International Journal of Software Engineering
and Knowledge Engineering
Vol. 22, No. 7 (2012) 945–964

DOI: 10.1142/S021819401250026X

FEATURE-BASED RATIONALE MANAGEMENT SYSTEM FOR SUPPORTING SOFTWARE ARCHITECTURE ADAPTATION*

BEDIR TEKINERDOGAN
Department of Computer Engineering
Bilkent University 06800 Bilkent, Ankara, Turkey
bedir@cs.bilkent.edu.tr

HASAN SOZER
Department of Computer Science
Özyeğin University, İstanbul, Turkey
hasan.sozer@ozyegin.edu.tr

MEHMET AKSIT
Department of Computer Science
University of Twente
Enschede, P. O. Box 217 7500 AE, The Netherlands
m.aksit@ewi.utwente.nl

Received 15 March 2010
Revised 20 January 2012
Accepted 1 March 2012

Each software architecture design is the result of a broad set of design decisions and their justifications, that is, the design rationale. Capturing the design rationale is important for a variety of reasons such as enhancing communication, reuse and maintenance. Unfortunately, it appears that there is still a lack of appropriate methods and tools for effectively capturing and managing the architecture design rationale. In this paper we present a feature-based rationale management approach and the corresponding tool environment ArchiRationale for supporting software architecture adaptation. The approach takes as input an existing architecture and captures the design rationale for adapting the architecture for a given quality concern. For this we define a feature model that includes the possible set of architectural tactics to realize the quality concern. The presented approach captures the rationale for deciding on feature selections and for selecting the corresponding architecture design alternatives. ArchiRationale customizes and integrates the Eclipse plugin tools XFeature, ArchStudio and XQuery to provide

*This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

tool support for capturing, storing and accessing the design rationale. We illustrate the approach for adapting a software architecture for fault tolerance.

Keywords: Architecture design rationale; architecture adaptation; tools.

1. Introduction

Research on architecture design in the last two decades has resulted in various useful approaches [1−3]. With the help of these approaches, the software architect makes a wide range of design decisions that leads to the selection of a particular design alternative. The reasons behind the design decisions, the justification, the alternatives considered, the trade-offs evaluated, and the argumentation that lead to the decision is defined as design rationale [4−7]. The explicit capturing, documentation and usage of the design rationale is important for many different reasons such as design communication, design evolution, design maintenance, design verification, and design reuse [8, 9]. Recent publications have discussed the concept of design rationale for software architecture and have provided better insight in the related concepts [7, 9−14]. In general these studies show that practitioners recognize the importance of documenting and usage of design rationale to support the reasoning about design choices. Yet, architecture design rationale is still not being documented in a consistent manner, if documented at all, due to "barriers to the use and documentation of design rationale" and as such "further research is needed to develop methodology and tool support for design rationale capture and usage" [9].

One of the important phases in capturing the design rationale is during the maintenance of the architecture whereby the architecture needs to be adapted to include new requirements. For example, an existing architecture might need to be enhanced for meeting some quality concerns such as persistence, security, reliability, etc. In such cases it is important to capture the design decisions together with the rationale behind these design decisions.

In this paper we provide a systematic approach to capture the design rationale for software architecture adaptation. The approach first defines a feature model [17] that includes the possible set of architectural tactics [2]. Hereby, architectural tactics are defined as a characterization of architectural decisions that are used to achieve a desired quality attribute response [18]. Based on the feature model of architectural tactics, the presented approach captures the rationale for deciding on feature selections, for adapting the architecture and for selecting the corresponding architecture design alternatives.

The approach that we present is implemented in the corresponding tool environment ArchiRationale, customizes and integrates the Eclipse plugin tools XFeature, ArchStudio and XQuery. ArchiRationale is a design rationale management system dedicated for supporting the rationale capture and rationale access for adapting software architecture for quality concerns. ArchiRationale expects as input a software architecture that has been designed using a given software architecture

design method. As such ArchiRationale is agnostic and not invasive to the adopted architecture design methods.

The reminder of this paper is organized as follows. In Sec. 2, we provide a case study that is used to illustrate the problem and importance of capturing and accessing architecture design rationale for adapting the architecture. Section 3 elaborates on rationale management systems and defines the context for the approach that we explain in this paper. In Sec. 4, we provide the metamodel for architecture design rationale on which the approach and the tools is based. Section 5 defines the architecture rationale process for adapting software architecture for quality. Section 6 describes our tool environment ArchiRationale using the example case. Section 7 characterizes our approach with respect to existing rationale management approaches. Section 8 provides the related work and finally Sec. 9 concludes the paper.

2. Motivation and Context

In this section we will discuss the motivation and the context of the rationale management system that we present in subsequent sections.

2.1. *Case study — initial architecture sub-headings*

In the following we present the case study MPlayer [19] which will be used throughout the paper to describe the problems and illustrate our approach. MPlayer embodies approximately 700 K lines of code and it is available under the GNU General Public License. The architecture consists of modules for reading input media demultiplexing the input to audio and video channels, maintaining the synchronization of audio and video, and displaying video frames, controlling playing of audio and presenting the graphical UI.

We assume that the architecture is the result of any architecture design method and do not consider the rationale behind this architecture. Instead we assume that the architecture needs to be adapted and it is the design rationale for architecture adaptation that we would like to capture. We have determined the basic requirements (cost-effectiveness, performance, availability) and adapted the architecture accordingly. As an example, in the following we will shortly discuss the adaptation of the architecture for fault tolerance.

2.2. *Adapting architecture for fault tolerance*

Fault-tolerance is a complementary technique to fault avoidance and fault removal for ensuring the systems reliability even if faults remain and they get activated [20]. When faults manifest themselves during system operations, fault tolerance techniques provide the necessary mechanisms to detect and recover from errors, if possible, before they propagate and cause a system failure. Error recovery is generally defined

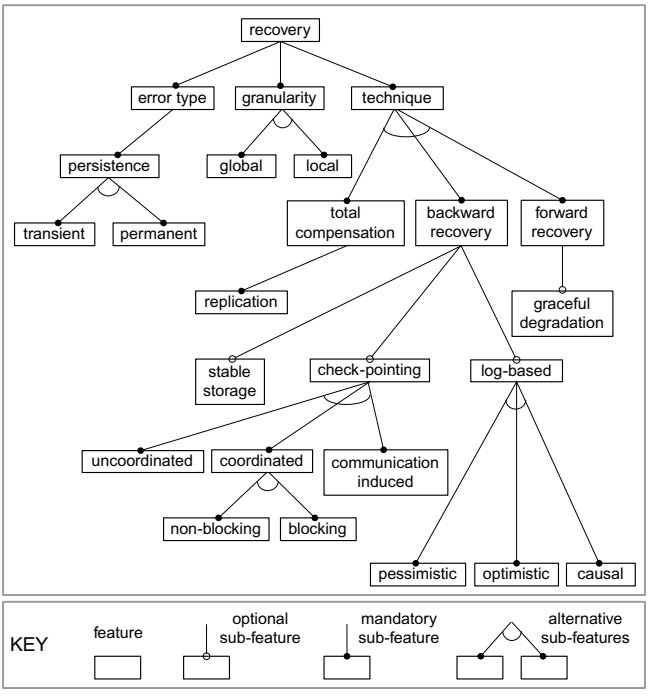
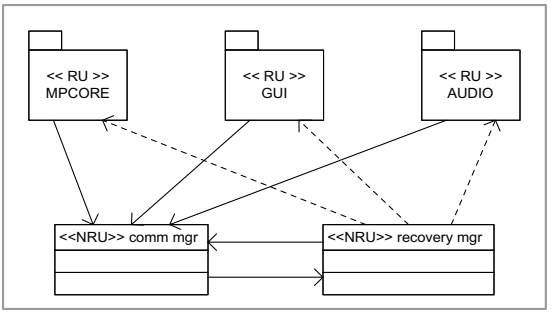


Fig. 1. A partial view of the architectural tactics space for recovery in fault-tolerance.

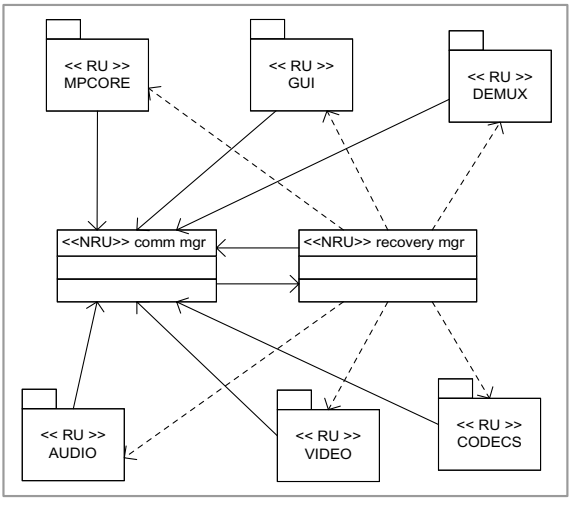
as the action, with which the system is set to a correct state from an erroneous state [20]. The domain of recovery is quite broad due to the different type of errors and the different requirements imposed by different type of systems (e.g., safety-critical systems, consumer electronics). Figure 1 shows a partial view of the feature diagram of recovery, which organizes the set of architectural tactics for fault tolerance. In fact the feature diagram defines the architectural tactics space, that is the possible set of architectural tactics for the given quality domain. Features are derived using a domain analysis process [20, 21].

As shown in Fig. 2, recovery can be characterized through three main features. The first feature is the error type that is recovered. This involves the characteristics of the errors like their persistence (persistent versus transient) and source (internal versus external) of the error [20]. The second important feature is the granularity of the recovery mechanism, which can be global or local. In the case of global recovery, the recovery mechanism can take actions on the system as a whole (e.g., restart the whole system). In the case of local recovery, erroneous parts can be isolated and recovered while the rest of the system is available. The third feature is the applied recovery technique, which is organized into three categories; *total compensation*, *forward recovery* and *backward recovery* [20]. We will not elaborate further on the details of recovery here. For the complete domain analysis we refer to [22]. Rather in

(a)



(b)



(c)

Fig. 2. Recovery design alternatives.

the following sections we will indicate the impact of the selection of particular features on the architecture design.

2.3. *Design alternatives*

We can now enhance the Mplayer architecture for particular recovery features, which are selected from the feature diagram in Fig. 1. Obviously, many different

feature decisions can be made and each of them will possibly lead to a different architecture design alternative. Architecture design alternatives may differ with respect to the granularity for recovery, the error detection protocols, the criticality of components etc.

Assume that it is required to choose to enhance the architecture to recover the complete system from transient failures using restart mechanisms. In this case, we can observe that different design alternatives are possible. Figure 2(a) presents, for example, one design alternative for this case. Here the *MPlayer* architecture (on the left) is enhanced with a central component *recovery mgr* (on the right) that restarts the complete system in case of failures. The stereotypes <<RU>> and <<NRU>> represent recoverable unit and non-recoverable unit, respectively. This alternative is less preferable in case the availability of the system is an important concern.

Instead of global recovery we might decide to apply local recovery. This will have also an impact on the architecture design alternative. Figure 2(b) is one example in this case, in which the system has been split up into three so-called recoverable units *MPCORE, GUI* and *AUDIO*. Hereby, each recoverable unit is a nonempty, disjoint subset of the set of system modules. The names of recoverable units, which are assigned based on the comprised module(s), are not significant. Compared to the design alternative of Fig. 2(a), this alternative is better with respect to availability since it is expected that failures in of the three modules will usually not lead to failures in other modules. As such the system can continue working, if needed at a reduced level. With respect to time performance the alternative of Fig. 2(b), however, is less favorable to the alternative in Fig. 2(a) because it includes more recoverable units and as such the recovery will require more time. Obviously, in this case study, performance and availability are the conflicting and most relevant quality attributes. Other quality attributes can be more relevant for different types of adaptation of the architecture. In Fig. 2(c) yet another alternative is given in which each module is considered as a separate recoverable unit. This alternative will be typically preferred if availability is prioritized over time performance. Of course we can continue searching for all design alternatives. In fact, each fault-tolerant architecture will be the result of a set of design decisions and there is a rationale behind these decisions (e.g., to reduce cost, to achieve high availability/performance).

From the examples above we can identify two different layers of decisions. First we need to decide on the features for recovery that need to be realized by the enhanced architecture. For example in Fig. 2(a) we have decided for global recovery and Fig. 2(b) on local recovery. Second, for a given set of feature decisions we can have different alternative architecture designs. For example, both the architecture alternatives in Figs. 2(b) and 2(c) are defined for local recovery. However, they decompose the system differently. So, a feature selection is a strategic design decision and it is followed by further decisions related to the resulting design alternatives.

It is important to make the architecture design rationale explicit because design decisions made for enhancing a quality concern can influence the design of the architecture significantly. The captured rationale can be used for similar purposes

like design communication, design evolution, design maintenance, design verification, and design reuse.

3. Rationale Management Systems

Design rationale management approaches can be categorized based on various criteria. An important classification is the distinction between *process-oriented* versus *feature-oriented approaches* [23, 4].

Process-oriented approaches are often applied for dynamic design domains in which the design principles are not well-established. Hereby, the approach considers the design rationale usually as a *history of the design process* [4, 24]. The representation of design rationale in this approach is usually graph-based in which the nodes represent questions, positions and arguments and the links the relations among these concepts. In the literature early approaches such as Ibis [25], QOC [26] , DRL [5] and PHI [27] can be categorized as process-based [4].

Feature-based rationale approaches have actually evolved from the process-based rationale approaches but differ in the rationale capturing approach. In a feature-based rationale approach the design rationale is based on features of a system rather than the arguments raised during the development process. A feature is defined as a characteristic of a domain that is relevant to its stakeholders [17]. Feature-based rationale approaches are typically used for well-defined domains with established design rules. Several feature-based rationale approaches can be identified in the literature including CRACK and GTMD [4].

To apply feature-based approaches, there should be certain features known in a mature-domain. In process-based approaches, there is no well-defined set of features or options. Different new questions, arguments can be raised during the development process. The approach that we described in this paper can be classified as a *feature-based approach*. We assume that the architecture is given and needs to be adapted for a particular quality concern. Using the case, for example, we assume that the MPlayer architecture is given and that this needs to be enhanced for fault tolerance. In general, in feature-based approaches a feature diagram is defined for the system and the decisions for features, that is, system properties, are recorded. In our approach a feature diagram is defined for architectural tactics that are used to implement quality concerns. In our case study, we focus on adapting for fault tolerance, and thus the rationale management process will be based on a feature diagram that organizes architectural tactics for fault tolerance. We will explain the details of the approach in the following sections.

4. Design Rationale Meta-Model

In Fig. 3 we provide the meta-model that defines the key concepts for capturing design rationale. The approach and the corresponding tool that we will explain in subsequent sections are based on this meta-model. The meta-model consists of four

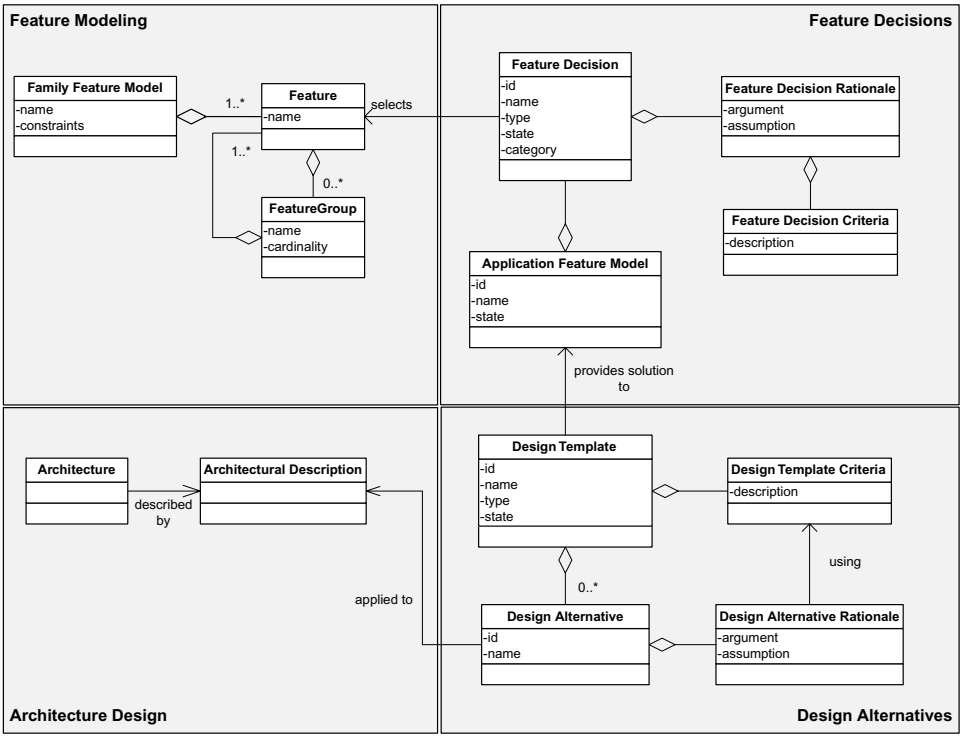


Fig. 3. Metamodel for design rationale.

sub-parts including: *Feature Modeling*, *Feature Decisions*, *Design Alternatives*, and *Architecture Design.*

The part *Feature Modeling* defines the metamodel for defining feature models. In our case we apply feature modeling for depicting the possible techniques for supporting a particular quality (i.e., architectural tactics). In the example case, a feature model is derived from a domain analysis to fault tolerance and fault-tolerant design, and as such should define the common and variant properties for techniques that are necessary for designing fault tolerant systems. A *Family Feature Model* represents the domain of architectural tactics for a given quality concern. A *Feature* represents an architectural tactic (a technique or design decision for adapting the architecture with respect to a quality concern).

The *Feature Decisions* part includes the concepts for selecting features and capturing the rationale behind these features. An *Application Feature Model* represents the features that are selected for adapting the architecture. *Application Feature Model* consists of multiple *Feature Decision* which are decisions that is made regarding the selection or deselection of a feature. A *Feature Decision Rationale* defines the motivation behind the feature decision. It will for example define why a given feature has been selected. *Feature Decision Rationale* has the attributes

argument and *assumption*. *Feature Decision Criteria* defines the criteria that are utilized in the argument for the feature decision rationale. Typically, feature decision criteria will include quality attributes or other criteria which is important to the stakeholders of the architecture. For fault tolerant design we will, for example, adopt the criteria *performance* and *availability* in selecting design decisions. In principle, a Feature Decision Criteria can be used by multiple Feature Decision Rationale objects. However, this is currently not supported by our tool, in which a set of criteria can be repeated for different design rationale.

The *Design Alternatives* part of the metamodel represents the approaches for realizing the selected feature decisions in the *Feature Decision Rationale*. A given *Application Feature Model* can be realized in many different ways. For this *Design Template* represents a generic template, which comprises architectural patterns, architectural tactics or general design heuristics. The design template for an application feature model is defined manually based on the included features. *Template Criteria* defines the criteria for selecting a particular design template. These criteria are separate from the criteria for selecting features and focus on the realization of the features. Similar to the general idea of patterns and tactics, a design template does not define a particular design yet, but must be instantiated for a given context. Each such instantiation is called a *Design Alternative.* Typically a design alternative defines the application of the comprised patterns, tactics or general design rules. In general, a design template can have zero or more design alternatives. For a template to be applied, the possible alternatives must be selected. *Design Alternative Rationale* defines the motivation for selecting a particular design alternative. The rationale is defined in the attributes *argument* and *assumption.*

The *Architecture Design* part of the metamodel includes the concept for modeling architectures.

5. The Approach

In this section we propose the approach for rationale management that is built on the design rationale management metamodel as defined in the previous section. As illustrated in Fig. 4 the approach is represented using a workflow diagram in which we distinguish between processes and artifacts. The relations represent data flow relations. We can distinguish the following steps:

(1) *Architecture Design*
The approach starts with an architecture design process that results in an architecture design. In our case study, we have used module and component-and-connector views of the architecture to define design templates and the existing architecture [1]. Other architectural views might also be applied when needed.

(2) *Modeling Architectural Tactic Space of Quality*
To depict the possible adaptations a family feature model is defined, which organizes the possible set of architectural tactics for enhancing the quality concern. The

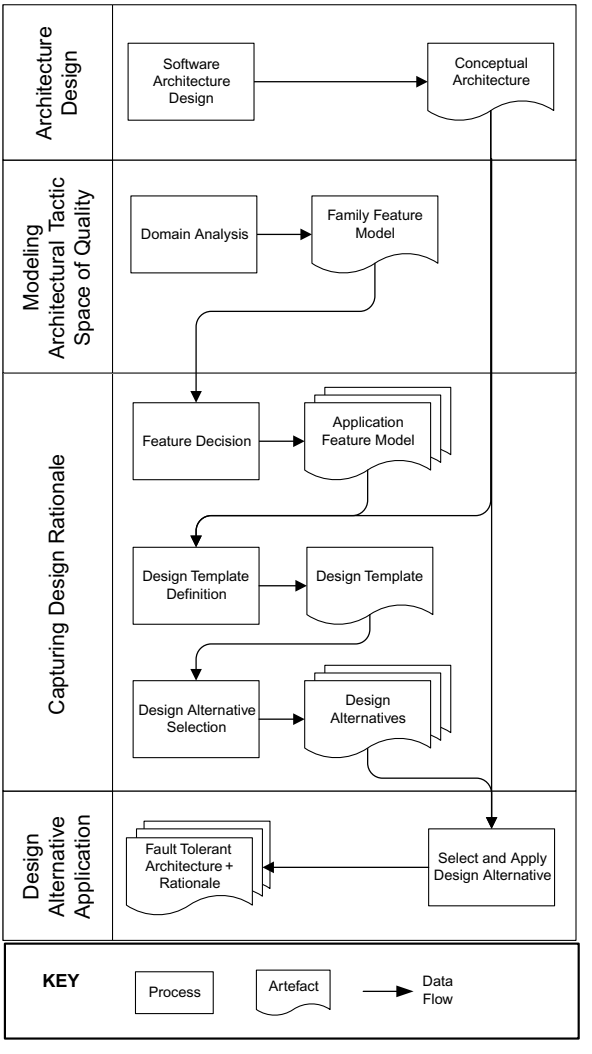


Fig. 4. Process for defining rationale for domain architecture.

features, i.e., the architectural tactics, are derived from the corresponding quality domain through a domain analysis process [17]. The feature model defines as such the so-called *architectural tactics space* for a given quality concern.

(3) *Capturing Design Rationale*

The rationale capturing process has three steps: (i) selecting the features in the family feature model resulting in the *application feature model* (ii) defining a design template that provides a solution to the selected features (iii) selecting design alternatives for applying the defined design template. In principle, we can thus capture design rationale for each of these three steps. However, we have focused on

the design decisions that are made in the first and third steps. As depicted in the workflow diagram (Fig. 4), these are the steps, which involve a selection from a set of (feature/design) alternatives. That is why the meta-model (Fig. 3) reflects two levels for the specification of design rationale.

(4) *Design Alternative Application*
Once the design alternatives have been depicted, the design alternatives are analyzed, selected and applied to the architecture. The result of the design alternative application is a fault-tolerant architecture together with the design rationale for adaptation. The stored design rationale can be used for communication, design evolution, design maintenance, design verification, and design reuse.

We assume that the artifacts delivered through the rationale process are stored in a repository and the necessary information can be queried. Hereby, we can think of the following related questions:

- What were the possible techniques (features) for adapting the architecture?
- Which techniques (features) have been chosen for adapting the architecture, why?
- What were the possible designs for the selected features? Which have been chosen, why?

6. ArchiRationale Tool

In this section we present *ArchiRationale* that provides a set of integrated tools to implement our rationale management approach. In the following we will describe how the tool is applied in supporting the design rationale management process.

6.1. *Overall tool architecture*

The *ArchiRationale* tool is built in the *Eclipse Platform*. To implement the rationale management approach, it customizes and integrates several open-source tools that are provided as Eclipse plug-ins and all based on the XML technology. *ArchStudio* is adopted to describe the architecture. *XFeature* is adopted and customized to express the feature models. *XQuery* is used to effectively retrieve, utilize and reuse information related to design rationale. Finally we have developed an architecture analysis tool, so-called *Recovery Designer*. The common basis of these tools is that they are all XML-based and provided as plug-in tools for the Eclipse platform. As a result, the seamless integration comes for free once they are customized for our purpose, design rationale management. A snapshot of the user interface of the tool is shown in Fig. 5. This snapshot is presented just to illustrate the general layout of the tool. The contents of the panes in the snapshot are not meant to be readable.

As it can be seen in Fig. 5, a typical Eclipse plug-in tool provides a user interface with 4 different panes; (1) Editing Pane (top-right), (2) Outline View Pane (top-left), (3) Properties Pane (bottom-right), and (4) Navigator Pane (bottom-left). In

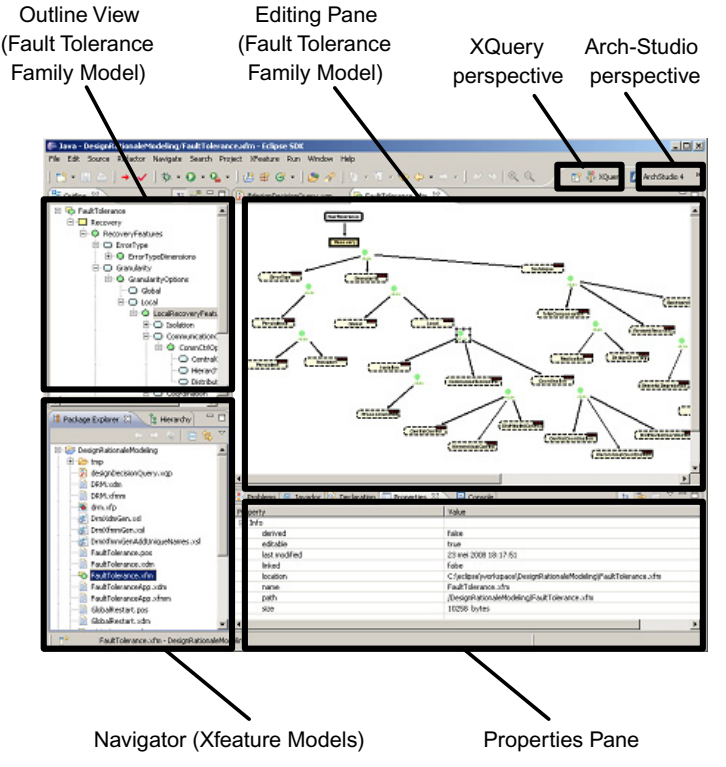


Fig. 5. ArchiRationale tool user interface.

general, the Editing Pane is used for viewing and editing the object of interest (e.g., design document, source file, or diagram). The Outline View Pane presents an outline of the object that is being edited and the Properties Pane lists the properties related to this object. The Navigator Pane is used for browsing the list of objects and resources that are part of a project. The exact set of panes, their purpose and layout (known as *perspectives* in the Eclipse platform) adapt based on the activated tools.

The perspective (i.e., the user interface) can be changed automatically depending on the type of object being edited or it can be activated from the toolbar at the top-right of the Eclipse platform. This part of the user interface is marked in Fig. 5, where we can switch to the *XQuery* or *ArchStudio* perspectives. In the following we will explain the steps of the approach and the related tools.

6.2. *Modeling architecture*

To model the architecture, the ArchStudio perspective of the Eclipse platform is activated. We model the existing architecture design and design alternatives with the *Archipelago* tool of ArchStudio.

6.3. *Modeling feature diagram*

The XFeature tool enables the specification of feature diagrams based on any meta-model that confirms to the XFeature Meta-Meta-Model. To customize the XFeature tool for design rationale management, we have specified the *Design Rationale Meta-Model* as an extension of the Feature Meta-Meta-Model. This specification is an XML schema, which is used by the XFeature tool to configure itself accordingly (e.g., visual editor, meta-model conformance checking) [28]. As an instance of the Design Rationale Meta-Model, we have defined the *Fault Tolerance Family Model*, which defines the fault tolerance techniques. The XFeature tool automatically validates the edited family models with respect to their meta-model. In this case, the *Fault Tolerance Family Model* is validated with respect to the *Design Rationale Meta-Model.*

Once the feature model for fault tolerance is defined and validated as a family model, we can define several application fault tolerance models by selecting features from the family feature model. For example, Fig. 6 shows a snapshot from the tool, where a feature diagram is being edited based on the generated *Fault Tolerance Family Model*. In Fig. 6, we see that a feature is selected and the properties related to the corresponding design rationale are specified. These properties (e.g., *Assumption*, *Argument*, *Criteria*) are inherited from the *Design Rationale Meta-Model* and they can be edited in the Properties Pane of the Eclipse platform as shown at the bottom

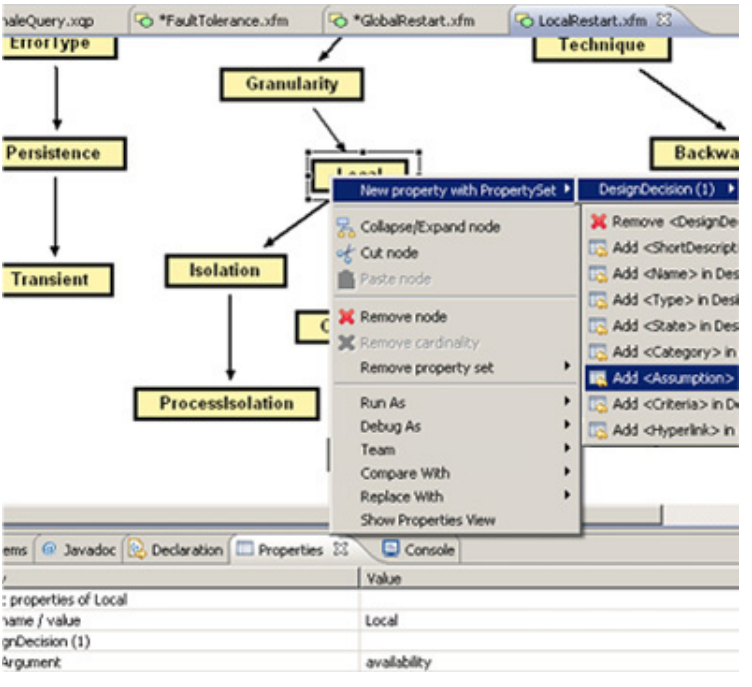


Fig. 6. Defining rationale for selected features representing architectural tactics.

of the snapshot in Fig. 6. All the properties related to the design rationale are stored together with the feature diagram as an XML file, which makes it easier to access and query the stored rationale information with existing tools.

Each feature diagram that is specified with respect to the Fault Tolerance architectural tactic space is a *Fault Tolerance View*. A fault tolerance view captures and stores the design rationale related to selections and choices made with respect to the fault tolerance architectural tactic space and it has certain design alternatives. These design alternatives are modeled with the tools of ArchStudio, which stores each of them as a separate XML file. These files are then coupled with the corresponding feature diagrams.

6.4. *Adapting architecture*

After a design template (i.e., architectural tactic, pattern, style) is selected, the next step is to adapt the architecture accordingly. Usually, there exist several additional design alternatives for adapting the architecture. These alternatives are specific to the selected design template. Here defining the design rationale requires the evaluation of several criteria and dedicated analysis techniques. For instance, [29] lists analysis techniques for fault-tolerance mechanisms that are based on design diversity and replication. In the application of such fault-tolerance mechanisms to a system, there exist choices like the number of replicated units and the parts of the system that will be replicated. These choices have an impact on the reliability achieved and the cost, which can be evaluated with the related analysis techniques. The design rationale at this step is basically formed by the criteria set being evaluated, the evaluation method (e.g., based on simulation or analytic models), types of formal models employed, their parameters, assumptions and the analysis results.

In our case study, we have developed analysis techniques for applying local recovery to architecture. We have automated the analysis process with a tool, *Recovery Designer*, which is integrated in the ArchStudio environment as a part of ArchiRationale.

For achieving local recovery, the architecture needs to be partitioned into a set of recoverable units [30]. This partitioning can be done in multiple, different ways and each alternative has an impact on *availability* and *time-performance*. Increasing the number of recoverable units will increase the availability of the system, but will decrease the time performance since more modules will be isolated from each other. On the other hand, keeping the modules together will increase performance, but will result in a lower availability since the failure of one module will affect the others as well. We have developed the *Recovery Designer* tool for evaluating the partitioning alternatives with respect to the two criteria, availability and performance, automatically.

We refer to [22] for the details of the analysis process and explanation of the tool. Obviously, the analysis results provide an important input for the design rationale. One of the decomposition alternatives is selected by evaluating analysis results and balancing the quality criteria (e.g., performance versus availability). *Recovery*

Designer also implements optimization algorithms for automatically selecting an alternative based on analysis results. Hereby, the rationale for selecting a design alternative is our goal in the optimization. This is defined by the objective function (e.g., maximize availability, minimize performance overhead).

6.5. *Querying feature diagram*

In the previous subsections we have explained *ArchiRationale* with a running example, where we start from specifying the fault-tolerance architectural tactic space, followed by feature selections in this space and defining design alternatives for the resulting feature diagram and finally applying these alternatives to architecture. In each step, there exist design decisions and justification behind these decisions (i.e., design rationale). The last step mainly involves dedicated analysis, where the rationale is basically part of the analysis methods, models and parameters as such can be replayed on-line. In the previous steps, however, the design rationale is captured as annotations on the feature diagram. Effective retrieving and reasoning about this information is essential for its reuse.

We have integrated the XQuery Development Tools in *ArchiRationale* to provide support for design rationale retrieval. We have created and validated parameterized query functions defined to query the captured design rationale, which is stored as an XML file. Example predefined query functions are *getDesignDecisions()*, *getDesignOptions()*, *getDecisionProperties()*, etc. These and new query functions can be used for searching and retrieving annotations on feature diagrams that confirm to our design rationale meta-model.

7. Characterizing the Approach

In the following we will first characterize *ArchiRationale* and then discuss the key distinctive properties with respect to other rationale management approaches.

7.1. *Key properties of archirationale*

In different surveys about rationale management systems different characteristics have been described to distinguish rationale management systems. Table 1 depicts the characterization of *ArchiRationale* which is nearly similar to the table that is used in [31]. The left column shows a list of properties that can be used to characterize rationale management systems [31]. The right column defines the values of these properties for *ArchiRationale*.

The *Goal* of *ArchiRationale* is to capture the design rationale in the context of adapting the architecture and as such to provide support for adapting the architecture.

System Type represents whether the rationale system is process-based or feature-based. As we have explained in Sec. 3 *ArchiRationale* is feature-based. Further it is important to note that features are not the properties of the system but rather techniques for realizing a particular quality concern.

Table 1. Characterization of ArchiRationale.

Goal	Support for Design Adaptation
System Type	Feature-Based, Features are techniques to realize quality
Services	Design Documentation, Constraint Checking, Design Adaptation
Represented Information	Feature Decisions, Design Options, Design Alternatives (three layers)
Representation Method	Formal (features) and semi-formal (decisions)
Capture Method	Methodological by-product after the initial architecture has been designed
Access Method	User-Initiated
Domain	Techniques to implement Quality Concerns (Architectural Tactic Space)
Design Type	Adaptation
Design Phase	Post-Architecture Design, Architectural Maintenance
Number of Designers	Any
Notation	Feature Modeling; Architecture description, XML-based

ArchiRationale provides different *Services* including Design Documentation, Constraint Checking, and Design Adaptation. Design Documentation is supported through storing (*Represented Information*) the family feature model, application feature model, design templates and design alternatives. Constraint checking is defined by validation through the *Design Rationale Metamodel* and the constraints defined in the *Family Feature Model*.

In *ArchiRationale* besides of the formal descriptions (features and design alternatives) also semi-formal decisions are stored, including argumentation and assumptions.

The design rationale is a *methodological by-product* after the initial architecture has been designed. The tool provides means to browse both the design alternatives and the rationale (selected features, arguments, assumptions, criteria).

Design rationale retrieval can be classified as user−initiative or system−initiative. In *ArchiRationale* the design rationale capture is user-initiated because the architect needs to explicitly store the design rationale.

The notation that is used to represent design rationale is based on the metamodel as defined in Sec. 4. *ArchiRationale* stores all the necessary information as XML documents.

7.2. *Distinctive properties of archirationale*

ArchiRationale is in fact complementary to other rationale management or architecture knowledge management tools as it has been defined in the different survey papers [4, 9, 35]. In this paper we will not repeat the survey of existing approaches but rather pinpoint the issues that differ or are more explicit in *ArchiRationale* with respect to other approaches:

- *Focus on adaptation of architecture*
  In general architecture rationale systems tend to focus on supporting the rationale for the development of an architecture. In *ArchiRationale* we have deliberately

narrowed the scope of the rationale by considering only the rationale for an existing architecture that needs to be adapted. In general this is a useful decision because it is very hard to maintain the complete rationale of a system, and very often organizations have to build on existing architecture rather than developing an architecture from scratch.

- *Defining rationale based on quality concern*
  The rationale management system in *ArchiRationale* is based on the adaptation of the architecture for a quality concern and does not include other decisions that could define the boundaries of the architecture. In this paper we have focused on adapting the architecture for fault tolerance. Similarly other quality concerns might be considered in *ArchiRationale* as well.
- *Focus on well-defined domain*
  ArchiRationale focuses in particular on well-defined domains for adapting the architecture. In this paper we have illustrated this for fault tolerance, which is a well-defined and mature domain. Because of the maturity of the fault tolerance domain we could easily define this as a space of architectural tactics, represented in a feature model, and use this to define the rationale for adapting the architecture. This is fundamentally different from process oriented approaches which are usually adopted for domains that are not stable yet.

In fact, the combinations of these three issues together with tool support that is built on existing architecture design tools makes *ArchiRationale* a distinctive and practical architecture rationale management system.

8. Related Work

Design rationale has been studied in different disciplines including engineering design in AI [6], human computer interaction [5] and software engineering [8]. Various surveys have been published that compare different systems that capture and use design rationale [14, 4, 9]. These studies have shown that design rationale is considered important by practitioners but it is rarely captured in practice.

The discussion around design rationale in general seems not to be different for software architecture in particular. The discussion on software architecture design rationale is actually not new and has already been initiated in the early foundations. An important related topic for architecture design rationale is certainly the work on architectural description using multiple views approaches [1]. One of the key goals of representing architectures using different architectural views is in fact the support for understanding and communication the rationale for the design decisions from the perspective of multiple concerns. Design rationale can also be considered as a documentation of the architecture but it differs in the sense that it documents more than just the end-result.

Several architecture design rationale approaches have been proposed in the literature [9, 10, 12–15, 32, 33]. Most of these apply a metamodel to support the

approach. In [10] a framework is provided for supporting architectural knowledge and rationale. In [32] Tyree and Akerman propose a detailed template for capturing the rationale of design decisions to understand the impact of the choices that are made by architects. In [16] Kruchten proposes classifications of design decisions and the relationship between them.

Architectural patterns or tactics describe common architectural strategies and design decisions [2]. Both provide hints about what kind of design decisions can be used but they do not provide a complete design decision perspective. In *ArchiRationale* design templates are linked to the selected features that led to the selected design template and as such supports design rationale. Further design alternatives for each design template are also stored in the repository and can be accessed.

In [13, 33] the authors discuss the relation between patterns and decision making and describe how architects can use patterns to capture certain architectural decisions in practice. In principle we could also use similar approaches to define and enhance the design templates in *ArchiRationale*.

9. Conclusion

Documenting and usage of design rationale is important to support communication, design evolution, design maintenance, design verification, and design reuse. In practice, however, it appears that architecture design rationale is still not being documented in a consistent manner because of lack of appropriate methodology and tool support. Design rationale often plays an important role when an existing software architecture needs to be enhanced to meet quality concerns. In this paper we have defined an approach and the tool *ArchiRationale* for capturing and accessing architecture design rationale for adapting an architecture for quality concerns. Our approach builds on and is in parts complementary to existing design rationale systems. Concretely, the contributions of this paper are the following. First we have provided a meta-model that defines the concepts and the relations among the architecture and the rationale management system. Second, we have provided a systematic rationale management approach for documenting and accessing the rationale for architecture design alternatives. Third, we provide an integrated tool environment *ArchiRationale* that supports the capturing and access of design rationale and the related artifacts. We have illustrated the approach for adapting the architecture for fault tolerance. However, the approach is general enough to be used for other quality concerns as well. We consider this as part of our future work.

Acknowledgments

We thank members of the TRADER project, for their feedback on earlier versions of this paper and their input about the TV domain knowledge and reliability issues.

References

1. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd edn. (Addison-Wesley Professional, 2011).
2. F. Bachmann, L. Bass, and M. Klein, Deriving Architectural Tactics: A Step Toward Methodical Architectural Design, CMU/SEI-2003-TR-004, ADA413644, Pittsburgh, PA, March 2003.
3. L. Dobrica and E. Niemela, A survey on software architecture analysis methods, *IEEE Trans. on Software Engineering* **28**(7) (2002) 638–654.
4. W. C. Regli, X. Hu, M. Atwood and W. Sun, A survey of design rationale systems: Approaches, representation, capture and retrieval, *Engineering with Computers* **16** (2000) 209–235.
5. J. Lee and K. Lai, What's in design rationale, *Human-Computer Interaction* **6**(3–4) (1991) 251–280.
6. J. Lee, Design rationale systems: Understanding the issues, AI in design, *IEEE Expert*, May/June 1997, pp. 78–85.
7. M. Babar, T. Dingsøyr, P. Lago and H. V. Vliet, *Software Architecture Knowledge Management: Theory and Practice* (Springer, 2009).
8. A. H. Dutoit, R. McCall, I. Mistrik and B. Paech, Rationale management in software engineering: Concepts and technuques, in A. H. Dutoit, R. McCall, I. Mistrik and B. Paech (eds.), *Rationale Management in Software Engineering* (Springer, 2007), pp. 1–48.
9. A. Tang, M. A. Babar, I. Gorton and J. Han, A survey of architecture design rationale, *Journal of Systems and Software* **79** (2006) 1792–1804.
10. R. Capilla, Embedded design rationale in software architecture, in *Proc. of Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 2009, pp. 305–308.
11. M. A. Babar, I. Gorton and B. Kitchenham, A framework for supporting architecture knowledge and rationale management, in A. H. Dutoit, R. McCall, I. Mistrik and B. Paech (eds.), *Rationale Management in Software Engineering* (Springer, 2007), pp. 237–254.
12. L. Bass, P. Clements, R. L. Nord and J. Staford, Capturing and using rationale for a software architecture, in A. H. Dutoit, R. McCall, I. Mistrik and B. Paech (eds.), *Rationale Management in Software Engineering* (Springer, 2007), pp. 237–254.
13. N. B. Harrison, P. Avgeriou and U. Zdun, Using patterns to capture architectural decisions, *IEEE Software* **24**(4) (2007) 38–45.
14. A. G. Jansen and J. Bosch, Software architecture as a set of architectural design decisions, in *Proc. of 4th Working IEEE/IFIP Conf. Software Architecture* (IEEE CS Press, 2005), pp. 109–119.
15. P. Kruchten, P. Lago and H. V. Vliet, Building up and reasoning about architecture knowledge, in *Proc. of the 2nd International Conference on Quality of Software Architectures*, 2006.
16. P. Kruchten, A taxonomy of architectural design decisions in software intensive systems. in *Proc. of the 2nd Groningen Workshop on Software Variability Management*, 2004, pp. 54–61.
17. K. Czarnecki and U. Eisenecker, *Generative Programming — Methods, Tools and Application* (Addison-Wesley, 2000).
18. F. Bachmann and L. Bas, Introduction to the attribute driven design method, in *Proc. of 23rd International Conference on Software Engineering*, 2001, pp. 745–746.
19. G. Candea, J. Cutler and A. Fox, Improving availability with recursive microreboots: A soft-state system case study, *Performance Evaluation* **56**(1–4) (2004) 213–248.

20. A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing* **1**(1) (2004) 11−33.
21. J. B. Dugan, Software system analysis using fault trees, Chapter 15 in *Handbook of Software Reliability Engineering*, ed. M. R. Lyu, (McGraw-Hill, New York, 1996), pp. 615−659.
22. H. Sozer, Architecting Fault-Tolerant Software Systems, PhD Thesis, University of Twente, 2008.
23. L. Kean, Feature-Based Design Rationale Capture Method for Requirements Tracing, Technical report, CMU-SEI, 1997.
24. J. E. Conklin and K. C. Burgess Yakemovic, A process-oriented approach to design rationale, *Human Computer Interaction* **6**(3&4) (1991) 357−391.
25. W. Kunz and W. Rittel, Issues as elements of information systems, Working paper 131, Center for Planning and Development Research, University of California, Berkeley, 1970.
26. A. MacLean, R. Young, V. Belloti and T. Moran, Questions, options, and criteria: Elements of design space analysis, *Human Computer Interaction* **6**(3&4) (1991) 201−250.
27. R. J. McCall, PHI: A conceptual foundation for design hypermedia, *Design Studies* **12**(1) (1991) 30−41.
28. XFeature official web site, http://www.pnp-software.com/XFeature, accessed 2011.
29. J. B. Dugan and M. R. Lyu, Software fault tolerance, in *Dependability Modeling for Fault-Tolerant Software and Systems*, ed. M. R. Lyu (Wiley, New York, 1995), pp. 109−138.
30. H. Sozer and B. Tekinerdogan, Introducing recovery style for modeling and analyzing system recovery, in *Proc. of the WICSA Conference*, 2008, pp. 167−176.
31. J. Burge, Design rationale, Technical report, Worcester Polytechnic Institute, Computer Science Dept., http://www.cs.wpi.edu/Research/aidg/DRRpt98.html (accessed: October 2011), 1998.
32. J. Tyree and A. Akerman, Architecture decisions: Demystifying architecture, *IEEE Software* **22**(2) (2005) 19−27.
33. W. Wang and J. E. Burge, Using rationale to support pattern-based architectural design, in *Proc. of 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge*, New York, USA, 2010, pp. 1−8.
34. M. Shahin, P. Liang and M. R. Khayyambashi, Rationale visualization of software architectural design decision using compendium, in *Proc. of 2010 ACM Symposium on Applied Computing*, New York, USA, 2010, pp. 2367−2368.
35. P. Liang and P. Avgeriou, Tools and technologies for architecture knowledge management, in *Software Architecture Knowledge Management: Theory and Practice* (Springer, 2009), pp. 91−111.