Chapter 4
Traceability in the Co-evolution of Architectural Requirements and Design

Antony Tang, Peng Liang, Viktor Clerc, and Hans van Vliet

Abstract Requirements and architectural design specifications can be conflicting and inconsistent, especially during the design period when requirements and architectural design are co-evolving. One reason is that stakeholders do not have up-to-date knowledge of each other's work to fully understand potential conflicts and inconsistencies. Specifications are often documented in a natural language, which also makes it difficult for tracing related information automatically. In this chapter, we introduce a general-purpose ontology that we have developed to address this problem. We demonstrate an implementation of semantic wiki that supports traceability of co-evolving requirements specifications and architecture design.

4.1 Introduction

Let us begin by considering a typical software architecting scenario:

> A team of business analysts and users work on a new software system in an organization. The business analysts and users document the business goals, use-case scenarios, system and data requirements in a requirements document. The team of software and system architects studies this document, which is in a draft version, and they start to create some designs. The architects realize that more information from the stakeholders is required, and they must validate the usability requirements with the operators to ensure they understand the efficiency requirements of the user interface; they also realize that they must understand the data retention and storage requirements from the business managers; finally, they have to analyze the performance requirements of the system. They find that the performance of retrieving data is slow and that hinders the data entry task. They have to discuss and resolve this issue together with the business analysts who represent the business operation unit. In the meantime, the business analysts have decided to add new functionalities to the system ...

In this scenario, many people are involved in the development of the system, and the knowledge used in the development is discovered incrementally over time. Common phenomena such as this occur every day in software development. Three problematic situations often arise that lead to knowledge communication issues in software design.

P. Avgeriou et al. (eds.), *Relating Software Requirements and Architectures*,
DOI 10.1007/978-3-642-21001-3_4,

The first problematic situation is that *knowledge is distributed*. System development always involves a multitude of stakeholders and each stakeholder possesses only partial knowledge about some aspects of a system. In this case, business users only know *what* they want, but they do not know *how* to make it work, and vice versa for the architects. In general, requirements are specified by many stakeholders such as end-users, business managers, management teams, and technology specialists. Architecture designs, in turn, are specified by architects, application software designers, database specialists, networking specialists, security specialists, and so on. As a result, the requirements and architectural design specifications that are created by different stakeholders are often conflicting and inconsistent.

Secondly, *information is imperfect*. Not all information about requirements and architecture design is explicitly documented and retrievable. The requirements and architecture design are for the most part recorded in specifications but some knowledge will remain only in the heads of those who are deeply involved in the software development project. The vast number of requirements and design entities in large-scale systems can potentially hide requirements and design conflicts. These conflicts can remain undetected until the relevant design concerns are considered in certain views and with certain scenarios. Additionally, not all relationships between the design entities and the requirements statements are captured sufficiently in the specifications to allow stakeholders to detect potential conflicts.

Thirdly, *requirements and architecture design can co-evolve over time*. Requirements and insight into how these requirements may be implemented evolve over time through exploration, negotiation, and decision-making by many people. In the scenario given at the beginning of this chapter, architects understand the performance constraints in data retrieval that the business users have no knowledge of. Because of the performance constraint, compromises in the design and requirements will have to be made. Sometimes, requirement decisions that have profound impact on the architecture design can be made before the start of the design activities. In this way, requirements documents can be signed off before architecture design commences. However, agreeing on these important requirement decisions is not always possible.

Owing to these issues, it is obvious that the development of requirements specifications and the architectural design specifications would overlap in time, implying that these specifications can co-evolve simultaneously. In order to allow stakeholders to communicate the potential impacts and conflicts between requirements and the architectural design during their co-evolution, different stakeholders must be able to trace between requirements and design to assess the viability of the solution during this process.

Traceability between requirements and design has been studied previously [1–4]. These methods use static trace links to trace different types of requirements, design, and code objects. They employ different ways to construct traces. However, these methods suffer from two issues: (a) the need to laboriously establish the trace links and maintain them as a system evolves; (b) they do not support on-going design activities. An improvement to these methods is to provide dynamic tracing at different levels of design abstraction. An example of this dynamism is a scoped

approach to the traceability of product line and product levels [5]. However, this approach is not suitable for general purpose traceability of requirements to architecture design.

In this research, we investigate how requirements and design relationships can become traceable when requirements and design objects are both incomplete and evolving simultaneously, and the static trace links used by conventional traceability methods are insufficient and out-of-date. Our work provides a general ontological model to support the traceability of co-evolving architectural requirements and design. Based on this ontology, we have applied semantic wikis to support traceability and reasoning in requirements development and architecture design.

This remaining of this chapter is organized as follows. Section 4.2 describes the issues on current traceability management from requirements to architecture design. Section 4.3 presents the traceability use cases for co-evolving architecture requirements and design with a metamodel that supports this traceability. Section 4.4 introduces the implementation of Software Engineering Wiki (SEWiki), a prototype tool that supports the dynamic traceability with an underlying ontology based on the traceability metamodel. Section 4.5 presents three concrete examples of using SE-Wiki to perform the traceability use cases. We conclude this chapter in Section 4.6.

4.2 Issues in Finding the Right Information

Requirements traceability is the ability to describe and follow the life of requirements [1]. Ideally, such traceability would enable architects and designers to find all relevant requirements and design concerns for a particular aspect of software and system design, and it would enable users and business analysts to find out how requirements are satisfied. A survey of a number of systems by Ramesh and Jarke [2] indicates that requirements, design, and implementation ought to be traceable to ensure continued alignment between stakeholder requirements and various outputs of the system development process. The IEEE standards recommend that requirements should be allocated, or traced, to software and hardware items [6, 7].

On the other hand, [1] distinguishes two types of traceability: *pre-requirements specification* and *post-requirements specification*. The difference between these two traceability types lies in *when* requirements are specified in a document. With the emergence of agile software development and the use of architecture frameworks, the process of requirements specification and design becomes more iterative. As a result, the boundary between pre- and post-requirement traceability is harder to define because of the evolving nature of requirements specification activity.

In this section, we examine the knowledge that is required to be traced, the challenges of using conventional requirements traceability methods that are based on static information, and compare that with an environment where information

changes rapidly and the capabilities to trace such dynamic requirements information must improve.

4.2.1 Architectural Knowledge Management and Traceability

Architectural knowledge is the integrated representation of the software architecture of a software-intensive system (or a family of systems), the architectural design decisions, and the external context/environment. For facilitating better design decision-making, architects require "just-in-time" knowledge [8]. Just-in-time knowledge refers to the right architectural knowledge, provided to the right person, at any given point in time.

Architectural knowledge should capture not just the outcomes of a design but also the major architectural decisions that led to it [9]. Capturing the architectural decisions facilitates a better decision-making process in shorter time, saving rework and improving the quality of the architecture [10, 11]. Hence, it is important to not only trace to the resulting architecture design, but also to the decisions, including their rationale, that led to that design.

Sound management of architectural knowledge can help in providing just-in-time knowledge by building upon two important knowledge management strategies [12]. *Personalisation* implies providing knowledge that urges the knowledge workers to interact with each other, by making known who possesses certain knowledge. *Codification*, on the other hand, focuses on identifying, eliciting, and storing the knowledge in e.g., repositories.

A hybrid strategy that uses both personalisation and codification aspects can be beneficial to sound architectural knowledge management, especially in the iterative process of architecting. When tracing back and forth between requirements and architecture, architects need specific support with adequate information relevant for addressing the design issues at hand. Hence, the proposed traceability method using semantic wikis is aligned with the current knowledge management strategy.

4.2.2 Requirements and Architecture Design Traceability

During the development life cycle, architects and designers typically use specifications of business requirements, functional requirements, and architecture design. Traceability across these artifacts is typically established as a static relationship between entities. An example would be to cross-reference requirement *R13.4* which is realized by module *M_comm()*. It is argued by [3] that relating these pieces of information helps the designers to maintain the system effectively and accurately, and it can lead to better quality assurance, change management, and software maintenance. There are different ways in which such traceability between requirements and architecture design can be

achieved. Firstly, use a traceability matrix to associate requirements to design entities in a document [13]. This is typically implemented as a table or a spreadsheet. Traceability is achieved by finding the labels in a matrix and looking up the relevant sections of the documents. Secondly, use a graphical tool in which requirements and design entities are represented as nodes and the relationships between them as arcs. Traceability is achieved by traversing the graph. Examples of such a system are provided by [2, 14]. Thirdly, use some keyword- and metadata-based requirements management tools. The metadata contains relationships such as *requirement X* is realized by *component Y*. The user would, through the tool, access the traceable components. Examples of such systems are DOORS [15], RequisitePro [16], and [17]. Fourthly, automatically generate trace relationships through supporting information such as source code [4], or requirements documents [18, 19].

Traceability is needed not only for maintenance purpose when all the designs are complete and the system has been deployed; static traceability methods can work well under this circumstance. Traceability is also needed when a system design is in progress, and the relationships between requirements and design entities are still fluid. The following scenarios are typical examples:

- When multiple stakeholders make changes to the requirements and the architecture design simultaneously during development
- Stakeholders are working from different locations and they cannot communicate proposed changes and ideas to the relevant parties instantly
- Requirements decisions or architectural decisions may have mutual impact on each other, even conflict with each other, but these impacts are not obvious when the two parties do not relate them

Under these circumstances, static traceability methods would fail because it is difficult to establish comprehensive traceability links in a documentation-based environment. In real-life, potential issues such as these are discussed and resolved in reviews and meetings. Such a solution requires good communication and management practice for it to work. A solution was proposed to use events to notify subscribers who are interested in changes to specific requirements [20]. This, however, would not serve for situations in which many new requirements and designs are being created.

In order to address this issue, this chapter outlines the use of a query-based traceability method to allow architects and requirements engineers to find relevant information in documents. This method applies a software engineering ontology to requirements and architecture design documentation.

4.2.3 Applying Semantic Wikis in Software Engineering

Software development is from one perspective a social collaborative activity. It involves stakeholders (e.g., customers, requirements engineers, architects,

programmers) closely working together and communicating to elicit requirements and to create the design and the resulting software product. This collaboration becomes more challenging when an increasing number of projects are conducted in geographically distributed environments – Global Software Development (GSD) becoming a norm. In this context, many CSCW (Computer Supported Collaborative Work) methods and related tools have been applied in software engineering to promote communication and collaboration in software development [21], but the steep learning-curve and the lack of openness of these methods and tools inhibit their application in industrial projects.

Semantic wikis combine wiki properties, such as ease of use, open collaboration, and linking, with Semantic Web technologies, such as structured content, knowledge models in the form of ontologies, and reasoning support based on formal ontologies with reasoning rules [22, 23]. As such, a semantic wiki intends to extend wiki flexibility by allowing for reasoning with structured data: semantic annotations to that data correspond to an ontology that defines certain properties. Once these semantic annotations are created, they are then available for extended queries and reasoning [22]. The combination of these features provides an integrated solution to support social collaboration and traceability management in software development. From one perspective, semantic wikis can facilitate social collaboration and communication in software development. Normal wikis have been used by the software industry to maintain and share knowledge in software development (e.g., source code, documentation, project work plans, bug reports, and so on) [24], requirements engineering [25], and architecture design [26]. With the semantic support of an underlying ontology and semantic annotations, semantic wikis can actively support users in understanding and further communicating the knowledge encoded in a wiki page by – for example – appropriately visualizing semantically represented project plans, requirements, architecture design, and the links between them [22]. From the other perspective, the underlying ontologies that support semantic wikis are composed of the concepts from software engineering and the problem domains, and the relationships between these concepts can be formally specified by the RDF [27] and OWL [28] ontology languages. This ontology representation helps users to search for semantic annotations encoded in the semantic wikis through concept relationships and constraints, and provides reasoning facilities to support dynamic traceability in software development.

Semantic wikis have been applied to different areas of software engineering, mostly in research environments. One application focuses on combining documents from Java code, and to model and markup wiki documents to create a set of consistent documents [29]. Ontobrowse was implemented for the documentation of architecture design [30]. Softwiki Ontology for Requirements Engineering (SWORE) is an ontology that supports requirements elicitation [31]. So far, we know of no ontological model that supports the traceability between requirements and architectural design.

4.3 What Needs to be Traced and Why?

4.3.1 Architectural Design Traceability

Many requirements traceability methods implicitly assume that a final set of requirements specifications exists from which traceability can be performed. Some methods require users to specify the traces manually [32], whilst others automatically or semi-automatically recover trace links from specifications [3, 17]. The assumption that a definitive set of unchanging documents exists does not always hold because tracing is also required when requirements and architecture design are being developed. This is a time when requirements and architecture design co-evolve. Architectural design activities can clarify non-functional requirements and trade-offs can compromise business requirements. During this time, a set of final specifications are not ready but traceability between related items can help architects find their ways.

Traceability between requirements and architecture design is generally based on the requirements and design specifications, but the other types of documented knowledge should also be traceable to the architecture design. This knowledge often defines the context of a system, e.g., technology standards that need to be observed in a design or the interface requirements of an external system.

In discussing the support for the traceability of group activities, [1] noted that *Concurrent work is often difficult to coordinate, so the richness of information can be lost*. There are some issues with supporting concurrent updates. Firstly, the information upon which a trace link is based has changed. For example, the requirement statement has changed. The trace link will need to be investigated and may be updated because information that is linked through it may be irrelevant or incorrect. It is laborious and therefore error prone to keep trace links up to date as requirements and designs change. Secondly, many decision makers exist and many parts of the requirements and designs can be changed simultaneously. In this situation, not all relevant information can be communicated to the right person at the right time. For instance, a business user adding a requirement to the system may not know that this change has a performance impact on the architecture design, thus she/he may not be aware that such a decision requires an architectural design assessment. In this case, some hints from an intelligent tracing system could help to highlight this need.

4.3.2 Traceability Use Cases in Co-evolving Architectural Requirements and Design

In order to develop traceability techniques to support requirements-architecture design co-evolution, we have developed a set of traceability use cases. These use

cases show examples of typical activities of architects that require support by a reasoning framework (see Sect. 4.1). The use cases are described following a technique introduced in [33] providing a scenario, problem and solution description, and a detailed description of the scenario.

Scenario 1 – Software Reuse An architect wants to check if existing software can be reused to implement a new functional requirement, and the new functionality is similar to the existing functionality.

Problem The architect needs to understand the viability of reusing software to satisfy existing and new functional and quality requirements.

Solution The architect first finds all the architecture components that realize the existing functional requirements which are similar to the new functional requirement. Then, the architect can trace the existing architecture components to determine what quality requirements may be affected, and whether the existing software is supporting the new requirement.

Scenario description

1. The architect thinks that the existing software can support a new functional requirement which is similar to existing functional requirements.
2. The architect selects the existing functional requirements and identifies all the software components that are used to realize them.
3. For each software component found, the architect identifies the related architectural structure and the quality requirements.
4. The architect assesses if the existing quality requirements are compatible with the quality requirements of the new functional requirement.
5. If so, the architect decides to reuse the components to implement the new functional requirement.

Scenario 2 – Changing Requirement An architect wants to update the architecture design because of a changing functional requirement.

Problem The architect needs to understand the original requirements and the original architecture design in order to cater for the change.

Solution The architect first finds all existing requirements that are related to the changing requirement. Then the architect identifies the decisions behind the original design. The architect can assess how the changing requirement would affect related existing requirements and the original design.

Scenario description

1. The architect identifies all the related artifacts (e.g., related requirements, architectural design decisions, and design outcomes) concerning the changing requirement.
2. The architect evaluates the appropriateness of the changing requirement with related existing requirements.
3. The architect extracts previous architectural design decisions and rationale for the changing requirement.
4. The architect identifies new design issues that are related to the changing requirement.

5. The architect proposes one or more alternative options to address these new issues.
6. The architect evaluates and selects one architectural design decision from alternative options. One of the evaluation criteria is that the selected decision should not violate existing architectural design decisions and it should satisfy the changing requirement.
7. The architect evaluates whether the new architectural design outcome can still satisfy those non-functional requirements related to the changing functional requirement.

Scenario 3 – Design Impact Evaluation An architect wants to evaluate the impact a changing requirement may have on the architecture design across versions of this requirement.

Problem The architect needs to understand and assess how the changing requirement impacts the architecture design.

Solution The architect finds all the components that are used to implement the changing requirement in different versions, and evaluates the impact of the changing requirement to the architecture design.

Scenario description

1. The architect extracts all the components that realize or satisfy the changing requirement in different versions, functional or non-functional.
2. The architect finds all the interrelated requirements in the same version and the components that implement them.
3. The architect evaluates how the changes between different versions of the requirement impact on the architecture design, and can also recover the decision made for addressing the changing requirement.

In order to support these traceability scenarios, a dynamic traceability approach is needed. This approach would require the traceability relationships to remain up-to-date with evolving documentation, especially when the stakeholders work with different documents and some stakeholders do not know what others are doing. In summary, the following traceability functions need to be provided for such an approach to work effectively:

- Support the update of trace links when specification evolves – this function requires that as documents are updated, known concepts from the ontology are used automatically to index the keywords in the updated documents, thereby providing an up-to-date relationship trace information.
- Support flexible definition of trace relationships – the traceability relationships should not be fixed when the system is implemented. The application domain and its vocabulary can change and the ways designers choose to trace information may also change. Thus the trace relationships should be flexible to accommodate such changes without requiring all previously defined relationships to be manually updated.
- Support traceability based on conceptual relationships – certain concepts have hierarchical relationships. For instance, performance is a quality requirement,

response time and throughput are requirements that concretize a performance requirement. A user may wish to enquire about the quality requirements of a system, the performance requirements, or, even more specifically, the response time of a particular function.

- Concurrent use by requirements engineers and architects – business architects, requirements engineers, data architects, and software architects typically work on their respective areas concurrently. They, for instance, need to find the latest requirements that affect their design, then make some design decisions and document them. As they do, their decisions in turn may impact the others who are also in the process of designing. The concurrent nature of software development requires that this knowledge and its traces are up-to-date.

4.3.3 *Traceability Metamodel*

The Traceability metamodel for Co-evolving Architectural Requirements and Design (T-CARD) is based on the IBIS notations (Issue, Position, Argument, and Decision) [34] to represent design argumentation. This metamodel is constructed to satisfy the traceability use cases identified earlier. The concepts and the relationships of T-CARD are presented in UML notation, grouped into the problem space and the solution space, as shown in Fig. 4.1. It consists of the following concepts:

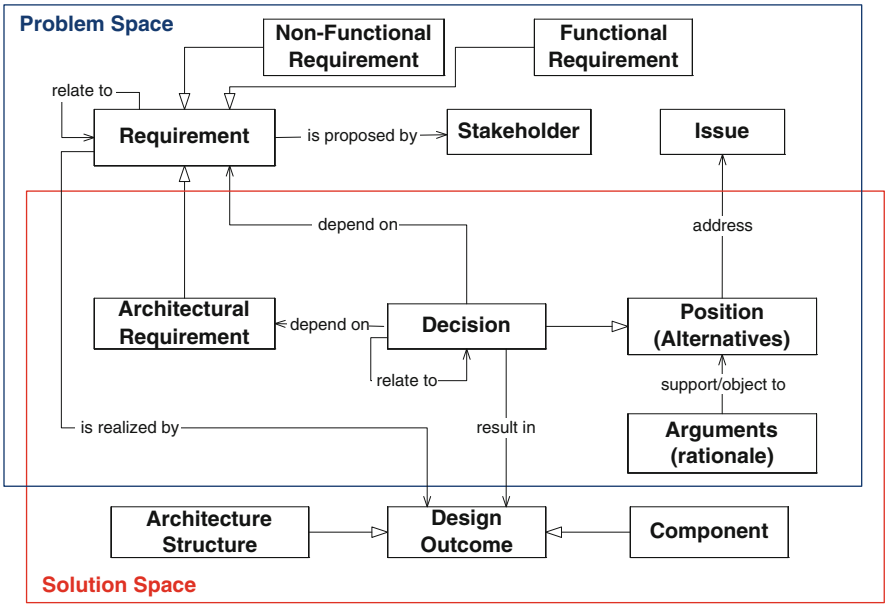

Fig. 4.1 Traceability metamodel for co-evolving architectural requirements and design

Stakeholder: refers to anyone who has direct or indirect interest in the system. A *Requirement* normally `is proposed by` a specific *Stakeholder*, which is the original source of requirements.

Requirement: represents any requirement statements proposed by a specific *Stakeholder*, and a *Requirement* can `relate to` other *Requirements*. There are generally two types of requirements: *Functional Requirements* and *Non-Functional Requirements*, and a *Requirement* `is realized by` a set of *Design Outcomes*. Note that the general relationship `relate to` between *Requirements* can be detailed further according to the use case scenarios supported.

Architectural Requirement: is a kind of *Requirement*, and *Architectural Requirements* are those requirements that impact the architecture design. An *Architecture Requirement* can also `relate to` other *Architectural Requirements*, and the `relate to` relationship is inherited from its superclass *Requirement*.

Issue: represents a specific problem to be `addressed` by alternative solutions (*Positions*). It is often stated as a question, e.g., what does the data transport layer consist of?

Position: is an alternative solution proposed to `address` an *Issue*. Normally one or more potential alternative solutions are proposed, and one of them is to be selected as a *Decision*.

Argument: represents the pros and cons argument that either `support` or `object to` a *Position*.

Decision: is a kind of *Position* that is `selected from` available *Positions* `depending on` certain *Requirements* (including *Architectural Requirements*), and a *Decision* can also `relate to` other *Decisions* [35]. For instance, a *Decision* may select some products that constrain how the application software can be implemented.

Design Outcome: represents an architecture design artifact that is resulted from an architecture design *Decision*.

Component and **Architecture Structure**: represent two types of *Design Outcomes*, that an *Architecture Structure* can be some form of layers, interconnected modules etc.; individual *Components* are the basic building blocks of the system.

The concepts in this metamodel can be classified according to the Problem and Solution Space in system development. The Problem and Solution Space overlap: *Architectural Requirement* and *Decision*, for example, belong to both spaces.

4.4 Using Semantic Wikis to Support Dynamic Traceability

The metamodel depicted in Fig. 4.1 shows the conceptual model and the relationships between the key entities in the Problem and Solution Space. This conceptual model, or metamodel, requires an ontological interpretation to define the semantics of the concepts it represents. In this section, we describe the ontology of our model to support the use cases of co-evolving architectural requirements and design.

An ontology defines a common vocabulary for those who need to share information in a given domain. It provides machine-interpretable definitions of basic concepts in that domain and the relations among them [36]. In software development, architects and designers often do not use consistent terminology. Many terms can refer to the same concept, i.e., *synonyms*, or the same term is used for different concepts, i.e., *homonyms*. In searching through software specifications, these inconsistencies can cause a low recall rate and low precision rate, respectively [30].

An ontology provides a means to explicitly define and relate the use of software and application domain related terms such as design and requirements concepts. The general knowledge about an application domain can be distinguished from the specific knowledge of its software implementation. For instance, system throughput is a general concept about quality requirements and that is measurable; it can be represented in a sub-class in the hierarchy of quality requirements class. In an application system, say a bank teller system, its throughput is a specific instance of a performance measure. Using an ontology that contains a definition for these relationships, this enables effective searching and analysis of knowledge that are embedded in software documents.

Ontology defines concepts in terms of classes. A class can have subclasses. For instance, the *throughput* class is a subclass of *efficiency*, meaning that throughput is a kind of performance measure. A throughput class can have instances that relate to what is happening in the real-world. Some examples from an application system are: *the application can process 500 transactions per second* or *an operator can process one deposit every 10 s*.

A class can be related to another class through some defined relationships. For instance, a bank teller system *satisfies* a defined throughput rate. In this case, *satisfies* is a property of the bank teller system. The property *satisfies* links a specific requirement to a specific throughput.

4.4.1 A Traceability Ontology for Co-evolving Architectural Requirements and Design

An ontology requires careful analysis and planning. If an ontology is designed for a single software application, then it may not be flexible and general enough to support other systems. To support general traceability of requirements and architecture design specifications, we define an ontology using the requirements and architecture metamodel (Fig. 4.1). The ontology (Fig. 4.2) is represented in a UML diagram that depicts the class hierarchy and the relationships between the classes. The dotted line represents the relationships between classes. The relationships are defined in terms of the properties within a class.

In this model, there are five key concepts, represented by five groups of classes. These concepts are commonly documented in requirements and architecture design

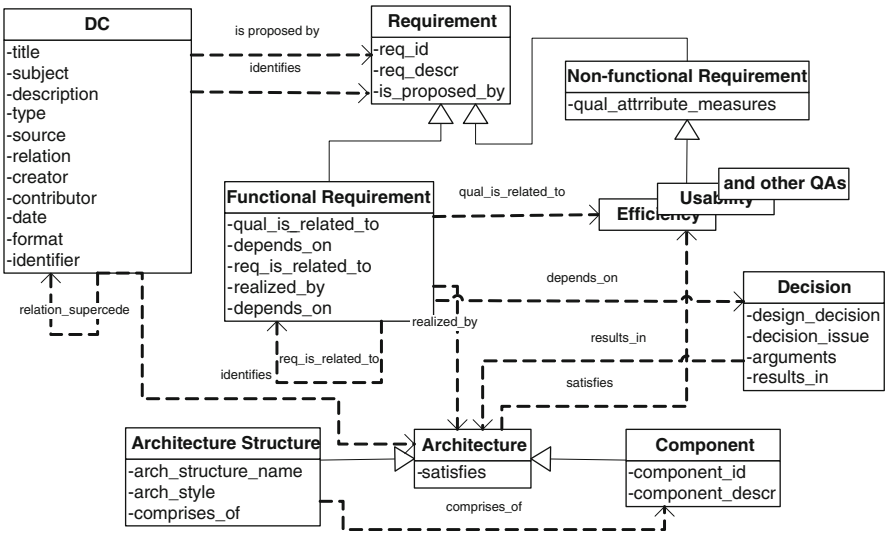

Fig. 4.2 Ontology for traceability between requirements and architecture

specifications, and the ontology is designed to represent these key concepts in software specifications:

- *DC* is a concept about the information of a document or a record. Dublin Core Metadata Initiative (DCMI) is an open organization engaged in the development of interoperable metadata standards that support a broad range of purposes and business models [37]. We make use of the concept defined in *dc:record* to identify the documents that are created for requirements and architecture purpose. In particular, we make use of the elements defined in the DC concept to support traceability of requirements and design across multiple versions of a single document. For example, a *DC* instance can identify the creator, the version, and the contributors of a requirement.
- *Requirement* is a concept that represents all the requirements of a system, including functional and non-functional requirements. A requirement has a unique identification and a description. These elements are implemented as properties (sometimes also referred to as slots) of the *Requirement* class. The properties of the *Requirement* class are inherited by all its subclasses. A requirement has an identifier and a description, so both functional and non-functional requirements have these properties as well. For example, an instance of a functional requirement would be a sub-class of *Requirement*. It would have a *req_id* of R1.1.3; a *req_descr* of *Change User Access*; it can be *realized_by* a component called *DefineAccessRight*. A user of the semantic wiki can ask the system for all requirements, and both functional and non-functional requirements would be retrieved.
- *Non-functional Requirement* represents all the quality requirements that must be satisfied by a system. Its subclasses such as efficiency and usability represent

different types of non-functional requirements. Non-functional requirements are sometimes measureable, e.g., throughputs. So, we use a property called `qual_attribute_measures` to capture this information for all measurable quality attributes.

- *Decision* represents the decisions that have been made. It has properties that capture the issues, arguments of a decision. For instance, the arguments for choosing an architecture design can be captured and linked to the design.
- *Architecture* represents the design outcomes of a decision, and the architecture realizes all requirements, both functional and non-functional. *Architecture* has two subclasses, *Architecture Structure* and *Component*. *Architecture Structure* represents the architecture styles that are used in an architecture design, such as multi-tier, web-based etc., whereas *Component* represents the individual building blocks that are used in an architecture. For instance, the ontology can capture the instances of a web-server architecture style and use the *comprise_of* property to link to *components* that generate dynamic HTML pages from a database application component.

Figure 4.2 depicts two class relationships: (a) class hierarchy represents an `is-a` relationship. So efficiency `is-a` non-functional requirement, and therefore it `is-a` requirement also; (b) a relationship between two disjoint classes is implemented through the property of a class. An example is that a requirement is proposed by a stakeholder. A stakeholder is represented in the ontology as a *dc:contributor*. In this case, both the DC record and the requirement are two disjointed classes linked together by the property field `is_proposed_by` in *Requirement* class. All the important relationships in this ontology are described below:

- A DC record `identifies` a document, be it a requirements document or an architecture design. This identification makes use of the standard elements provided by the DC metamodel. The amount of information that is contained in a document, whether it is one or a set of requirements, is up to the user. The key elements are: (a) the *title* and *subject* identify a requirement or a design; (b) the *source* identifies the version of a requirement; (c) the *relation* identifies if the document `supercedes` another document; (d) the *identifier* is the URI of the semantic wiki page. (e) the *contributor* identifies the stakeholders who contribute to the requirement or the design.
- Functional Requirement `depends_on` a decision. If a decision or a rationale of a design decision has been documented, then the requirement can be explained by the documented decision.
- Functional Requirement `qual_is_related_to` non-functional requirements. Often a requirements specification explicitly defines what quality is required by a system. In such cases, traceability can be provided if this relationship is captured in the ontology.
- Decision `results_in` an architecture. When business analysts and architects capture a decision, the outcome or the architecture design of a decision, including its rationale, can be traced to the decision. When `results_in` relationship is used in combination with the `depends_on` relationship, architects can query

what components are used to realize a specific requirement and why, for instance.

- Functional Requirement `is_realized_by` an architecture design. Designers, programmers, and testers often need to know the implementation relationships. If a decision has been documented and included in the ontology, then this relationship can be inferred from the original requirement. However, design decisions are often omitted, and so the implied realization link between requirements and design outcomes becomes unavailable. In order to circumvent this issue, we choose to establish a direct relationship between requirements and architecture.
- Architecture Design `satisfies` some non-functional requirements. This relationship shows that an architecture design can satisfy the non-functional requirements.

Together these relationships mark and annotate the texts in requirements and architecture specifications, providing the semantic meaning to enable architects and analysts to query and trace these documents in a meaningful way. Each trace link is an instance of the ontology relationships. Traceability is implemented by a semantic wiki implementation that supports querying or traversing.

4.4.2 SE-Wiki Implementation

In this section, we describe a semantic wiki implementation for Software Engineering, called SE-Wiki, which is implemented based on Semantic MediaWiki (SMW) [38]. We present how the ontology described in Sect. 4.4.1 is implemented with other semantic features in SE-Wiki. SMW is one of the prototype implementations of semantic wikis. There are two reasons for selecting SMW as the basis of SE-Wiki: (1) SMW implements most of semantic functions, including ontology definition and import, semantic annotation and traceability, and semantic query etc., which provide fundamental capabilities to perform the use cases presented in Sect. 4.3.2; and (2) SMW is a semantic extension of MediaWiki[1], which is the most popular wiki implementation on the Web, e.g., used by Wikipedia[2]. The popularity and maturity of MediaWiki will make SE-Wiki easily adoptable by industry.

The SE-Wiki uses and extends the capability of SMW by applying the semantic features in the software engineering domain, from documentation, issue tracing, reuse, and collaboration to traceability management. In this chapter, we focus on the traceability management for the co-evolution of architectural requirements and design, combined with the ontology that supports dynamic traceability between

[1] http://www.mediawiki.org/

[2] http://www.wikipedia.org/

Table 4.1 Ontology definition in SMW

Ontology construct	SMWConstruct	Example in SMW
`Class`	Category	*[[Category:Requirement]]*
`Class property`	Property	*[[req id::FR-001]]*
`Class relationship`	Property that links to the instance of other `Class`	*[[is proposed by::Stakeholder A]]*
`SubClassOf`	Category subcategorization	In the editing box of *Category:Functional Requirement*, specify *[[Category: Requirement]]*

architectural requirements and design. The implementation details of SE-Wiki are presented below.

Ontology support: as mentioned before, a semantic wiki is a wiki that has an underlying ontology that is used to describe the wiki pages or data within pages in the wiki. The ontology model elaborated in Sect. 4.4.1 is composed of four basic constructs, which can be defined in SMW as shown in Table 4.1. For example, *[[Category:Requirement]]* defines the class *Requirement*.

Semantic annotation: SMW only supports semantic annotation of wiki pages without supporting semantic annotation of data within wiki pages. This means that each semantic annotation in SMW is represented as a wiki page that belongs to a certain concept in the ontology model. In SE-Wiki, it is quite easy to annotate a requirement or architecture design artifact by adding text *[[Category:Concept Name]]* in the editing box of the wiki page based on the ontology defined or imported.

Semantic traceability refers to the semantic tracing between semantic annotations. In common wikis implementation, traceability is established by links between wiki pages without specific meaning of these links, while in semantic wikis, the semantics of these links are specified and distinguished by formal concept relationships in an ontology, which is beneficial to our purpose. For example, *Functional Requirement 001* `is_proposed_by` *Stakeholder A*. The *Functional Requirement 001* and *Stakeholder A* are semantic annotations that belong to concept *Functional Requirement* and *Stakeholder* respectively. The concept relationship `is_proposed_by` between *Functional Requirement* and *Stakeholder* is used to trace semantically the relationship between the two annotations. In SE-Wiki, a semantic tracing can be established by an instance of Property in SMW between two wiki pages (i.e., semantic annotations), e.g., for above example, we can add text *[[is proposed by::Stakeholder A]]* in the editing box of *Functional Requirement 001* to create the semantic tracing.

Semantic query is used to query semantically the data (i.e., semantic annotations recorded in SE-Wiki) with semantic query languages, e.g., SPARQL [39] or a special query language supported by SMW. The capability of semantic queries is supported by the underlying ontology of the SE-Wiki, for example, *show all the Functional Requirements proposed by Stakeholder A*. Two methods for semantic query are provided in SE-Wiki: semantic search and in-line query. Semantic search provides a simple query interface, and user can input queries and

get the query results interactively. For example, query input *[[Category:Functional Requirement]][[is proposed by::Stakeholder A]]* will return all the functional requirements proposed by *Stakeholder A*. Semantic search is applicable to temporary queries that vary from time to time. In-line query refers to the query expression that is embedded in a wiki page in order to dynamically include query results into pages. Consider this in-line query: *ask: [[Category:Requirement]][[is proposed by::Stakeholder A]] | ?is proposed by*. It asks for all the requirements proposed by *Stakeholder A*. In-line query is more appropriate in supporting dynamic traceability between software artifacts, e.g., when a functional requirement proposed by *Stakeholder A* is removed from a requirements specification, the requirements list in the wiki page of *Stakeholder A* will be updated automatically and dynamically.

Example uses of these semantic features supported in SE-Wiki for the traceability use cases are further described in the next section.

4.5 Examples of Using SE-Wiki

In this section, we present several examples of applying SE-Wiki for performing the use cases presented in Sect. 4.3.2. We show how the semantic features in SE-Wiki can be used to support the co-evolution of architectural requirements and design. We draw these examples from the NIHR (National Institute for Health Research of United Kingdom) Portal Project [40]. The system aims to provide a single gateway to access information about health research and manage the life-cycles of research projects for the broad community of NIHR stakeholders, including e.g., researchers, managers, and complete research networks. We apply the SE-Wiki to the requirements and design specifications from this project. Then we demonstrate the use cases that we have defined to show SE-Wiki support for the traceability in co-evolving architectural requirements and design.

As presented in Sect. 4.4.2, some basic semantic functions are provided by SE-Wiki, including:

Ontology support: the underlying ontology concepts and the semantic relationships between concepts are defined in SMW.

Semantic annotation is used to annotate a requirement or architecture design artifact documented in a wiki page with a concept (i.e., *Category* in SMW).

Semantic traceability is supported by semantic tracing which is established between semantic annotations, and semantic traces will follow the semantic relationships defined at the ontology level.

Semantic query: the semantic annotations of requirements or architecture design artifacts allow SE-Wiki to query the annotations semantically by simple or complex queries. Queries can be entered manually through the query interface or embedded as in-line query in the wiki pages.

With the support of these basic semantic functions, we demonstrate how the use cases presented in Sect. 4.3.2 can be achieved with the examples from the NIHR

Portal project. In order to implement the use cases, all the relevant requirements and architecture specifications must be semantically annotated based on the traceability ontology specified in Sect. 4.4.1, e.g., in a sample requirement statement: *Student would like to download course slides from course website*., *Student* is annotated as an instance of concept *Stakeholder*, *would like to* is annotated as an instance of concept relationship `is_proposed_by`, and *download course slides from course website* is annotated as an instance of concept *Requirement*.

These semantic annotations are performed by business analysts and architects as they document the specifications. The main difference between this method and some other requirements traceability methods is that individual requirement and design are semantically annotated, and their traceability is enabled by reasoning with the ontology concepts.

4.5.1 Scenario 1 Software Reuse

Description: An architect wants to check if existing software can be reused to implement a new functional requirement, which is similar to existing functional requirements that have been implemented (see Sect. 4.3.2).

Example: A new functional requirement *Track Usage: The Portal tool should be able to track usage of resources by all users* is proposed by the *Portal Manager*. The architect thinks that this new functional requirement is similar to an existing functional requirement: i.e., *Change User Access: The Portal tool should be able to change user's access rights to resources*[3]. The architect wants to check if the existing software (i.e., design outcomes/architecture) that is used to implement the requirement *Change User Access* can be reused to implement the new requirement *Track Usage*, especially with regards to the quality requirements.

Since the requirements and architecture specifications are already semantically annotated in SE-Wiki, semantic query can be employed to query the direct and indirect tracing relationships from an instance of *Functional Requirement* (i.e., the existing functional requirement *Change User Access*) to all the concerned *Design Outcomes* that `realize` this functional requirement, and all the *Non-Functional Requirements* that the *Design Outcomes* can `satisfy`. The snapshot of this scenario through semantic query is shown in Fig. 4.3. The top part of this figure is the editing box for semantic query input, and the lower part shows the query results.

As described in the example, the architect first extracts all the *Design Outcomes* that are used to `realize` the existing functional requirement *Change User Access*, and then queries all the *Non-Functional Requirements* that are `satisfied` by these *Design Outcomes*, in order to evaluate whether these *Design Outcomes* can be

[3]Resources in NIHR Portal project refer to all the information maintained by the Portal, e.g., sources of funding for different types of research.

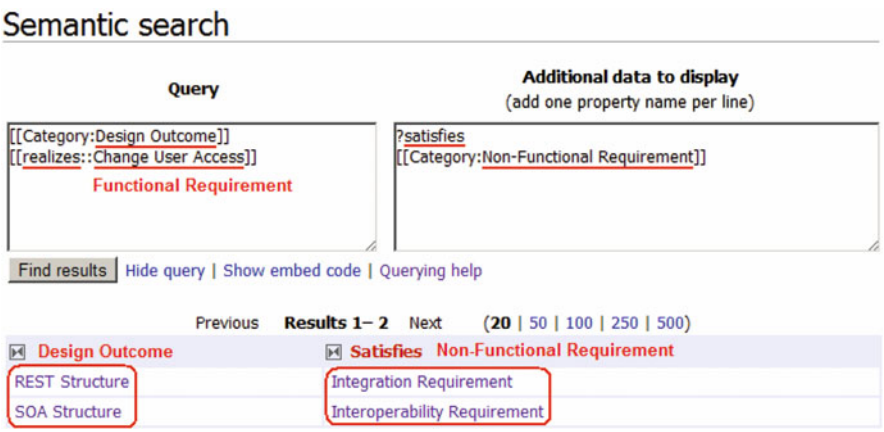

Fig. 4.3 Scenario 1 through semantic query interface in SE-Wiki

reused or not for the implementation of the new functional requirement *Track Usage*. This query is composed of two parts: the query input in the upper left of Fig. 4.3 *[[Category:Design Outcome]][[realizes::Change User Access]]* extracts all the *Design Outcomes* that realize *Change User Access* requirement, i.e., *REST Structure* and *SOA Structure*, which are directly related with *Change User Access* requirement; the query input in the upper right *?satisfies [[Category:Non-Functional Requirement]]* returns all the *Non-Functional Requirements*, i.e., *Integration Requirement* and *Interoperability Requirement*, which are indirectly related with *Change User Access* requirement through the *Design Outcomes*.

With all the *Non-Functional Requirements* and their associated *Design Outcomes* related to *Change User Access* requirement, which are all shown in one wiki page, the architect can have a whole view of the implementation context of the new functional requirement *Track Usage*, and assess the compatibility of these *Non-Functional Requirements* with the *Non-Functional Requirements* related to the new functional requirement. With this information, the architect will decide whether or not to reuse these *Design Outcomes* for the implementation of the new functional requirement *Track Usage*.

When new *Design Outcomes* are added to `realize` a requirement, in this case the requirement *Change User Access*, the semantic query will return the latest results (i.e., updated *Design Outcomes* realizing *Change User Access*). This allows SE-Wiki to support dynamic changes to requirements and architecture design which normal wikis cannot achieve with static trace links.

Under the current ontology definition, other possible software reuse scenarios can be supported by SE-Wiki, some of them are:

- Find all components that support a particular kind of quality requirements, and satisfy some quality requirements thresholds.
- Find all components that are influenced by two specific quality requirements simultaneously.

- Find the architecture design and all the components within it that support an application sub-system.
- Trace all components that are influenced by a design decision to assess if the components are reusable when the decision changes.

4.5.2 Scenario 2 Changing Requirement

Description: An architect wants to update an architecture design according to a changing requirement (see Sect. 4.3.2).

Example: A functional requirement *Change User Access: The Portal tool should be able to change user's access rights to resources.* is changed into *Change User Access: The Portal tool should only allow System Administrator to change user's access rights to resources*. Accordingly, the design based on this requirement should be updated as well. To achieve this, the architect should make sure that this changing requirement has no conflict with related existing requirements, and understand the context of this requirement before updating the design. The architect first extracts all the related artifacts concerning this changing requirement by navigating to the wiki page of this requirement in SE-Wiki, which records all the artifacts (e.g., requirements, architectural design decisions, and design outcomes) related to this requirement as shown in Fig. 4.4.

In this wiki page, the architect can easily evaluate those related artifacts concerning the changing requirement by navigating to their wiki pages. For example, the changing requirement *Change User Access* is related to the requirement *Track Usage: The Portal tool should be able to track usage of resources by all users*. There are two types of traces shown in this page: outgoing and incoming traces, which are both supported by the concept relationships defined in underlying ontology. Outgoing traces are recorded by *property*, e.g., *requirement ID*, *is proposed by*, etc. These outgoing traces show how this requirement relates to other artifacts, in a one-to-one or often one-to-many relationships. Incoming traces are shown in this page by in-line queries, which is another kind of semantic query feature provided by SE-Wiki as presented in Sect. 4.4.2. There are three in-line queries to show the incoming traces in Fig. 4.4, for example, the first incoming trace *Decision: Portal Personalization* `depends_on` *Change User Access* is created by in-line query *ask:[[Category:Decision]] [[depend on::Change User Access]] | ? depend on*. These incoming traces show how other artifacts relate to this requirement. The advantage of incoming traces generated by in-line queries is that the results of the in-line query shown in the wiki page will be updated dynamically according to the query results at run-time, which is most beneficial to evaluate and synchronize requirements and architecture design when both of them co-evolve simultaneously by different stakeholders, for example, when a new *Design Outcome* is made to `realize` the changing requirement *Change User Access*, then the incoming traces about the *Design Outcomes* that `realize` *Change User Access* will be updated automatically in this wiki page.

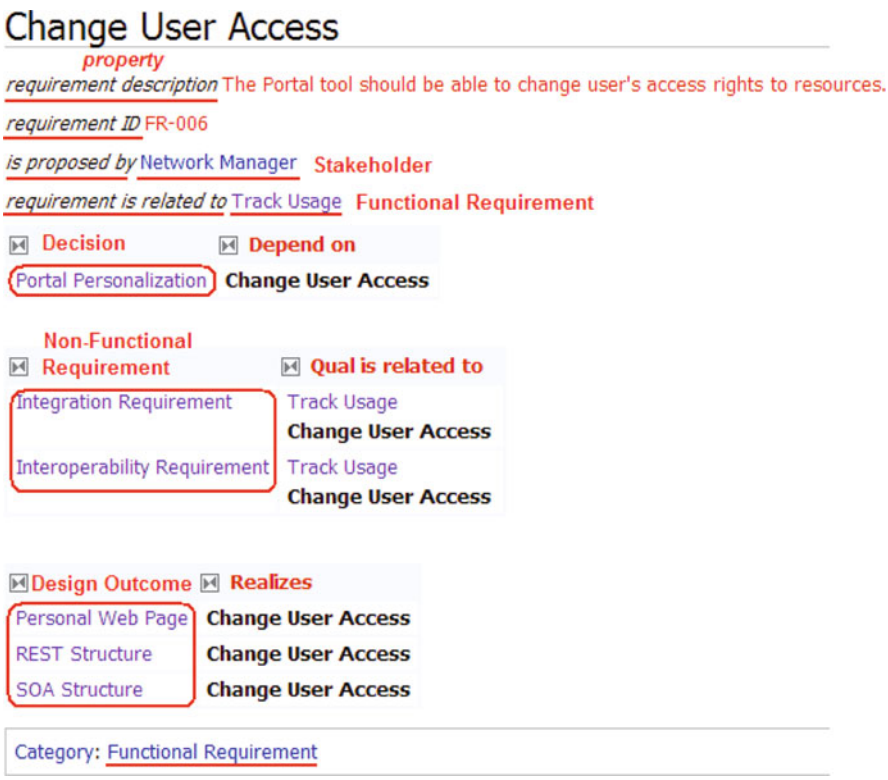

Fig. 4.4 Scenario 2 through in-line semantic query in SE-Wiki

In Scenario 2, the architect evaluates and finds that related requirement *Track Usage* is not affected by the change of requirement *Change User Access*. But the architect finds an issue *Access Control by Identity* caused by the changing requirement. To address this issue, a design option *Identity Management: Provide an identity management infrastructure in portal personalization management* is selected by the architect and documented as a *Decision*. A design outcome *Identity Management Component* is designed to `realize` the changing requirement *Change User Access*. All these updates on related artifacts are recorded in this requirement wiki page through incoming and outgoing traces as shown in Fig. 4.5.

With the information found in this page, the architect can further evaluate whether the newly-added decision *Identity Management* is compatible with other existing *Designs*, e.g., *Portal Personalization*, and whether the updated *Design Outcomes* still `satisfy` those related *Non-Functional Requirements*, e.g., *Integration Requirement*. The *Decisions* and *Design Outcomes* may change accordingly based on these further evaluations.

A number of other use cases that are similar to the changing requirement can also be supported by SE-Wiki:

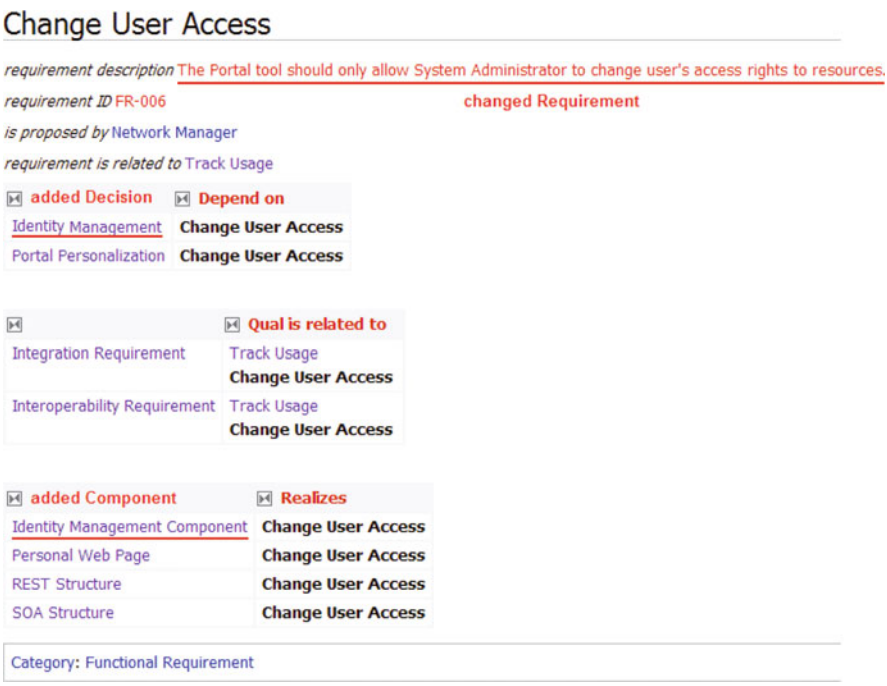

Fig. 4.5 Updated results of scenario 2 through In-line semantic query in SE-Wiki

- Find all functional requirements that may be affected by the changing requirement.
- Find all non-functional requirements that have quality impacts on a functional requirement.
- Find all functional requirements that would be influenced by a change in the non-functional characteristic of a system, e.g., performance degradation.

4.5.3 Scenario 3 Design Impact Evaluation

Description: Requirements are frequently changed from one software version to the next, and an architect tries to evaluate and identify the impacts of the changing requirements on architecture design, so that requirements and architecture design are consistent.

Example: The requirement *Change User Access* is updated in the next version, i.e., *Version 1: The Portal tool should be able to change user's access rights to resources*, and *Version 2: The Portal tool should only allow System Administrator to change user's access rights to resources*. The architect extracts different versions of the requirement with the same *requirement ID* using a semantic query

(e.g., *[[Category:Requirement]][[is identified by::DC 001]]*), in which *DC 001* is the DC element to identify the version of a requirement. The architect finds the components for implementing the requirements by clicking the wiki page of the requirement in different versions. The architect then finds the other components for implementing related requirements through reasoning support (e.g., iteratively traverse all the related requirements), which is based on the reasoning rules and relationships defined on ontology. According to the information, the architect can identify the changes to the architecture design in two sequential versions of the requirement. From that she/he can evaluate the change impacts to the architecture design. A comparison of the wiki pages of requirements across two versions (left side is a latest version of the requirement *Change User Access*, and right side is a previous version of *Change User Access*, which is superseded by the latest version) is shown in Fig. 4.6. The requirement changes between versions with changed decisions and design (circled in Fig. 4.6) will be further evaluated for design impact analysis.

A number of other use cases that employ the reasoning framework can also be performed by SE-Wiki:

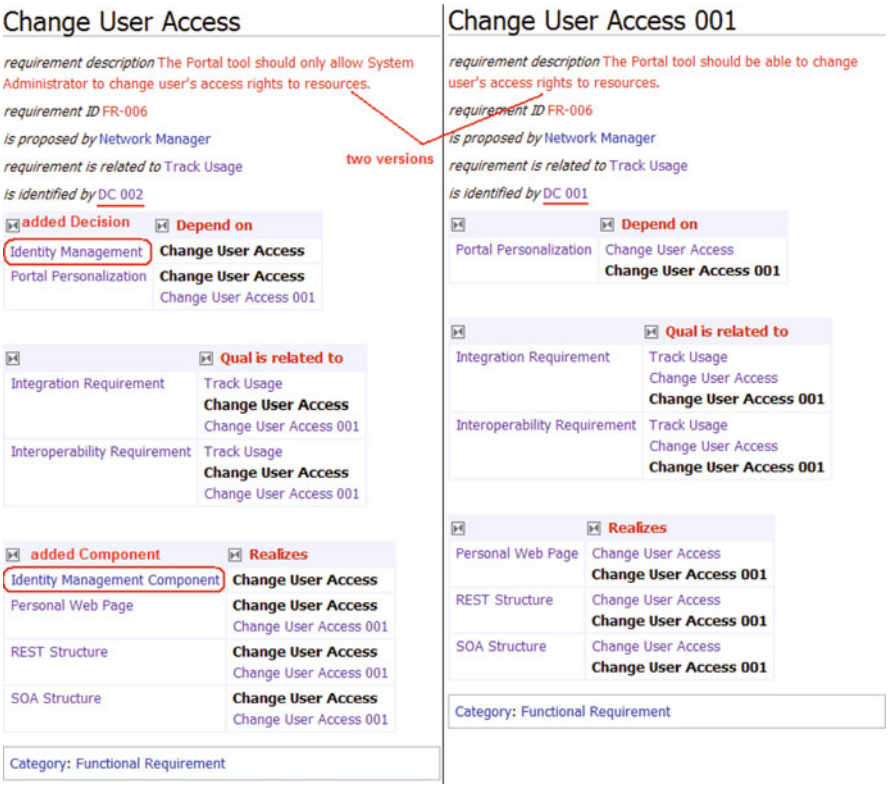

Fig. 4.6 Scenario 3 through comparison in SE-Wiki

- An architect wants to get a list of open quality requirements for which architectural decisions are needed.
- An architect wants to evaluate and detect the soundness of the software artifacts, e.g., a design decision is wanted when an architecture is used to realize a functional requirement.
- An architect can identify the architecture design components that have been changed from the previous software version.
- Analysts or architects can find the latest changes to a requirement or a design of interest.
- Analysts or architects can find changes that have been made by certain people or within a certain period of time.

4.6 Conclusions

Large-scale software development involves many people/stakeholders who develop requirements and architectural design. Often, these people are dispersed geographically, and the decisions that they make on the requirements and design evolve over time. This situation has created a knowledge communication issue that can cause conflicts and inconsistencies in requirements and design. Traceability methods based on static trace links cannot address this problem because the stakeholders often do not know what has been changed, let alone creating those trace links. Moreover, specifications and communications such as emails and meeting minutes are mostly documented in a natural language, making the search of related information difficult.

We solve this problem by providing a new method that makes use of semantic wiki technologies. We propose a general-purpose ontology that can be used to capture the relationships between requirements and architectural design. These relationships are derived from the use cases that we have identified. Semantic MediaWiki has been used to implement SE-Wiki. SE-Wiki supports a traceability metamodel and implements traceability use cases using a traceability ontology. Furthermore, SE-Wiki supports semantic annotation and traceability, and the annotated semantic wiki pages provide an information base for constructing semantic queries. This approach allows business analysts and designers to find up-to-date and relevant information in an environment of co-evolving requirements and designs.

Acknowledgments This research has been partially sponsored by the Dutch "Regeling Kenniswerkers", project KWR09164, Stephenson: Architecture knowledge sharing practices in software product lines for print systems, the Natural Science Foundation of China (NSFC) under Grant No. 60950110352, STAND: Semantic-enabled collaboration Towards Analysis, Negotiation and Documentation on distributed requirements engineering, and NSFC under Grant No.60903034, QuASAK: Quality Assurance in Software architecting process using Architectural Knowledge.

References

1. Gotel OCZ, Finkelstein ACW (1994) An analysis of the requirements traceability problem. In: IEEE International Symposium on Requirements Engineering (RE), 94–101
2. Ramesh B, Jarke M (2001) Towards reference models for requirements traceability. IEEE Trans Software Eng 27(1):58–93
3. Spanoudakis G, Zisman A, Perez-Minana E, Krause P (2004) Rule-based generation of requirements traceability relations. J Syst Softw 72(2):105–127
4. Egyed A, Grunbacher P (2005) Supporting software understanding with automated requirements traceability. Int J Software Engineer Knowledge Engineer 15(5):783–810
5. Lago P, Muccini H, van Vliet H (2009) A scoped approach to traceability management. J Syst Softw 82(1):168–182
6. IEEE (1996) IEEE/EIA Standard – Industry Implementation of ISO/IEC 12207:1995, Information Technology – Software life cycle processes (IEEE/EIA Std 12207.0–1996)
7. IEEE (1997) IEEE/EIA Guide – Industry Implementation of ISO/IEC 12207:1995, Standard for Information Technology – Software life cycle processes – Life cycle data (IEEE/EIA Std 12207.1–1997)
8. Farenhorst R, Izaks R, Lago P, van Vliet H (2008) A just-intime architectural knowledge sharing portal. In: 7th Working IEEE/IFIP Conference on Software Architecture (WICSA), 125–134
9. Bass L, Clements P, Kazman R (2003) Software architecture in practice, 2nd edn. Addison Wesley, Boston
10. Ali-Babar M, de Boer RC, Dingsøyr T, Farenhorst R (2007) Architectural knowledge management strategies: approaches in research and industry. In: 2nd Workshop on SHAring and Reusing architectural Knowledge – Architecture, Rationale, and Design Intent (SHARK/ADI)
11. Rus I, Lindvall M (2002) Knowledge management in software engineering. IEEE Softw 19(3): 26–38
12. Hansen MT, Nohria N, Tierney T (1999) What's your strategy for managing knowledge? Harv Bus Rev 77(2):106–116
13. Robertson S, Robertson J (1999) Mastering the requirements process. Addison-Wesley, Harlow
14. Tang A, Jin Y, Han J (2007) A rationale-based architecture model for design traceability and reasoning. J Syst Softw 80(6):918–934
15. IBM (2010) Rational DOORS – A requirements management tool for systems and advanced IT applications. http://www-01.ibm.com/software/awdtools/doors/, accessed on 2010-3-20
16. IBM (2004) Rational RequisitePro - A requirements management tool. http://www-01.ibm.com/software/awdtools/reqpro/, accessed on 2010-3-20
17. Hayes JH, Dekhtyar A, Osborne J (2003) Improving Requirements Tracing via Information Retrieval. In: 11th IEEE International Conference on Requirements Engineering (RE), 138–147
18. Assawamekin N, Sunetnanta T, Pluempitiwiriyawej C (2009) Mupret: an ontology-driven traceability tool for multiperspective requirements artifacts. In: ACIS-ICIS, . 943–948
19. Hayes JH, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans Software Eng 32(1):4–19
20. Cleland-Huang J, Chang CK, Christensen M (2003) Event-based traceability for managing evolutionary change. IEEE Trans Software Eng 29(9):796–810
21. Mistrk I, Grundy J, Hoek A, Whitehead J (2010) Collaborative software engineering. Springer, Berlin
22. Schaffert S, Bry F, Baumeister J, Kiesel M (2008) Semantic wikis. IEEE Softw 25(4):8–11
23. Liang P, Avgeriou P, Clerc V (2009) Requirements reasoning for distributed requirements analysis using semantic wiki. In: 4th IEEE International Conference on Global Software Engineering (ICGSE), 388–393
24. Louridas P (2006) Using wikis in software development. IEEE Softw 23(2):88–91

25. Hoenderboom B, Liang P (2009) A survey of semantic wikis for requirements engineering. SEARCH http://www.cs.rug.nl/search/uploads/Publications/hoenderboom2009ssw.pdf, accessed on 2010-3-20
26. Bachmann F, Merson P (2005) Experience using the web-based tool wiki for architecture documentation. Technical Note CMU, SEI-2005-TN-041
27. Lassila O, Swick R (1999) Resource Description Framework (RDF) Model and Syntax. http://www.w3.org/TR/WD-rdfsyntax, accessed on 2010-3-20
28. McGuinness D, van Harmelen F (2004) OWL web ontology language overview. W3C recommendation 10:2004–03
29. Aguiar A, David G (2005) Wikiwiki weaving heterogeneous software artifacts. In: international symposium on wikis, WikiSym, pp 67–74
30. Geisser M, Happel HJ, Hildenbrand T, Korthaus A, Seedorf S (2008) New applications for wikis in software engineering. In: PRIMIUM 145–160
31. Riechert T, Lohmann S (2007) Mapping cognitive models to social semantic spaces-collaborative development of project ontologies. In: 1st Conference on Social Semantic Web (CSSW) 91–98
32. Domges R, Pohl K (1998) Adapting traceability environments to project-specific needs. Commun ACM 41(12):54–62
33. Lago P, Farenhorst R, Avgeriou P, Boer R, Clerc V, Jansen A, van Vliet H (2010) The GRIFFIN collaborative virtual community for architectural knowledge management. In: Mistrk I, Grundy J, van der Hoek A, Whitehead J (eds) Collaborative software engineering. Springer, Berlin
34. Kunz W, Rittel H (1970) Issues as elements of information systems. Center for Planning and Development Research, University of California, Berkeley
35. Kruchten P (2004) An ontology of architectural design decisions in software-intensive systems. In: 2nd Groningen Workshop on Software Variability Management (SVM)
36. Noy N, McGuinness D (2001) Ontology development 101: a guide to creating your first ontology
37. Powell A, Nilsson M, Naeve A, Johnston P (2007) Dublin Core Metadata Initiative – Abstract Model. http://dublincore.org/documents/abstract-model, accessed on 2010-3-20
38. Krotzsch M, Vrandecic D, Volkel M (2006) Semantic Mediawiki. In: 5th International Semantic Web Conference (ISWC), 935–942
39. Prud'Hommeaux E, Seaborne A (2006) SPARQL Query Language for RDF. W3C working draft 20
40. NIHR (2006) NIHR Information Systems Portal User Requirements Specification. http://www.nihr.ac.uk/files/pdfs/NIHR4.2 Portal URS002 v3.pdf, accessed on 2010-3-20