Rationale Support for Maintenance of Large Scale Systems

Janet E. Burge and David C. Brown
Worcester Polytechnic Institute
Computer Science Department
Worcester, Massachusetts, 01609, USA
jburge@cs.wpi.edu, dcb@cs.wpi.edu

August 5, 2003

Abstract

Software maintenance has long been one of the most difficult and expensive phases of the software life-cycle. Maintenance is especially difficult for large-scale systems. The more code involved, the larger the chance that there may be unexpected interactions that may cause problems when updates and corrections are made during maintenance. The large number of developers who were probably involved at various points in the system's creation means that it is likely to be difficult to answer questions about the *intent* behind the design and implementation decisions. The designer's, or developer's, intent can be captured as their Design Rationale. Unlike standard design documentation, which is a description of the final design, Design Rationale (DR) offers more: not only the decisions, but also the reasons behind each decision, including its justification, other alternatives considered, and argumentation leading to the decision.

To drive and evaluate our research into using rationale for software maintenance, we are developing the SEURAT (Software Engineering Using RATionale) system to support the software maintainer. This system will present the relevant DR when required and allow entry of new rationale for the modifications. The new DR will then be verified against the existing DR to check for inconsistencies. There are several types of inferences that should be made: structural inferences to ensure that the rationale is complete, evaluation, to ensure that it is based on well-founded arguments, and comparison to rationale collected previously for similar modifications to see if the same reasoning was used.

1 Introduction

1.1 Problem and Motivation

Software maintenance has long been one of the most difficult and expensive phases of the software life-cycle. Maintenance costs can be more than 40 percent of the cost of developing the software in the first place [18]. One reason for this is that the software lifecycle is a long one. Large projects may take years to complete and spend even more time out in the field being used (and maintained). The panic over the "Y2K bug" highlighted the fact that software systems often live on much longer than the original developers intended. Compared to hardware, software is "easy" to modify during maintenance—software maintenance changes can often be more extensive and more frequent than maintenance performed on less mutable systems. The likelihood that these changes may add defects to the system is increased by the fact that the combination of a long life-cycle and the typically high personnel turnover in the software industry increases the probability that the original designer is unlikely to be available for consultation when problems arise.

Maintenance is especially difficult for large-scale systems. The more code involved, the larger the chance that there may be unexpected interactions that may cause problems when updates and corrections are made during maintenance. The large number of developers who were probably involved at various points in the system's creation means that it is even more likely to be difficult to answer questions about the intent behind the design and implementation decisions. This increases the probability that new decisions made during maintenance will conflict with this original intent with adverse consequences.

All these reasons argue for as much support as can be provided during maintenance. Semi-automatic systems, such as Reiss's constraint-based system [27], working on the code, abstracted code, design artifacts, or meta-data, can already provide a lot of support. Design Rationale, however, provides an additional opportunity for assistance during maintenance.

1.2 Approach

The designer's, or developer's, intent can be captured as their Design Rationale. Unlike standard design documentation, which is a description of the final design, Design Rationale (DR) offers more: not only the decisions, but also the reasons behind each decision, including its justification, other alternatives considered, and argumentation leading to the decision [22]. This additional information offers a richer view of both the product and the decision making process by providing the designer's intent behind the decision [28]. This DR would then be available for use by the software maintainer to determine where software changes should be performed and to determine the impact of these changes.

To drive and evaluate our research into using rationale for software maintenance, we are developing the SEURAT (Software Engineering Using RATionale) system to support the software maintainer. This system will present the relevant DR when required and allow entry of new rationale for the modifications. The new DR will then be verified against the existing DR to check for inconsistencies. There are several types of inferences that should be made: structural inferences to ensure that the rationale is complete, evaluation, to ensure that it is based on well-founded arguments, and comparison to rationale collected previously for similar modifications to see if the same reasoning was used. In the latter, the previous rationale could be used as a guide in determining the rationale for the new modification. The system will also propagate, or assist in propagating, any necessary changes to the existing DR as well as alerting the maintainer if the code modifications are the same as those made earlier and then rejected. We have developed the requirements for the SEURAT system based on earlier work on the InfoRat (Inferencing over Rationale) system [5] as well as a preliminary study on using rationale for software maintenance [6].

Our work focuses on the *use* of DR, in order to explore and evaluate its potential. It is *not* intended to be a fully-fledged programming and maintenance support system.

In this paper, we first describe related work in this area (Section 2), the representation for our software design rationale (Section 3), and the argument ontology developed to support semantic inferencing (Section 4). This is then followed by a discussion of the inferencing supported by the system (Section 5), and the proposed system (Section 6). We then conclude with a summary of our approach (Section 7).

2 Related Work

Design rationale research has typically focused on three aspects of rationale: DR representation, DR capture, and DR use. Design Rationale representations vary from informal representations such as audio or video tapes, or transcripts, to formal representations such as rules embedded in an expert system [11]. A compromise is to store information in a semi-formal representation that provides some computation power but is still understandable by the human providing or using the information.

Semi-formal representations are often used to represent argumentation. Argumentation notations provide a structure to indicate what decisions were made (or not made) and the reasons for and against them. Argumentation formats date back to Toulmin's representation [29] of datums, claims, warrants, backings and rebuttals. This is the origin of most argumentation representations. More recent argumentation formats include Questions, Options, and Criteria (QOC) [23], Issue Based Information System (IBIS) [11], and Decision Representation Language (DRL) [21]. Each argumentation format has its own set of terms but the basic goal is to represent the decisions made, the possible alternatives for each decision, and the arguments for and against each alternative.

Argumentation has been used in rationale representations that were created specifically for software design. Potts and Bruns [26] created a model of generic elements in software design rationale that was then extended by Lee [21] in creating his Decision Representation Language (DRL), the language used in SIBYL [20]. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale [25]. This system is used to select design patterns based on the designers intent and other constraints.

There are also many different ways to capture DR. One approach is to build the rationale capture into a system used for the design task. Active Design Documents (ADD), a system that does routine, parametric design [15], uses rationale already built into a knowledge base and associates it with the user's decisions. Some systems capture DR by integrating the system into an existing design tool. This is done by M-LAP (Machine-Learning Apprentice System) [3]. In M-LAP, user actions are recorded at a low level and formed into useful sequences using machine-learning techniques. This is also done in the RCF (Rationale Construction Framework) [24]. RCF uses its theory of design metaphors to interpret actions recorded in a CAD tool and convert them into a history of the design process.

Capturing rationale is often expensive and time consuming and can only be justified if there are compelling uses for the rationale. Systems such as JANUS (Fischer, et. al., 1995), critique the design and provide the designers with rationale to support the criticism. Others, such as SIBYL [20], verify the design by checking that the rationale behind the decisions is complete. C-Re-CS [19] performs consistency checking on requirements and recommends a resolution strategy for detected exceptions. InfoRat [5] performs inferencing over the rationale to verify that the rationale is complete and consistent, and to also evaluate that decisions made were well supported.

There has also been work on using design rationale in software design. DRIM (Design Recommendation and Intent Model) was used in a system to augment design patterns with design rationale [25]. Co-MoKit [12] uses a software process model to obtain design decisions and causal dependencies between them. WinWin [1] aims at coordinating decision-making activities made by various ”stakeholders” in the software development process. Bose [2] 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. Chung, etc. al. [8] developed an NFR Framework which uses non-functional requirements to drive the software design process, producing the design and its rationale.

3 Rationale Representation

We have generated an initial rationale representation, RATSpeak. We have chosen to represent our rationale in a semi-structured argumentation format because we feel that argumentation is the best means for expressing the advantages and disadvantages of the different design options considered.

We chose to base RATSpeak on DRL [21] because DRL is the most comprehensive of the rationale languages. Even so, it was necessary to make some change because DRL did not provide a sufficiently explicit representation of some types of argumentation (such as indicating if an argument was for or against an alternative).

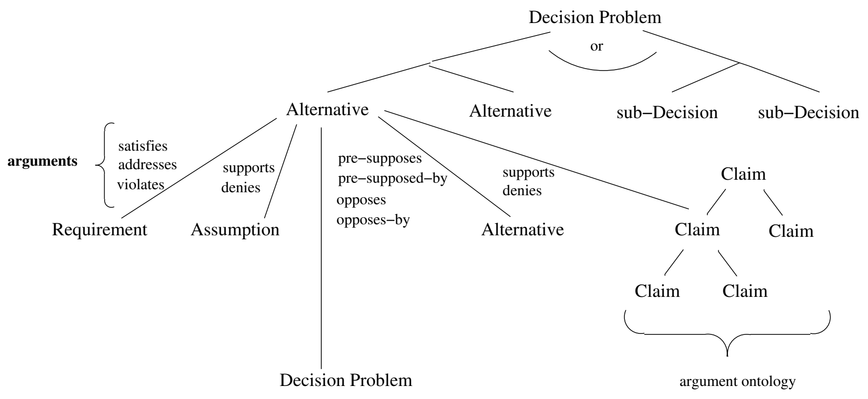

Figure 1: RATSpeak Argumentation Structure

RATSpeak uses the following elements as part of the rationale:

- *Requirements* - these are the requirements, both functional and non-functional. These can either be represented explicitly in the rationale or stored as pointers to requirements stored in a requirements document or database. For the purposes of our examples, we will show them as part of the rationale. Requirements serve two purposes in RATSpeak, one is as the basis of arguments for and against alternatives. This allows RATSpeak to capture cases where an alternative supports or violates a requirement. The other purpose is so that the rationale for the requirements themselves can be captured.
- *Decision Problems* - these are the decisions that must be made as part of the development process. They tend to be expressed in the form of questions.
- *Questions* - these are questions that need to be answered before the answer to the decision problem can be defined. These can be procedures or programs that need to be run or simple requests for information. While questions are not a standard argumentation concept, they can augment the argumentation by specifying the source of the information used to make the decisions, which would be very useful during software maintenance.
- *Alternatives* - these are alternative solutions to the decision problems. Each alternative will have a status that indicates if it is accepted, rejected, or pending.
- *Arguments* - these are the arguments for and against the proposed alternatives. They can either be requirements (i.e., an alternative is good or bad because of its relationship to a requirement), claims about the alternative, assumptions that are reasons for or against choosing an alternative, or relationships between alternatives (indicating dependencies or conflicts). Each argument is given an *amount* (how much the argument applies to the alternative, i.e., how flexible, how expensive) and an *importance* (how important the argument is to the overall system or the specific decision).
- *Claims* - these are reasons why an alternative is good or bad. Each claim maps to an entry in an *Argument Ontology* of common arguments for and against software design decisions. Each claim also indicates what direction it is in for that argument. For example, a claim may state that a choice is NOT safe or that an alternative IS flexible. This allows claims to be stated as either positive or negative assertions.

- *Assumptions* - these are similar to claims except that their truth is in doubt. Assumptions do not map to items in the Argument Ontology.
- *Argument Ontology* - this is a hierarchy of common argument types that serve as types of claims that can be used in the system. These are used to provide the common vocabulary required for semantic inferencing.
- *Background Knowledge* - this contains Tradeoffs and Co-Occurrence Relationships that give relationships between different arguments in the Argument Ontology. This is not considered part of the argumentation but is used to check the rationale for any violations of these relationships.

Figure 2 shows the relationships between the different rationale entities.

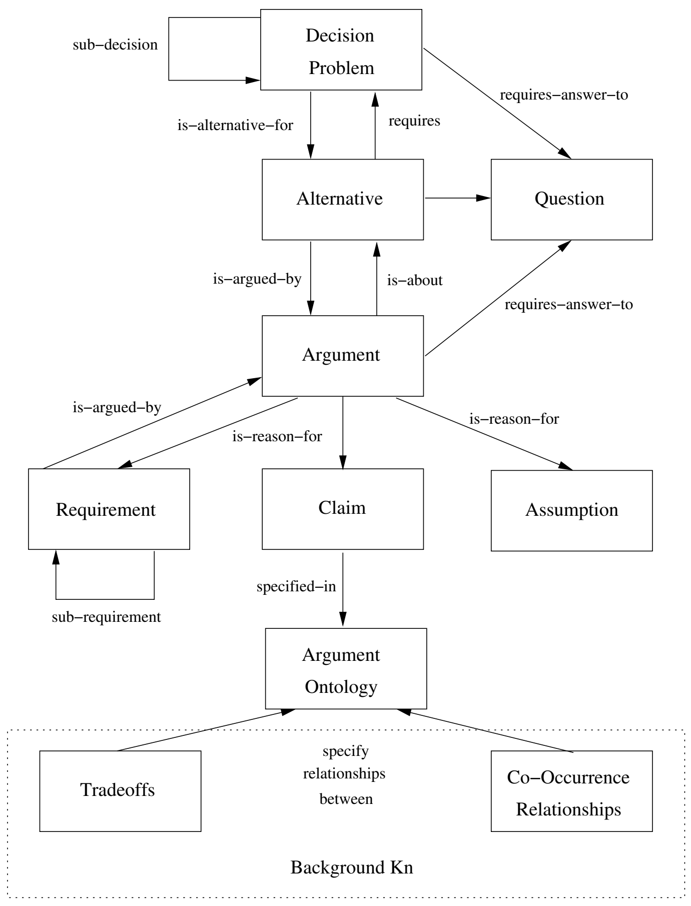

Figure 2: Relationship Between Rationale Entities

4 Argument Ontology

One key element in the RATSpeak representation is the Argument Ontology. Our work on InfoRat showed the importance of providing a common vocabulary to support inferencing over the content of the rationale as well as its structure. To support this, we have developed an ontology of reasons for choosing one design alternative over another. This ontology forms a hierarchy of terms with abstract reasons at the root and increasingly detailed reasons out towards the leaves.

RATSpeak provides the ability to express several different types of arguments for and against alternatives. One type of argument is whether an alternative satisfies or violates a requirement. Other arguments refer to assumptions made or dependencies between alternatives. A fourth type of argument involves claims that an alternative supports or denies a Non-Functional Requirement (NFR). These NFRs, also known as "ilities" [13] or quality requirements, refer to overall qualities of the resulting system, as opposed to functional requirements, which refer to specific functionality. As we describe in [7], the distinction between functional and non-functional is often a matter of context. RATSpeak also allows NFRs to be represented as explicit requirements.

There have been many ways that NFRs have been organized. CMU's Quality Measures Taxonomy [10] organizes quality measures into Needs Satisfaction Measures, Performance Measures, Maintenance Measures, Adaptive Measures, and Organizational Measures. Bruegge and Dutoit [4] break a set of design goals into five groups: performance, dependability, cost, maintenance, and end user criteria. Chung, et. al. [8] provide an un-ordered list of NFRs as well as specific criteria for performance and auditing.

For the RATSpeak argument 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 on the various quality categories to look for how a software system might choose to address these qualities. The 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. In maintenance, the maintainers are more likely to be looking at the lower-level decisions and will need specific reasons why these decisions contribute to a desired quality of the overall system. It is probable that decisions made at the implementation level are likely to correspond to detailed reasons in the ontology, while higher level decisions are more likely to use reasons at the more abstract levels.

After determining a list of detailed reasons for choosing one alternative over another, an Affinity Diagram [17] 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 (the CMU taxonomy, and Bruegge and Dutoit's design goals). Also, NFRs from the Chung list were used to fill in gaps in the ontology.

Figure 3 shows the first two levels of the Argument Ontology.

```
Affordability Criteria                Dependability Criteria
        Development Cost                      Security
        Deployment Cost                       Robustness
        Operating Cost                        Fault Tolerance
        Maintenance Cost                      Reliability
        Upgrade Cost                          Safety
        Administration Cost                   Availability

Adaptability Criteria                 End User Criteria
        Extensibility                         Usability
        Modifiability                         Integrity
        Adaptability
        Portability                   Needs Satisfaction Criteria
        Scalability                           Verifiability
        Reusability                           Traceability
        Interoperability
                                      Performance Criteria
Maintainability Criteria                      Response Time and Throughput
        Readability                           Memory Efficiency
        Supportability                        Resource Utilization
```

Figure 3: Top Levels of the Argument Ontology

Each of these criteria then have sub-criteria at increasingly more detailed levels. As an example, Figure 4 shows the sub-criteria for Usability. The ontology terms are worded to be part of an argument: i.e., "<*alternative*> is a good choice because it <*ontology entry*>" where <*ontology entry*> includes the appropriate verb (supports, provides, etc.) for the argument, e.g., "reduces development time". The SEURAT system will be designed so that this ontology will be easily extensible by the user to incorporate additional arguments that may be missing from the ontology. With use, the ontology will continue to be augmented and will become more complete over time. It is possible to add deeper levels to the hierarchy but that will make it more time consuming for the developer to find the appropriate item when adding rationale. Hence ontology depth is a tradeoff that must be made.

The ontology is also intended to be easily extended to incorporate domain-specific arguments that will apply to the system under development. The arguments in SEURAT, including those given here, are only a starting point.

```
Increases Physical Ease of Use       Increases Recoverability
   {provides | supports} effective use  supports undo of user actions
     of screen real-estate              corrects user errors
   minimizes keystrokes
   {provides | supports} increased   Increases Acceptability
     visual contrast                    increases aesthetic value
   is easy to read                      avoids offensiveness

Increases Cognitive Ease of Use      Provides User Customization
   provides reasonable default values   {provides | supports} customization
   provides user guidance               supports different levels of user
   {encourages | supports} direct          expertise
     manipulation
   minimizes memory load on the user Supports Internationalization
   provides feedback                    reduces cultural dependencies
   {conforms to | utilizes} user        supports internationalization
     experience
   increases visibility of function  Increases Accessibility
     to users                           supports visual accessibility
   uses predictable sequences of        supports auditory accessibility
     actions                            supports mobility accessibility
   increases intuitiveness              supports cognitive accessibility
   {provides | supports} an appropriate
     metaphor
```

Figure 4: Usability Arguments

Similar hierarchies have been developed for the remainder of the categories in Figure 3. One thing to note is that it is not a strict hierarchy—there are many cases where items contributing toward one quality also apply to another. One example of this is the strong relationship between scalability and performance—throughput and memory use, while primarily thought of as performance aspects, also impact the scalability of the system. In this case, and others that are similar, items will belong to more than one category.

The argument ontology also includes a user-modifiable default importance for each item. These are present so that SEURAT users can specify this information once if the importance value should hold for an entire system. The importance is used in weighing the different arguments during inferencing. The importance can be overridden for each claim or argument but is stored with the ontology to allow this information to be global if desired.

Other relationships that need to be captured are tradeoffs and co-occurrences. These are cases where two items in the ontology often either oppose each other in arguments or support each other in arguments.

RATSpeak captures these as background knowledge stored as part of the rationale. This background knowledge refers to the items in the argument ontology and stores the relationships between them.

5 Inferencing

Design Rationale is very useful even if it is only used in the traditional way as a form of documentation that provides extra insight into the designer's decision-making process. DR can provide even more useful information about the design and modifications made to the design if there is a way to perform inferences over it. Due to the nature of DR, the results may be in the form of warnings or questions (as opposed to conclusions) that help the maintainer act carefully and consistently. In the following sections we describe a number of different inferences that could be performed over rationale that was structured using the RATSpeak representation.

There are two types of inferences that can be performed. Syntactic inferences are those that are concerned mostly with the *structure* of the rationale. They look for information that is missing. Semantic inferences require looking into the *content* of the rationale. The SEURAT system will include the following syntactic inferences:

- Checking for selected alternatives with no supporting arguments;
- Checking for selected alternatives with more arguments against than for;
- Checking for decisions where no alternatives were selected;
- Checking for decisions where one alternative has more arguments than others (may indicate bias or missing information).

Many of these inferences have been implemented as CLIPS [9] rules. Figure 5 shows a set of rules that work together to check for selected alternatives with no supporting arguments.

Semantic inference was explored by the InfoRat [5] system, which used a common vocabulary of reasons so that the content of the arguments could be compared. SEURAT will support the following semantic inferences:

- Checking if the best supported alternative was not selected (based on the importance of the arguments given);
- Checking if contradictory arguments were used (the same criteria used for and against an alternative);
- Checking consistency of argument abstraction (were some alternatives argued with more or less detailed criteria);
- Checking abstraction levels;
- Checking for selected alternatives that violate requirements;
- Checking for requirements that were not satisfied or addressed;
- Checking for violations of tradeoffs and co-occurrences captured in the background knowledge;
- Reporting statistical information of frequency of specific arguments.

```
; favorable arguments
(defrule checkFavorable
   (alternative (name ?a))
   (argument (name ?r))
   (argues (argument ?r) (alternative ?a))
   ( or (argument (name ?r) (argtype supports))
        (or (argument (name ?r) (argtype satisfies))
            (argument (name ?r) (argtype addresses))))
=> (assert (favorable ?r ?a)))

; check for presupposed arguments
(defrule checkPresupposed
   (argument (name ?r))
   (alternative (name ?a))
   (argues (argument ?r) (alternative ?a))
   (argument (name ?r) (argtype presupposed)(alt ?a2))
   (alternative (name ?a2) (status selected))
=> (assert (presupposed ?a)))

; if *any* arguments are favorable we want this to return false
(defrule checkNotSupported
   (alternative (name ?a) (status selected))
   (not (presupposed ?a))
   (not (favorable ?r ?a))
=> (assert (notsupported ?a)))
```

Figure 5: Rules Checking for Unsupported Alternatives

```
;requirements traceability - violates
(defrule checkViolatedRequirements
   (requirement (name ?q))
   (decision (name ?d))
   (alternativeFor (alternative ?a) (decision ?d))
   (argues (argument ?r) (alternative ?a))
   (argument (name ?r) (req ?q) (argtype violates))
   (alternative (name ?a) (status selected))
=> (assert (violatesReq ?q ?d ?a)))
```

Figure 6: Rule Checking for Violated Requirements

Most of these inferences have been implemented as CLIPS rules. Figure 6 shows a rule that looks for requirements that were violated.

In addition to the inferencing, SEURAT will also support querying information about the rationale. This will let the maintainer see what portions of the design and/or implementation affect which requirements, functional or otherwise.

6 SEURAT: Software Engineering Using RATionale

We are in the process of building the SEURAT system in order to demonstrate how rationale can be used in software maintenance. SEURAT will use commercial and open source development tools to store the design and implementation. These tools will then be augmented with the ability to store, view, and inference over the rationale. Figure 7 shows a diagram of the proposed SEURAT system.

This diagram shows that the main user interaction will be through two off-the-shelf tools: an Interactive Development Environment (IDE) that is used for writing and building source code and a UML design tool used in capturing the software design. SEURAT will be initially aimed at Java development and will use

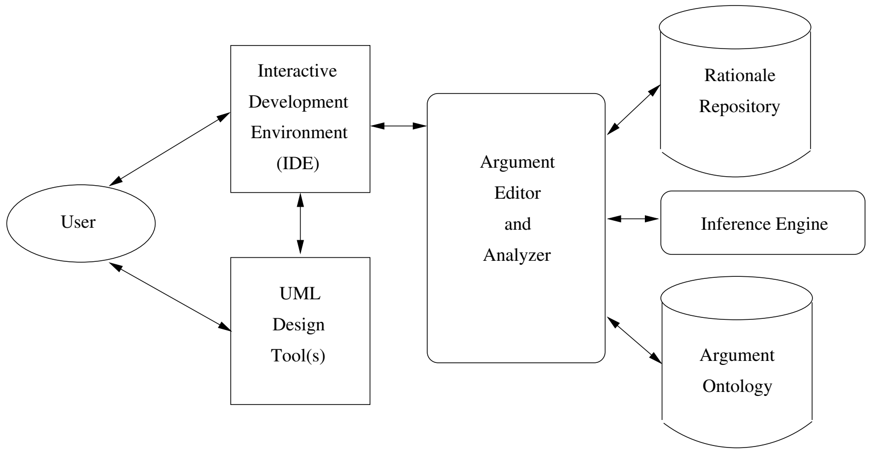

Figure 7: Proposed SEURAT System

the Eclipse IDE [16]. Eclipse is Open Source and can be extended to integrate with a context sensitive Argument Editor and Analyzer. There are a number of UML design tools that interface with Eclipse. One of those will be selected so that it is possible to enter and display rationale as notations to UML diagrams.

The inferencing will take place upon user command and will be performed by the Inference Engine. This will be developed using the Java Expert System Shell (JESS) [14], which is a CLIPS-based expert system shell. The CLIPS inference rules already developed will be used here, as well as any additional ones needed.

The rationale will be stored in a Rationale Repository. This will be done using a relational database. The Argument Ontology will be stored in a similar database. SEURAT will start with a general ontology that can be extended for the needs of a particular software system.

The SEURAT system will support a variety of uses for the rationale. For example, assume a software maintainer is working on a deployed system that was initially developed to support a relatively small number of users. This information appears in the rationale in two ways: an assumption that there would be only a few users at a time and the importance given to the scalability criteria in the argument ontology. SEURAT would be used to decrease the plausibility of the assumption and increase the importance of scalability as an argument. SEURAT would then re-evaluate the decisions made and inform the user if this results in selected alternatives that are no longer the best-supported. In addition, the maintainer could use SEURAT to look for where the scalability argument and assumption appear in the rationale as clues to where the software design and implementation need to be changed to allow additional users.

7 Summary and Conclusions

Several steps have been taken toward the development of SEURAT, the Software Engineering Using RATionale system. These include the development of a rationale representation, an argument ontology, and a preliminary set of inferences that can be performed over the rationale in order to support software maintenance. The SEURAT system will integrate the capture and use of rationale with development tools that could be used by the software maintainer to make modifications to the software.

Such a tool would be invaluable during the maintenance of large-scale software systems and will complement other programming and maintenance environments. One of the chief difficulties in maintaining a large system is knowing the reasons behind the choices made by the developers during design and implementation. The lack of this knowledge makes it difficult for the maintainer to do their job and increases the risk that defects are introduced during maintenance. The presence of rationale would serve as a form of “corporate memory” by capturing design information that would be lost if the developers left the company or if they were inaccessible to the maintainers [25]. Further support will be provided by giving the maintainer the ability to inference over the rationale to both view the reasons for the choices, evaluate the choices made, and determine the potential impact of new decisions.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful feedback on the paper and Mikhail Mikhailov, Tom Mens, and Juan Ramil for Latex assistance.

References

[1] B. Boehm and P. Bose. A Collaborative Spiral Software Process Model Based on Theory W. In *Proc. 3rd International Conf. on the Software Process*, pages 59–68, Reston, VA, 1994.

[2] P. Bose. A Model for Decision Maintenance in the WinWin Collaboration Framework. In *Proc. of the Conf. on Knowledge-based Software Engineering*, pages 105–113, Boston, MA, 1995.

[3] M. Brandish, M. Hague, and A. Taleb-Bendiab. M-LAP: A Machine Learning Apprentice Agent for Computer Supported Design. In *Artificial Intelligence in Design Workshop Notes on Machine Learning in Design*, Stanford, CA, 1996.

[4] D. Bruegge and A. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice Hall, 2000.

[5] J.E. Burge and D.C. Brown. Inferencing Over Design Rationale. In J. Gero, editor, *Artificial Intelligence in Design ’00*, pages 611–629. Kluwer Academic Publishers, 2000.

[6] J.E. Burge and D.C. Brown. Discovering a Research Agenda for Using Design Rationale in Software Maintenance. Technical Report WPI-CS-TR-02-03, WPI, 2002.

[7] J.E. Burge and D.C. Brown. NFRs: Fact or Fiction? Technical Report WPI-CS-TR-02-01, WPI, 2002.

[8] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[9] CLIPS. *CLIPS Reference Manual Volume I: Basic Programming Guide, Version 6.10*. 1998. http://www.ghgcorp.com/clips/download/documentation.

[10] CMU. Quality measures taxonomy. Technical report, CMU, 2002. http://www.sei.cmu.edu/str/taxonomies/view_qm.html.

[11] J. Conklin and K. Burgess-Yakemovic. A Process-oriented Approach to Design Rationale. In T. Moran and J. Carroll, editors, *Design Rationale Concepts, Techniques, and Use*, pages 293–328. Lawrence Erlbaum Associates, 1995.

[12] B. Dellen, K. Kohler, and F. Maurer. Integrating Software Process Models and Design Rationales. In *Proc. of the Conf. on Knowledge-based Software Engineering*, pages 84–93, Syracuse, NY, 1996.

[13] R.E. Filman. Achieving Ilities. In *Proc. of the Workshop on Compositional Software Architectures*, Monterey, CA, USA, 1998.

[14] E.J. Friedman-Hill. *Jess, The Java Expert System Shell.* Sandia National Laboratories, Livermore, CA, 1998.

[15] A. Garcia, H. Howard, and M. Stefik. Active Design Documents: A New Approach for Supporting Documentation in Preliminary Routine Design. Technical Report 82, Stanford University Center for Integrated Facility Engineering, 1993.

[16] IBM. *Eclipse Technical Platfom Overview*. 2003. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[17] K. Jiro. *KJ Method: A Scientific Approach to Problem Solving*. Tokyo: Kawakita Research Institute, 2000.

[18] F. P. Brooks Jr. *The Mythical Man-Month.* Addison Wesley, 1995.

[19] M. Klein. An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture. *Concurrent Engineering Research and Applications*, pages 37–46, 1997.

[20] J. Lee. SIBYL: A Qualitative Design Management System. In P. Winston and S. Shellard, editors, *Artificial Intelligence at MIT: Expanding Frontiers*, pages 104–133. MIT Press, 1990.

[21] J. Lee. Extending the Potts and Bruns Model for Recording Design Rationale. In *Proc. of the 13th International Conf. on Software Engineering*, pages 114–125, Austin, TX, 1991.

[22] J. Lee. Design Rationale Systems: Understanding the Issues. *IEEE Expert*, 12(3):78–85, 1997.

[23] A. MacLean, R.M. Young, V. Bellotti, and T.P. Moran. Questions, Options and Criteria: Elements of Design Space Analysis. In T. Moran and J. Carroll, editors, *Design Rationale Concepts, Techniques, and Use*, pages 201–251. Lawrence Erlbaum Associates, 1995.

[24] K. Myers, N. Zumel, and P. Garcia. Automated Capture of Rationale for the Detailed Design Process. In *Proc. of the 11th National Conf. on Innovative Applications of Artificial Intelligence*, pages 876–883, Menlo Park, CA, 1999.

[25] F. Pena-Mora and S. Vadhavkar. Augmenting Design Patterns with Design Rationale. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, pages 93–108, 1996.

[26] C. Potts and G. Bruns. Recording the Reasons for Design Decisions. In *Proc. of the International Conf. on Software Engineering*, pages 418–427, Singapore, 1988.

[27] S.P. Reiss. Constraining Software Evolution. In *Proc. of the International Conference on Software Maintenance*, pages 162–171, Montreal, Quebec, Canada, 2002.

[28] S. Sim and A. Duffy. A New Perspective to Design Intent and Design Rationale. In *Artificial Intelligence in Design Workshop Notes for Representing and Using Design Rationale*, pages 4–12, Lausanne, Switzerland, 1994.

[29] S. Toumlin. *The Uses of Argument.* Cambridge University Press, 1958.