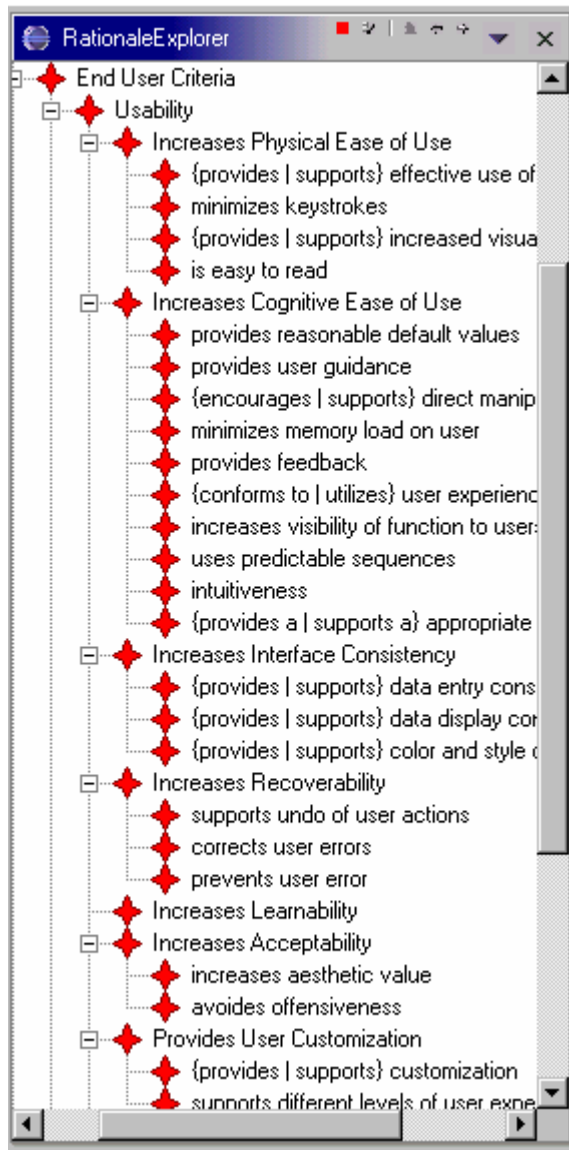


Figure 5. Sub-criteria for usability

After determining a list of detailed reasons that might be used for choosing one alternative over another, an Affinity Diagram [24] was used to cluster similar reasons into categories. These

categories were then combined again. The more abstract levels of the hierarchy were based on a combination of the NFR organization schemes listed earlier. Also, NFRs from the Chung list were used to fill in gaps in the ontology.

The Argument Ontology is not intended to be a complete set but a starting point from which developers and researchers can add additional terms as needed. It is also not a strict hierarchy. For example, while performance is an important goal in and of itself, increasing performance also contributes toward the scalability of a system. This means that performance (and its children) appears both as a top level category in the ontology and as a sub-category under scalability.

### 3.2.2 Using NFRs

A key inference performed by SEURAT is the evaluation of the support for each alternative. This takes into account the importance of each argument. The importance can be entered by the developer or, in the case of arguments referring to NFRs, inherited from the Argument Ontology. The developer can override this importance at either the Claim level (Claims can be shared by multiple arguments) or at the Argument level (to affect only that specific argument). Figure 6 shows an example of the Argument-Claim-Ontology level relationships.

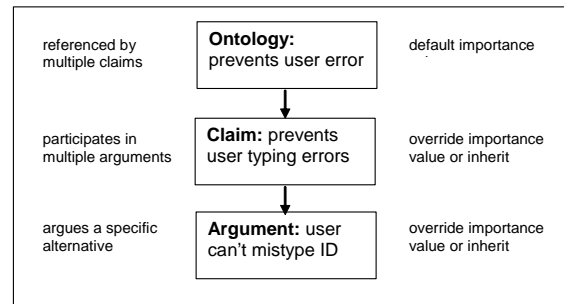


Figure 6: Argument-Claim-Ontology Relationship Example

Changing the importance at the Argument Ontology level will then propagate that change to any arguments where the importance has not been overridden by the developer. This is a key capability of SEURAT and allows the user to perform “what-if” inference to see how the design and implementation might need to be changed if an NFR becomes more or less important to the system. This is done by using the rationale to trace from the alternatives affected by the requirement change to the code or other development artifacts that implement them.

SEURAT can be used to examine which Argument Ontology elements appear the most frequently in the rationale. This can be done for all the rationale or for rationale for selected alternatives only. Figure 7 shows the results of a query over selected alternatives for a Conference Room Scheduling System. In this example, “Reduces Development Time” was the most frequently referenced element in the Argument Ontology. If the importance of this argument is reduced from its initial level of “HIGH” to the lowest possible importance of “NOT” and the arguments are re-evaluated, the SEURAT Rationale Task List indicates four new warning messages, shown in Figure 8, indicating that four decisions should be re-evaluated because the alternative selected is no longer the one with the most support.

This alternative re-evaluation capability becomes increasingly valuable as a system evolves and as priorities change. The ability to find where corners may have been cut due to time constraints (as shown as shown in the previous example) is certainly useful.

There also may be cases where scalability and performance may not have been priorities initially but come into play as the system is deployed.

Ontology Entry	Total	For	Against
Reduces Development Time	12	9	3
minimizes keystrokes	5	5	0
intuitiveness	3	3	0
{allows   supports} additional users	2	2	0
reduces coupling	2	2	0
{provides   supports} code readability	2	1	1
Increases Scalability	1	1	0
provides reasonable default values	1	0	1
minimizes connections to be set up	1	0	1
{is a   uses a} efficient algorithm	1	1	0
provides user guidance	1	1	0
{provides   supports} effective use of s...	1	1	0

Figure 7. NFR Frequency in Arguments

Description	Rationale
Alt 'ascii file giving a list of room names' selected but not the best	how do we know what the confere...
Alt 'display name with meeting topic' selected but not the best	how to display which user "owns" ...
Alt 'error line on main display' selected but not the best	how to display error messages
Alt 'start date' selected but not the best	what should be key for schedule ta...

Figure 8. New Warnings when Importance Changes

### 3.3 Traceability via Rationale Associations

The traditional use of traceability is to determine the impact of the requirements on the software product to ensure that all requirements have been met and to determine the probable impact if a requirement changes. Earlier we described how requirements can be traced to alternatives. Traceability support continues by allowing the developer to associate each alternative with the development artifact that implements it. These traceability links traverse from the requirements to the alternatives that address or satisfy them and then to the code that implements the alternative. The links are created manually when the rationale is captured.

SEURAT has been integrated with Eclipse so that when an alternative is associated with a code element, such as a class, attribute, or method, an indicator appears in the Eclipse Package Explorer to show that the code has associated rationale. An Eclipse Bookmark is also created to describe the association. The bookmark can be used in two ways: to provide direct access to the code using the Eclipse Bookmark Display to jump to the code and by using the bookmark indicator in the editor to display the alternative when the mouse is placed over it.

The rationale can also be associated with other types of elements by specifying an identifier in the relevant alternative's editor but that does not yet provide a direct link to UML or other documentation.

## 4. RELATED RESEARCH

This section will briefly describe selected research about rationale and also requirements traceability.

### 4.1 Rationale

Much of the work on capture, representation, and use of rationale, or more specifically design rationale, has been in the fields of engineering design [29] and Human Computer Interaction (HCI) [32]. There has also been significant work done using rationale for software development [18].

Potts and Bruns [35] created a model of generic elements in software design rationale that was then extended by Lee [30] in creating his Decision Representation Language (DRL), the language used as the basis of the representation used by SEURAT. DRIM (Design Recommendation and Intent Model)

was used in a system to augment design patterns with design rationale [34]. This system is used to select design patterns based on the designers' intent and other constraints. WinWin [2] aims at coordinating decision-making activities made by various "stakeholders" in the software development process. WinWin has also been used to assist in identifying quality requirements (often the same as NFRs) and conflicts between them [23]. To capture decision rationale in WinWin, Bose [5] defined an ontology for the decision rationale needed to maintain the decision structure. The goal was to model the decision rationale in order to support decision maintenance by allowing the system to determine the impact of a change and propagate modification effects. The Bose ontology captures win conditions, issues, options, and agreements as the primary elements in the rationale. This is a simpler representation than the SEURAT representation.

There are also systems that perform consistency checking. C-ReCS [27] performs consistency checking on requirements and recommends a resolution strategy for detected exceptions. Reiss [36] has developed a constraint-based, semi-automatic maintenance support system that works on the code, abstracted code, design artifacts, or meta-data to assist with maintaining consistency between artifacts.

## 4.2 Requirements Traceability

There are numerous techniques for requirements traceability. Excellent summaries can be found in [11] and [22]. Cleland-Huang [11] categorizes traceability techniques based on their support for trace creation and impact identification. Here we will describe the two techniques most strongly related to our work: Goal Centric Traceability [10][11] and Cysneiros and Leite's work associating NFRs with UML [15]. Both these projects build on work done by Chung, et al. [9] in developing their NFR framework. The NFR framework uses non-functional requirements to drive the design process, producing the design and its rationale.

Goal Centric Traceability [10] consists of four phases: goal modeling, impact detection, goal analysis, and decision making. The goal modeling phase uses Chung's Softgoal interdependency graph (SIG) [9] to capture the NFRs and tradeoffs. Impact detection automatically creates links between the SIG elements and a functional model of the system captured in UML class diagrams using ontological keywords. Goal analysis propagates changes made to the goal contributions by the user through the SIG to determine their impact. Decision making describes the process where the stakeholders assess the impact of the change and decide if they should proceed or not. GCT has also been expanded to trace NFRs to architectural assessment models (AAMs) [11]. Event Based Traceability (EBT) techniques [11] are used for both association and impact assessment. In EBT, software artifacts subscribe to requirements that they are related to using an event server. If a change occurs, the requirement sends an event to the dependent artifacts.

As described earlier, Cysneiros and Leite [15][16] investigated ways to build conceptual models that reflect NFRs. This involves looking at both the non-functional view, built on Chung's NFR framework, and a functional view consisting of UML. They use a Language Extended Lexicon (LEL) to provide a common vocabulary to link functional and non-functional views and provide traceability between them. The LEL is initially built to

capture the vocabulary of the "University of Discourse" that defines the software's domain [15]. Each entry, or symbol, contains notions, behavioral responses, and aliases. Notions provide the meaning of the symbol and its relationship to entities. Behavioral responses describe what the symbol does, or what can happen because it exists. The LEL initially contains functional information and NFRs are added later. Once NFRs have been added, they can be included in goal graphs developed using Chung's NFR framework [9]. The vocabulary defined by the LEL is used in use cases and scenarios to provide traceability to the functional models.

Cleland-Huang evaluates the support of existing traceability methods for NFR trace creation, impact identification, and impact scope identification in [11]. SEURAT provides manual support for trace creation: all associations must be entered manually in the rationale. Impact identification is automatic once the manual associations have been performed and impact scope evaluation is also automatic once the associations are formed.

Three traceability elements required to support NFR impact analysis are also defined [11]. These are impact detection, tracing between NFRs and FRs; traceability between lower level and higher level goals; and traceability to architectural assessment models. SEURAT provides some traceability between NFRs and FRs by allowing the capture of rationale for the FRs in terms of related NFRs. Traceability between lower level and higher level goals is provided indirectly by the hierarchy of NFRs in the argument ontology and by the ability to capture decision rationale at different levels (such as decisions resulting from the selection of a particular alternative). The Claims appearing in arguments, which are mapped to the Argument Ontology, bear some resemblance to the application-specific goals given in the SIGs used in GCT. SEURAT can provide manually entered traceability from requirements to development artifacts but does not perform any automated traceability.

Boehm and In [3] developed the Quality Attribute Risk and Conflict Consultant (QARCC) to identify conflicts between quality attributes. The set of quality attributes in their knowledge base is similar to the higher levels of the SEURAT Argument Ontology and some, although not all, of the product and process strategies captured in QARCC can be mapped to lower level elements in the Argument Ontology. They explicitly differentiate between product and process strategies while the SEURAT ontology does not. The SEURAT hierarchy uses a deeper hierarchical structure to capture quality attribute relationships. Quality attribute conflicts can also be used to detect conflicts between software requirements [19] by relating software requirements to quality requirements and identifying conflicts by looking at conflicts between quality attributes as well as overlap between artifacts involved in meeting potentially conflicting software requirements. SEURAT supports capturing conflicts between quality attributes by allowing capture of tradeoffs between elements in the Argument Ontology. SEURAT does not use this information to detect conflicts between software requirements but it does use this information to detect missing information from the rationale.

## 5. SUMMARY AND CONCLUSIONS

Software requirements, both functional and non-functional, play a key role in the software decision-making process. These decisions, alternatives considered, and the requirements used to select between alternatives can be captured in the rationale for the software system. This rationale also can provide traceability from requirements to the software (code and other artifacts) via the alternatives selected and eventually implemented.

A rationale-based approach is a manual process and does not solve all traceability problems. Traceability is only complete if the rationale is and traceability links must be manually maintained along with the program design and code. However, the traceability provided by SEURAT is a beneficial byproduct of the decision-making process rather than the primary goal of the tool. Also, SEURAT can be used to analyze the rationale to determine if there are requirements that are violated or that have not been used in the decision-making process. The failure to document the role of all the requirements in decision-making would indicate either incomplete rationale or possible failure in meeting the requirements.

There are some extensions that could be made to SEURAT to enhance its support for traceability. One would be to add the ability to capture dependencies between requirements as part of the requirement rationale. This could be implemented in a similar way as the dependencies between alternatives are captured now. This would allow SEURAT to detect problems introduced if a requirement is disabled that has other requirements dependent on it. These dependencies could also be used to describe conflicts between requirements. It would also be useful to export the requirement-alternative-artifact relationships into a traceability matrix. This matrix could then be augmented with the output of other, automated, traceability tools.

As mentioned earlier, there are some similarities between the Argument Ontology and the upper levels of Softgoal Interdependency Graphs (SIG) [9]. It would be interesting to export the Argument Ontology and the Claims that refer to it into the form of SIGs and compare that to the work done by researchers using SIGs [11] [15] for traceability. The SIGs could potentially be used to look for gaps in the rationale and capturing the rationale could be used as a way to elicit and refine Softgoals.

The rationale for a software system has many benefits. It can assist in software decision-making by providing a means for expressing and evaluating alternatives, it can support collaboration by allowing multiple viewpoints to be expressed and negotiated, and it can support maintenance by providing insight into the developer's intent behind earlier decisions. In addition, rationale can provide support for traceability of functional and non-functional requirements, to augment the power of other tools.

## 6. ACKNOWLEDGMENTS

Our thanks to the anonymous reviewers for their suggestions on improving this paper.

## 7. REFERENCES

- [1] Antón, A. I. and Potts, C. The use of goals to surface requirements for evolving systems. In *Proceedings of the 20th international Conference on Software Engineering*

- (Kyoto, Japan, April 19 - 25, 1998). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 157-166.
- [2] Boehm, B. and Bose, P. Collaborative spiral software process model based on Theory W, In *Proc. 3rd International Conf. on the Software Process*, IEEE Computer Society Press, CA, 1994, 59-68.
- [3] Boehm, B. and In, H. Identifying quality-requirement conflicts, *IEEE Software*, 13, 2 (March, 1996) 25-35
- [4] Boehm, B., Brown, J., Kaspar, H., Lipow, M., MacLeod G. and Merrit, M. Characteristics of Software Quality. *TRW Series of Software Technology*, Vol. 1. North-Holland, 1979
- [5] Bose, P. A. Model for decision maintenance in the WinWin collaboration framework, In: *Proc. of the Conf. on Knowledge-based Software Engineering* (Boston, Massachusetts, November 12-15, 1995), IEEE Computer Society Press, CA, 105-113.
- [6] Bruegge, D., and Dutoit, A., *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, 2000.
- [7] Burge, J.E. *Software Engineering Using design RATIONale*, Ph.D. Thesis, Worcester Polytechnic Institute, <http://www.wpi.edu/Pubs/ETD/Available/etd-050205-085625/>, May 2005.
- [8] Burge, J.E. and Brown, D.C. Rationale-based support for software maintenance. In *Rationale Management in Software Engineering*, Dutoit AH, McCall R, Mistrik I, Paech B (Eds.) Springer, Germany, 2006, 273-296
- [9] Chung, L., Nixon, B.A., Yu, E. and Mylopoulos, J. *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.
- [10] Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhan, E., and Christina, S. Goal Centric Traceability for managing non-functional requirements, In *Proc. of the International Conf. on Software Engineering* (St Louis, MO, May 15-21, 2005), 2005, pp. 362-371.
- [11] Cleland-Huang, J. Toward improved traceability of non-functional requirements. In *Proceedings of the 3rd international Workshop on Traceability in Emerging Forms of Software Engineering* (Long Beach, California, November 08 - 08, 2005). TEFSE '05. ACM Press, New York, NY, 14-19.
- [12] Cleland-Huang, J., Chang, C.K., and Wise, J. Automating performance related impact analysis through event based traceability, *Requirements Engineering Journal*, Springer-Verlag, 8, 3 (Aug. 2003) 171-182.
- [13] CMMI Product Team, CMMI For Development. Version 1.2. CMU/SEI-2006-TR-008, 2006
- [14] CMU: 2002, Quality measures taxonomy, [http://www.sei.cmu.edu/str/taxonomies/view\\_qm.html](http://www.sei.cmu.edu/str/taxonomies/view_qm.html)
- [15] Cysneiros, L. M. and Leite, J. C., Nonfunctional requirements: from elicitation to conceptual models, *IEEE Trans. Softw. Eng.*, 30, 5 (May, 2004) 328-350.
- [16] Cysneiros, L. M. and do Prado Leite, J. C. Using UML to reflect non-functional requirements. In *Proceedings of the*

- 2001 Conference of the Centre For Advanced Studies on Collaborative Research (Toronto, Ontario, Canada, November 05 - 07, 2001). D. A. Stewart and J. H. Johnson (Eds.), IBM Centre for Advanced Studies Conference. IBM Press, 202-216.
- [17] Dick, J. Design traceability, *IEEE Software*, November/December 2005, pp. 14-16.
- [18] Dutoit, A., McCall, R., Mistrik, I. and Paech, B. (Eds.) *Rationale Management in Software Engineering*, Springer-Verlang, Berlin, 2006
- [19] Egyed, A. and Grünbacher, P. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *IEEE Software*, 21, 6 (November, 2004), 50-58.
- [20] Filman, R.E. Achieving ilities, In *Proc. of the Workshop on Compositional Software Architectures* (Monterey, CA, USA, 1998)
- [21] Gotel O. and Finkelstein A. An analysis of the requirements traceability problem. In: *Proceedings of the 1st Int. Conf. on Requirements Engineering* (Colorado Springs, CO, April, 1994), IEEE Computer Society Press, 94-101.
- [22] Huffman Hayes, J. and Dekhtyar, A. A Framework for comparing requirements tracing experiments, *International Journal on Software Engineering and Knowledge Engineering (IJSEKE)*, 15, 5 (October, 2005) 751-781.
- [23] In, H., Boehm, B., Rodgers, T., and Deutsch, M. Applying WinWin to quality requirements: a case study. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada, May 12 - 19, 2001). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 555-564.
- [24] Jiro, K. *KJ Method: A Scientific Approach to Problem Solving*, Tokyo: Kawakita Research Institute, 2000.
- [25] Jung, H., Kim, S., and Chung, C. Measuring software product quality: A Survey of ISO/IEC 9126. *IEEE Softw.* 21, 5 (Sep. 2004), 88-92.
- [26] King, J.M.P., and Bañares-Alcantara, R. Extending the scope and use of design rationale records, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 11, 2 (April 1996) 155-167.
- [27] Klein, M. An exception handling approach to enhancing consistency, completeness and correctness in collaborative requirements capture, *Concurrent Engineering Research and Applications*, March 1997, pp. 37-46.
- [28] van Lamswerde, A. Goal-oriented requirements engineering: a guided tour, In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering* (Toronto, Canada, 2001) 249-263.
- [29] Lee, J. Design rationale systems: understanding the issues. *IEEE Expert: Intelligent Systems and Their Applications*, 12, 3 (May. 1997), 78-85.
- [30] Lee, J. Extending the Potts and Bruns model for recording design rationale. In *Proceedings of the 13th International Conference on Software Engineering* (Austin, Texas, United States, May 13 - 17, 1991). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 114-125.
- [31] Manola F. Providing Systematic Properties (Ilities) and Quality of Service in Component-Based Systems, Technical report, Object Services and Consulting, Inc, 1999
- [32] Moran, T. and Carroll, J. (Eds) *Design Rationale Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, Mahwah, NJ, 1995.
- [33] Nuseibeh B. and Easterbrook S. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (Limerick, Ireland, 2000) 35-46.
- [34] Peña-Mora, F. and Vadhavkar, S. Augmenting design patterns with design rationale, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11, 2 (April, 1996) 93-108.
- [35] Potts, C. and Bruns, G. Recording the reasons for design decisions. In *Proceedings of the 10th international Conference on Software Engineering* (Singapore, April 11 - 15, 1988). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 418-427.
- [36] Reiss, S.P. Constraining software evolution, *Proc. of the International Conf. on Software Maintenance* (Montreal, Quebec, Canada, 2002) 162-171.
- [37] Roman G. A taxonomy of current issues in requirements engineering, *Computer*, 1985, 14-22.
- [38] Thayer R.H. and Dorfman M. Introduction, issues, and terminology. In: *System and Software Requirements Engineering*, Thayer R.H. and Dorfman M. (Eds), IEEE Computer Society Press, Los Alamitos, CA, 1st edition, 1990, 1-3.
- [39] Vetschera R. Preference-based decision support in software engineering. In: *Value-Based Software Engineering*, Biffel, S., Aurum, A., Boehm, B., Erdogmus, H. and Grunbacher, P. (Eds.), Springer, 2006, 67-89.