An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to Other Artifacts in the Software Engineering Lifecycle

Rafael Capilla[1], Olaf Zimmermann[2],
Uwe Zdun[3], Paris Avgeriou[4], Jochen M. Küster[2]

[1] Universidad Rey Juan Carlos, Madrid, Spain
rafael.capilla@urjc.es
[2] IBM Research, Zurich, Switzerland
olz,jku@zurich.ibm.com
[3] Vienna University of technology, Vienna, Austria
uwe.zdun@univie.ac.at
[4] University of Groningen, Groningen, The Netherlands
paris@cs.rug.nl

Abstract. Software architects create and consume many interrelated artifacts during the architecting process. These artifacts may represent functional and nonfunctional requirements, architectural patterns, infrastructure topology units, code, and deployment descriptors as well as architecturally significant design decisions. Design decisions have to be linked to chunks of architecture description in order to achieve a fine-grained control when a design is modified. Moreover, it is imperative to identify quickly the key decisions affected by a runtime change that are critical for a system's mission. This paper extends previous work on architectural knowledge with a metamodel for architectural decision capturing and sharing to: (i) create and maintain fine-grained dependency links between the entities during decision identification, making, and enforcement, (ii) keep track of the evolution of the decisions, and (iii) support runtime decisions.

Keywords: architectural design decisions, architectural knowledge, metamodel, runtime decisions, traceability, evolution.

1 Introduction

Existing software architecture design processes [1] lack adequate mechanisms to explain the line of reasoning that architects follow in order to make design decisions. Reasoning about the architectural design is considered a tacit process that exists only in the architect's mind; the decisions that lead to a software architecture are often overlooked during architecture design and thus not systematically documented. In recent years, the software architecture community has established design decisions as first-class entities that should be captured alongside with other design elements. Therefore, the creation of software architectures is now also seen as the result of a set

of design decisions rather than just as an assembly of components and connectors [2]. Making decisions explicit preserves architectural knowledge when staff is exchanged, e.g., when subject matter experts join the development team only temporarily or when transitioning from development to maintenance. As mentioned in [3], long-term benefits such as reduced maintenance effort should motivate users to capture the design rationale explicitly in the form of architectural decisions. This particularly holds true in successive iterations of the system as it evolves.
This paper extends previous work on architectural knowledge with a metamodel for architectural decisions to: (i) create and maintain fine-grained dependency links between the entities during decision identification, making, and enforcement, (ii) keep track of the evolution of the decisions, and (iii) support runtime decisions. Section 2 describes the background and the motivation of this research. In Section 3 we present a metamodel supporting traceability to keep track of the decisions made and their relations to design elements and artifacts. Section 4 then outlines the implementation of the metamodel in several prototype tools. Section 5 discusses a case study in the Service-Oriented Architecture (SOA) domain to demonstrate how the extensions of the metamodel are of practical use for SOA design. Section 6 describes the related work and section 7 summarizes the conclusions and future work.

2 Motivation and Problem Identification

A variety of research prototype tools have been developed to support design decisions in software architecture. From our experience developing and using various tools for architectural decision modeling, e.g., the Architectural Decision Knowledge Wiki [4], Architecture Design Decision Support System [5], and The Knowledge Architect [6], we observed three major shortcomings related to the creation and maintenance of the traceability links between the architectural knowledge and other artifacts:

1. The coarse *link granularity* in existing metamodels makes models easy to populate, but does not support a fine-grained tracing and tracking of decisions in relation to atomic design elements such as attributes in a class model or tasks in a business process model. Support for fine-grained trace links in current architectural decision modeling tools is weak or inexistent as some of the tools import UML design models externally and decisions can be only linked to coarse-grained artifacts.
2. Existing metamodels do not put special attention on *history and evolution of decisions*. Only a few of them treat evolution of decisions and architecture partially. One reason for this limitation is that most commercial and open source UML modeling tools do not offer explicit support for architecture evolution (e.g., Jude Community, Magicdraw).
3. The decision making process suggested by existing metamodels assumes that all decisions can be made at design time; *deferring decisions to runtime* is not supported. At present, the existing architecture decision modeling prototype tools do not offer support for runtime decisions that can be traced back to the architecture or to requirements when a piece of code or a system module change.

The first problem area addressed in this paper is *link granularity*. Links connecting key design decisions to architectural artifacts should include relationships to smaller parts of the design. Such an approach helps to achieve the precision required to estimate the impact of changes accurately. Small but important decisions should also be captured and linked properly. For instance, a decision to introduce a new UML package or class seemingly constitutes a more coarse-grained decision than the decision to add a new attribute to an existing class; however, the attribute may express a key architectural concern, e.g., it might flag an architecture component to be subject to financial and general IT controls audits or it might demarcate a system transaction boundary in a service composition. In many cases, fine-grained decisions are derived from coarse-grained ones made before; however, the lack of accuracy of existing traceability models do not offer a way to track the impact on the design or code. Thus, it is required to introduce trace links with narrower and more precise scope to achieve more precision in the traceability of architectural decisions during decision identification, making, and enforcement.

The second problem pertains to the maintenance of a system, as the design decisions made in the past might become obsolete, and the *history and evolution of decisions* should be recorded in the same way versioning repositories store the history and evolution of source code. This is useful for a number of reasons. In certain cases during system evolution, the architects have to revisit past decisions and revert to them if a new decision appears to be wrong. In other cases, architects may need to roll back the design, and start a new decision path from that point. Finally new stakeholders that become involved in a project can be educated much more efficiently by studying the evolution of decisions over time and the rationale that lead to the existing set of decisions and the present design.

As a third problem, we observed that today the dynamicity of certain systems may imply that certain decisions affect architectures that have already been deployed but have to be modified during runtime. For instance, a composite service which replaces an atomic service with another one due to new quality-of-service conditions during execution requires *deferring decisions to runtime*. Such deferred decisions have to be tracked back to the architecture and requirements so that conformance to them can be ensured. Supporting runtime decisions becomes increasingly relevant in modern operating environments and deployment infrastructures such as virtualized data centers: each instantiation of a virtual software image may decide for a slightly different set of quality properties. Examples include the heap and disk size of virtual UNIX machines (*infrastructure-as-a-service scenario*), Java and relational data source settings of Web application servers (*platform-as-a-service*), and login and encryption policies of hosted Web conferences (*software-as-a-service*). These decisions are based on user preferences and current resource consumption (system load); these two types of decision drivers only become known at runtime. Consequently, it makes sense to defer the detailed architectural decisions about these infrastructure settings to runtime (while at design time certain architectural templates that constrain the runtime configuration options can be predefined).

In our previous work [4, 7] we introduced a conceptual framework for decision modeling with reuse to extend recent research on design decisions. Our work focused on the following main contributions:

1. A **decision-making process** which comprises *decision identification* to delimit the scope, *decision making* to choose a feasible design alternative for each design issue, and *decision enforcement* to share the results of the decision making step with relevant stakeholders.
2. A **decision-capturing and sharing metamodel** supporting the decision making process. This metamodel is specified as a Unified Modeling Language (UML) class diagram and a formal definition based on elementary set and graph theory [4]. The metamodel, illustrated in Figure 1, relies on three main core domain entities: *ADIssue*, *ADAlternative*, and *ADOutcome* (*AD* stands for *Architectural Decision*). An *ADIssue* captures an architectural problem that requires a design solution whereas *ADAlternative* instances capture the pros and the cons of the design choices an architect has (i.e., the possible solutions available and the criteria for choosing or not choosing such option). Finally, *ADOutcome* instances capture project-specific knowledge including the justification and the consequences of decisions actually made. This metamodel is implemented in the Architectural Decision Knowledge Wiki/Architectural Decision Knowledge Web Tool, which is a collaboration system and decision modeling tool [4]. Other existing tools are based on similar metamodels [5], [6].

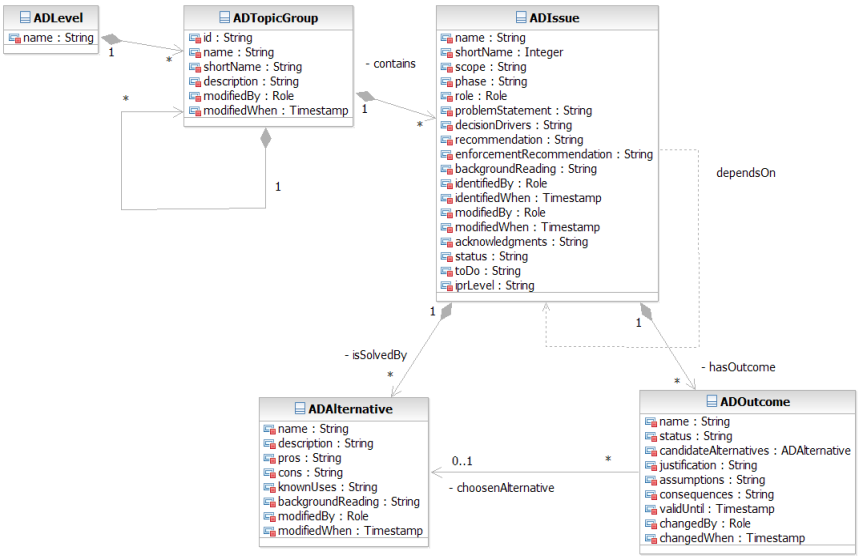

Fig. 1. Metamodel for architectural design decisions implemented in the Architectural Decision Knowledge Wiki tool.

With regards to the problems of link granularity, history and evolution of decisions and deferring decisions, the existing metamodel does not offer support. We will later explain how it can be extended to support these concepts. We worked with more than one hundred practicing architects, who applied and appreciated the metamodel as well as the SOA guidance model instantiated from it [4], [7]. As part of our validation activities, we conducted a user survey. Among other things, users pointed out:

1. Decisions have to be visited multiple times and sometimes revised as the design evolves; any waterfall process or big design upfront is not adequate for most real-world projects. Decisions are hardly made in isolation.
2. The lifetime of decisions transcends their identification, making, and enforcement; they have to be evaluated once a system is implemented, at least in prototypical form. Only then it becomes evident whether made decisions have led to a design and implementation that allows the system to meet the quality attributes that have been stated for it.
3. There is a desire to model links from decisions to other model elements and artifacts represented more explicitly (e.g., types of requirements appear as decision driver text in the metamodel in Figure 1, but are not first class metamodel entities that can be linked to). The scope attribute of an issue (in the metamodel in Figure 1) can identify the type of design model element an issue pertains to, but at present this textual information does not link to any artifacts used in the design process.

The metamodel extensions specified in this paper are motivated in this user feedback. We base our proposed metamodel extensions on the metamodel that underlies in Architectural Decision Knowledge Wiki/Architectural Decision Knowledge Web Tool because this tool is populated with a SOA guidance model comprising more than 500 issues and 2000 alternatives recurring in SOA design; architectural patterns described in the literature are among these alternatives (only a subset of these issues and alternative descriptions have been published so far). Hence, we count on a significant amount of knowledge to describe different types of design issues from a realistic point of view. However, our metamodel extensions are designed in such a way that they can be implemented in other tools as well (assuming that these tools support extensibility of their respective metamodels). To support this claim, we outline how we implemented the new concepts in an extensible commercial requirements engineering product later in this paper

3 Enhanced Trace Links and other Metamodel Extensions

To overcome the three problems mentioned before, we extended the conceptual metamodel of Figure 1. Our main rationale for adding new elements is to support explicit trace links to small architectural artifacts that help to check the integrity of the decision network, to evaluate the impact of changes, to keep track of the history and evolution of changes, and to record the root causes of changes. This new metamodel is shown in Figure 2. In the remainder of this section we describe the new classes and new elements highlighting them in italicized text.

Links to Design Artifacts: Two new classes, *ADDesignElement* and *ADDesignArtifact*, specify the parts of the architecture that result from one or more design decisions represented by outcome instances. *ADDesignElement* instances represent elements of modeling languages. For example, if we map to Unified Modeling Language (UML), it refers to a UMLNamedElement (i.e., any UML element that can be named). This includes coarse grained elements such as

components and connectors, but also more fine grained elements such as class attributes. *ADDesignArtifact* aggregates and assembles such elements into project deliverables such as a platform-independent, technology-neutral functional component model. *ADDesignElement* instances are defined to have an *ADDesignElementType*, which also becomes the type of the scope attribute of the *ADIssue* class. In the architectural decisions viewpoint, the relationships between two newly introduced subclasses of *ADOutcome*, *ADDecidedOutcome* and *ADDeferredOutcome* (the existing metamodel introduced the *ADOutcome* class to record actual decisions made to solve a problem including its rationale), and *ADDesignElement* (with subclass *ADRuntimeElement*, introduced below) allows us to define trace links to individual parts of an architecture. *ADDecidedOutcome* and *ADDeferredOutcome* indicate that enforcing a decision at design time differs from enforcing a decision at runtime (with respect to the artifacts in which the decision materializes; e.g., UML class or conceptual application server node at design time vs. Java class or XML deployment descriptor at runtime). Such fine-grained linkage down to the level of individual architectural elements (e.g., UML components and connectors, physical topology units and hosting links, attributes of UML components or Java classes or XML elements) increases the precision and expressivity of the decision models. In summary, we have now introduced external links from decisions to structural and behavioral models, which were not supported previously.

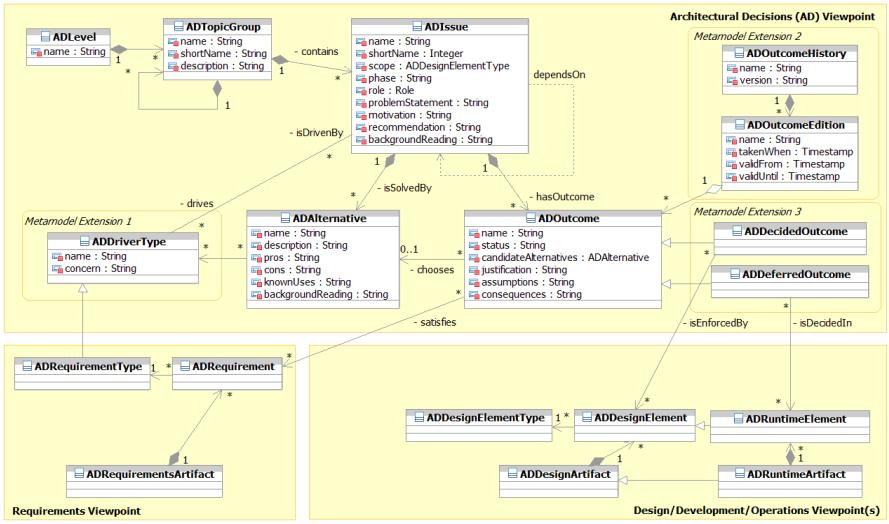

Fig. 2. UML metamodel for capturing design decisions with focus on maintenance, evolution, and runtime concerns.

In the decision making process, several alternatives (*ADAlternative*) can be captured, considered, and evaluated before a decision is made. An external link, from requirements to decisions, can be established via the new class *ADDriverType*, which gathers the origins and influencers of decisions, such as types of functional and non-functional requirements. Because an issue is a reusable knowledge entity, the *ADDriverType* class supports only types of requirements (e.g., quality attributes such as performance and modifiability), but not real instances of such requirements: the

additional class *ADRequirement* serves this purpose. *ADRequirement* instances may represent analysis artifacts such as business process models, use cases, or user stories as well as non-functional requirements such as software quality attributes (e.g., sub-second response time performance, modifiability via multi-platform support, etc.). *ADRequirementsArtifact* instances compile a number of individual requirements. Each *ADRequirement* instance is classified by its kind, which is expressed by the *ADRequirementType* class. As a result of the improvement, we removed the decisionDrivers attribute initially defined in the *ADIssue* class (e.g., a problem that has to be solved). Thus, the new metamodel supports now full traceability from requirements to decisions and other design artifacts.

Decision History and Evolution: The evolution of decisions is described by means of the *ADOutcomeEdition* class, which establishes a chain of decisions that change over time. For instance, a corporate system may have to replace its middleware after several years of successful production use because new enterprise-level requirements demand a technological change in the organization. Hence, this decision made in the past for selecting the right middleware may have became obsolete and may have to be replaced by a new one. The *ADOutcomeHistory* class keeps track of the history of changes to a decision made years or months ago (i.e., collections of related *ADOutcomeEdition* instances, each of which referring to a single *ADOutcome instance*).

Support for Runtime Decisions: Some systems may change their status, operation mode (e.g., a system that updates its software version changes its operation mode from normal operation to maintenance mode until the reconfiguration process finishes and the system returns to the normal mode), or configuration during runtime due to external or internal conditions. Hence, the decisions that led to, for instance, a given product architecture might have to be modified, and in some cases lead to a different architecture. In such cases, certain decisions have to be replaced temporarily by new ones or they can also become obsolete for a given time period. Therefore, we introduce the *ADRuntimeElement* class (atomic) and the *ADRuntimeArtifact* class (composite) to reflect such situations and represent the code pieces that enforce the decisions represented by instances of the *ADDeferredOutcome* class. As decisions that change during runtime cause the architecture to be modified according to the depth of the change, adding support for runtime decisions improves traceability between artifacts; runtime artifacts can serve as link targets. These finer grained traceability links can determine the parts of architectures that have to be modified when changes happen. To our knowledge, this feature has not been implemented before in other tools and models capturing design rationale. Hence, we extend and enhance previous works for systems that require more surveillance or adaptability due to, for instance, new context conditions. Examples of issues that cannot always fully be resolved at design time are:

- Specifically to Service-Oriented Architecture (SOA), capturing runtime decisions and linking these to code assets is required. For example, our metamodel can describe the decision in a composite Web Service (a type of design element) to dynamically modify the Business Process Execution Language (BPEL) workflow that realizes the composite Web service, e.g., to engage a new subprocess to

reflect a certain business rule or other runtime condition. Such late decision is often based on new quality-of-service conditions that modify the Service Level Agreement (SLA) for a given period (e.g., regarding guaranteed response times). Our metamodel uses the classes *ADRuntimeArtifact* and *ADDeferredOutcome* to express such situations.

- The decision how to route a service invocation request that represents an atomic activity in an executable business process model (i.e., *dynamic service composition*). Note that this decision can only be deferred to runtime if such flexibility does not violate regulatory constraints such data privacy and system and process assurance compliance (such concerns can be modeled as AD*DriverType* and linked to issues according to the metamodel presented in Figure 2).
- The decisions enable to customize certain software features when reusing a particular application package, middleware component, or product family (e.g., using variation points in software product lines [8], [9]). For instance, a database management system might support distributed two-phase commit (2PC) protocol at an extra performance and license cost; when the decision to use the system is made, it might not be known yet whether the 2PC support is required. This decision might even change over time, which can be expressed as a series of chained AD*OutcomeEdition* instances.
- The decision to delegate some of the responsibilities to end users that are performed by architects/developers in traditional software engineering (*situational application* development via Web-centric container architectures such as mashups). For instance, such design issues might deal with user interface patterns, data formats (e.g., MIME types), and information provider selections.

4 Implementation in Existing and Emerging Tools

This section outlines how the enhancements in the extended metamodel can be supported by three existing architectural knowledge management and modeling tools: ADDSS [5], The Knowledge Architect [6], and Architectural Decision Knowledge Wiki/Web Tool [4]. These tools share several goals and usage scenarios, but differ in their origins, use cases, and tool architecture. We discuss all three independently developed tools to illustrate the generality of our approach by explaining how the extended metamodel can be supported by them. In addition, we present an actual implementation of the extended metamodel on top of a commercial requirements engineering and management platform which supports metamodel extensions and Web-based artifact linking.

4.1 Architecture Design Decision Support System (ADDSS)

In this tool [5], the model underlying the tool supports explicit traces to requirements (*ADDriverType*) and architectures (*ADDesignElement*, *ADDesignArtifact*) as well as between design decisions, but links between decisions and smaller parts of the architecture can not be specified in a fine grained fashion. To overcome this, Figure 2

specifies a class *ADDesignElement* and establishes links from the *ADOutcome* to provide fine grained links to small design artifacts. Evolution in ADDSS is only supported by several attributes; there is no way to define a chain of decisions history as in the proposed metamodel of Figure 2 (using the ADOutcomeEdition and *ADOutcomeHistory* classes). Finally, ADDSS does not support runtime decisions like in our proposed solution. Hence, the *ADRuntimeElement*, *ADRuntimeArtifact* and *ADDeferrredOutcome* classes should be incorporated into ADDSS' metamodel to enable tracking runtime decisions.

4.2 The Knowledge Architect (KA)

This tool suite [6], [10] is comprised of a number of specialized tools for capturing, (re)using, translating, sharing, and managing software architectural knowledge. The Knowledge Architect entails specialized support for integrating the various architecting activities [11] and supporting collaboration between the stakeholders of these activities. The different tools support different activities (e.g. analysis, design, sharing) and therefore each tool has a specialized Architectural Knowledge (AK) metamodel to deal with the different types of knowledge produced and consumed during the architecting process. The different metamodels are integrated into the central knowledge repository of the tool suite. Traceability can be achieved in two ways: a) within each metamodel, traceability links are established between the AK concepts (e.g., between "decisions", "concerns", "decisions topics" and "alternatives" in the document knowledge client of the KA) b) across different metamodels traceability links can be established within the knowledge repository (e.g. "decisions" and "concerns" are common concepts of both the document knowledge client and the analysis model knowledge client of the KA). The KA can be extended in two ways to support the metamodel of Figure 2: a) all the tools have extensible metamodels (not hard-coded but completely customizable), thus the new concepts and relations can be added in a straightforward way; b) the central knowledge repository itself stores knowledge in Resource Description Framework (RDF) format and can directly accommodate the metamodel extensions of Figure 2. As an example the classes *ADDecideOutcome* and *ADDeferredOutcome* can simply inherit from the class *Decision*, while *ADDriverType* can inherit from the class *Concern* (both Decision and Concern belong to the document knowledge client metamodel). The extensions for history and evolution are not necessary to be implemented as the KA, as the tool suite uses the versioning system of Sesame to track the evolution of each knowledge entity.

4.3 Architectural Decision Knowledge Wiki/Architectural Decision Knowledge Web Tool

Architectural Decision Knowledge Wiki is a Web 2.0 collaboration tool supporting the decision modeling capabilities and original UML metamodel first published in [7]. A version 1.0 was originally implemented in PHP and released in March 2009; in October 2009, a Java reimplementation of the tool was released under the name Architectural Decision Knowledge Web Tool [4]. The tool supports about 50 decision modeling and making use cases. It assembles *ADIssue* and their *ADAlternative* on a

decision identification tab (these metamodel entity types are jointly referred to as *decisions required*). *ADOutcome* instances are created and updated on a second *decision outcome* tab (capturing *decisions made*), which exposes a simple decision state management workflow to the user (with open/decided/approved/rejected states). To support the extended metamodel introduced in the previous sections, the following additional features and components are required:

1. The *ADDriverType* class is a result of refactoring the decision driver attribute in *ADIssue*; hence, the new capability can be implemented by *refactoring* the user interface components displaying the decision identification tab as well as the underlying server-side business logic and database schema. Having performed these refactorings, the fine-grained traceability links can be added to the decision identification tab; advanced user interface features such as pop-ups can be added.
2. The *ADOutcomeHistory* and *ADOutcomeEdition* classes can be realized by implementing the *edition pattern*. The business logic and the database schema of the existing implementation already do so; on top of that, an additional *decision evolution* tab can be added to the user interface to display the decision making history.
3. Deferring decisions to runtime can be supported by introducing a *new state* "deferred" for outcome instances; this requires to update the user interface components supporting the decision making tab, as well as the state machine implemented in the business logic realizing *ADOutcome* instance creation and lifecycle management.

4.4 Implementation in IBM Rational Requirements Composer

To investigate and demonstrate the technical feasibility, practicality, and usability of these enhancements, we created a demonstrator in a requirements modeling and management platform prior to implementing them in the actual tools (following the well-established design principles such as user interface storyboarding and prototyping).

For our proof-of-concept we used a recently released requirements engineering and storyboarding tool, *IBM Rational Requirements Composer (RRC)*. Version 2.0 of this Jazz repository-based product became generally available on jazz.net in November 2009. The RRC metamodel by default supports artifacts such as business process models, use case diagrams, storyboards, but also supplemental rich text documents representing features and non-functional requirements. All artifacts as well as external resources can be linked to each other via Web URLs. Via attribute groups, the default metamodel can be extended.

We first created custom attribute groups to represent the original metamodel and then added new attribute groups representing *ADDriverType* and *ADDeferredOutcome*. *ADOutcomeHistory* does not require product configuration; it is supported by the server component of the RRC product (via the snapshotting capabilities which stores model versions in the Jazz repository). Next, we instantiated SOA model elements (instances) via templates we created from sample rich text artifacts which use the newly defined attribute groups. The sample model elements

were populated from the existing SOA guidance model available in Architectural Decision Knowledge Web Tool (via copy-paste). Finally, fine grained traceability links were added to demonstrate requirements to decisions linkage.

The sample links from requirements to issues and back (introduced in the previous section and shown in the extended UML diagram in Figure 2) demonstrate the technical feasibility of our concepts; the links reside on the individual requirement/issue/outcome instance level, not on document-to-document level. This paves the way for requirements to decisions integration as suggested by our metamodel extensions. Concerns expressed as *ADDriverType* become first class citizens in the user interface (tagged as architecturally significant requirements) and the architecture of the tool (unlike in the original implementations). In conclusion, this implementation demonstrated that the extended metamodel is generic and expressive enough to be supported in multiple tools.

5 Instantiation for SOA Enterprise Applications

We applied our extended metamodel to an industrial case study from the telecommunications industry. This industrial case study concerns the modernization of an existing, business-to-business *order management system* (OM) in a major telecommunications company employing a wholesaler-retailer business model [12]. In this business process-centric scenario, a key business requirement (concern) was to ensure enterprise resource integrity over multiple channel interactions and time. User channels included the Internet (providing end user self services) and call centers. Two of the order management processes consisted of up to 19 steps and could run for up to 24 hours. Market deregulation and increasing competition caused the concrete problem of having to coordinate competing requests for the same physical resources in the shared telephony network. This coordination was seen to improve customer satisfaction (measured as number of successful order requests).

This business environment led to many architectural design challenges. Key technical requirements in this order management context were multi-channel request coordination and process instance and timeout management. A business transaction started via the Internet-based self-service channel had to be able to continue via call center (back office) interaction. Different VSP retailers reserved resources in a single network owned by the wholesaler, so incomplete requests had to be undone after a certain amount of time. The system context and resource integrity management requirement suggested introducing a process layer as a governing architecture element. This process layers serves one user channel per user type. These channels reside in the presentation layer of the order management system. The required long-running process instance tracking and timeout management could be implemented in a macroflow engine [13] dedicated for this task (called). Short-running, transactional flows could be handled by dedicated microflow engines [13].

All these concerns are addressed in the logical architecture of the production solution which is outlined in Figure 3 and explained in detail in [12]. While such UML class diagram can give an architectural overview, many detailed concerns cannot be covered on this level of refinement. For instance, many technology- and product-specific design issues and the rationale of the decision outcomes should be

explained in detail elsewhere. More specifically (in the context of this paper and the proposed metamodel extensions), the architecture elements should be traced back to the outlined requirements, the evolution of the system from a plain Java Web application to a process-based SOA should be captured, and the necessity to defer certain decisions to runtime should be captured.

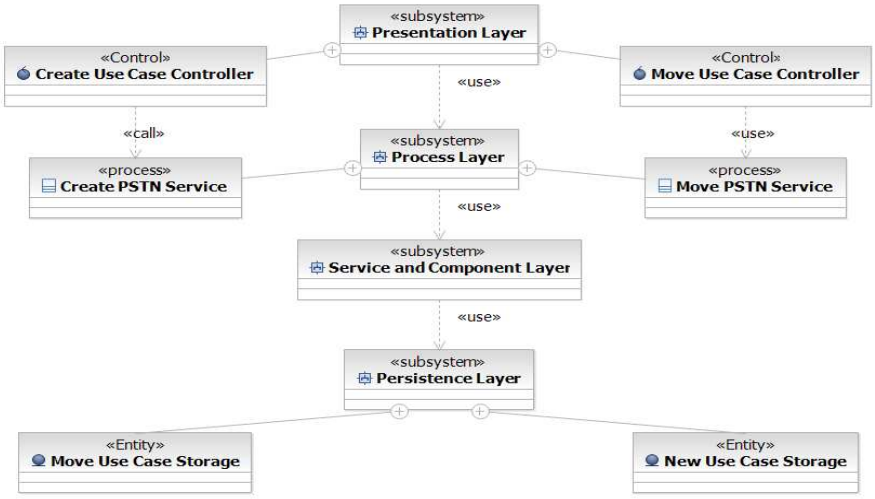

Fig. 3. Functional components of the order management system

Let us map the model elements in Figure 3 back to the metamodel from Figure 2. All UML classes representing functional components are instances of ADDesignElement (irrespective of their stereotypes); the class diagram itself is an instance of ADDesignArtifact. The ADDesignArtifactType of this class diagram artifact is "functional component model"; the ADDesignElementType of the ADDesignElement instances is "(functional) UML component" (we can view component stereotypes such as "subsystem", "control component", and "process component" as subtypes; however, this subtyping is not expressed by our metamodel). Example of traceability links will be given in the next subsection and Figure 5. We uses the extended metamodel of Figure 2 to illustrate how these design/modeling problems in the Order Management (OM) case study can be modeled.

Early in the project, a decision was required to decide for the main architectural concepts. In particular, a process-based SOA and the related architectural patterns were chosen because the solution was supposed to be flexible and adaptable. One of the important conceptual decisions in this context was to decide whether a service composition layer should be introduced into the architecture (the outcome of this decision led to the inclusion of the Process Layer component in Figure 5).

Figure 4 shows a (heavily simplified) instance of the metamodel for this decision, working with a subset of the design elements from Figure 3. Both instances of the core classes of the existing metamodel (*ADIssue*, *ADAlternative*, *ADOutcome*) and our metamodel extensions are illustrated (*ADRequirement*, *ADDesignElement*, *ADOutcomeHistory*, etc.). A sample decision <<ADReqType>> Portability and a

concrete <<ADRequirement>> Runs on 2 Platforms (i.e., solution can on at least two platforms) were identified for one required and made decision (<<ADIssue>> Workflow Language with selected <<ADAlternative>> BPEL).

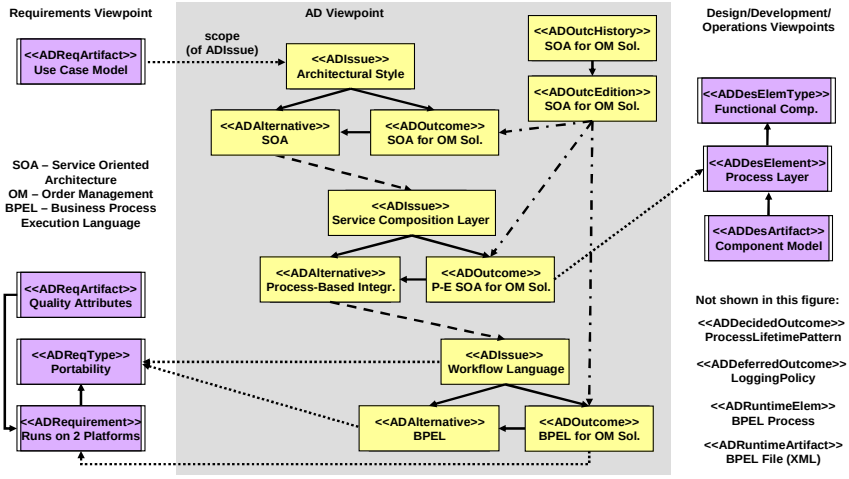

Fig. 4. Architectural decisions made in case study with links to design model context a.k.a. exemplary application (instantiation) of the AD metamodel for the case study

At this stage, we couldn't test the evolution of the decisions as we only produced the first version of the architecture of the OM system using, but we captured the evolution of the system from a plain Java Web application to a process-based SOA.

Furthermore, decisions that might change at runtime can be tracked using the proposed metamodel extension (i.e.: AD*RuntimeArtifacts*, AD*RuntimeElements*) and the class that enforces the decisions (AD*DeferredOutcome*). In the order management SOA, the system transaction boundary and the logging settings might differ for certain components in the process layer and for components in the service layer shown in Figure 3. This metamodel extension is not illustrated in Figure 4.

6 Related Work

Several research prototype tools [11], [14] for capturing, using, and documenting architectural design decisions have recently appeared; many of these use templates and metamodels for capturing knowledge attributes and managing decision dependencies [15], [16]. Tools such as PAKME, ADDSS, Archium, The Knowledge Architect, and AREL offer traceability mechanisms between decisions and other software artifacts at different levels. Some of these tools support the evolution of trace links between decisions and forward and backward traces. The traceability supported by the tools can be used to estimate those artifacts that are impacted by the change in a decision, as the majority of the mentioned tools lack fine grained links between decisions and small architectural artifacts (e.g., a UML class or component instead of

an entire subsystem). In addition, the approach presented in [17] highlights the role of traceability in software architecture evolution and describe a method to manage such traceability for design decisions using a model-driven development approach.

Software product lines (SPL) need to model also the dependencies of feature models (i.e.: in practice they constitute a decision model) for different phases of the software life-cycle. Modeling dependencies and dealing with traceability problems in SPL is discussed in [18], where a wide list of dependency types between features are defined as constraints a software product must satisfy, while in [19] the authors explain how metamodels from PAKME and ADDSS tools can be merged to support product lines concepts and model dependency links between architectural design decisions and the variability rules associated to a feature model. Other works refer to Dynamic Software Product Lines (DSPLs) [20] to provide the necessary binding for runtime variation points to adapt the software to changes in the environment. The authors state that it is impossible to foresee al the variability a SPL requires, and use dynamic architectures and support for runtime decisions to be able to support system configuration and binding at runtime (for automatic decision-making). Designing and managing runtime variation points in architecture is also described in [21], where patterns are used to provide such facility in SPL and add the necessary flexibility for domain-specific applications (e.g.. custom Web servers that cannot be stopped when deploying or configuring components).

Lago et al. [22] discuss three different traceability issues during SPL derivation, and they focus on those traceability links between feature models and structural models (i.e.: architecture-level decisions). In [23], a Dependency Structure Matrix (DSM) is used to represent and manage dependencies in complex software architecture and to reveal underlying architectural patterns. Acceptable and unacceptable dependencies are expressed using design rules to describe the semantics of such dependencies.

All the aforementioned approaches lack explicit support for runtime decisions that can be deferred and tracked back from code to the architecture and to the design decision. Furthermore, in most cases they support coarse grained links between decisions and other software artifacts. Evolution is only partially supported in two existing tool prototypes. Hence, our approach improves these features and enriches previous metamodels and tools with runtime decisions. Other approaches that consider fine grained traceability paths between different artifacts do not consider the inclusion of design decisions as we do.

Traceability between decisions and from decisions to artifacts is related to traceability between requirements and model elements in general. This general problem of establishing and maintaining traceability has been studied in the literature and different approaches exist. Maeder et al. [24] present an approach for automating traceability maintenance under changes by classifying changes and automating updates of the traceability graph. Such an approach could in principle also be applied to traceability management for architectural decisions. Cleland-Huang and Chang [25] propose a traceability method that is based on the publish-subscribe architecture in order to keep traceability links up to date. It remains for future work to investigate the best approach to maintain traceability links between architectural decisions and requirements.

7 Conclusion and Future Work

Our approach revisits and enhances previous models and tools as we provide full traceability between individual decisions and other software artifacts using fine grained links, even if the decision networks becomes more complex to manage and to maintain. We are aware that capturing fine grain trace links introduces additional costs to maintain the links over time and this cost should not be higher than the expected benefits, but the architect must decide when to define such links to smaller parts of the architecture that must be traced (e.g., a critical software component in a system composed by a few number of classes is replaced at runtime by a new component with extended functionality and defined by a new UML for which a new trace link must be create for its corresponding design decision). With such links we achieve a better control of individual decisions and we are able to find out in detail which parts of the architecture are affected by a change in the requirements or code. Because certain software systems may vary their context conditions during runtime, they require adequate models to support runtime decisions that can be deferred. Hence, we extend previous works to track runtime decisions and make software architects aware of changes that may affect the design. Other extensions would include supporting the full context of decisions that evolve and store not only the decisions but also the issues, drivers, and requirements that accomplish a particular solution. Finally, other non-SOA domains like self-adaptive systems can benefit from tracking runtime decisions as a way to monitor better those changes that happen during system execution.

References

1. Bass, L., Clements, P., Kazman R.: *Software Architecture in Practice*, Second Edition, Addison Wesley, (2003)
2. Bosch, J.: Software Architecture: The Next Step, Proceedings of the *1st European Workshop on Software Architecture* (EWSA 2004), Springer-Verlag, LNCS 3047, pp. 194-199, (2004)
3. Kruchten, P., Capilla, R., Dueñas, J.C.: The Decision's View Role in Software Architecture Practice. IEEE Software, vol 26(2), 36-42, (2009)
4. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. Journal of Systems and Software volume 82(8), 1249-1267, (2009)
5. Capilla, R., Nava, F., Pérez, S., Dueñas, J.C.: Web-based Tool for Managing Architectural Design Decisions (SHARK'066), ACM SIGDOFT Software Engineering Notes 31(5), (2006)
6. Jansen, A., Vries, T.d.., Avgeriou, P., Veelen, M.v.: Sharing the Architectural Knowledge of Quantitative Analysis, *Proceedings of the Quality of Software-Architectures* (QoSA), 220-234, (2008)
7. Zimmermann, O., Gschwind, T., Küster, J.M., Leymann, F., Schuster, N.: Reusable Architectural Decision Models for Enterprise Application Development. (QoSA'07), Springer-Verlag LNCS 4880, 15-32, (2007)
8. Bosch, J.: Design and use of Software Architecture: Adopting and Evolving a Product-Line Approach, Addison-Wesley, (2000)
9. Pohl, K., Böckle, G., Linden, F.v.d: Software Product Line Engineering: Foundations, Principles, and Techniques, Springer, (2005)

10. Liang, P., Jansen, A., Avgeriou, P.: Collaborative Software Architecting through Architectural Knowledge Sharing, in A. Finkelstein, J. Grundy, A. van der Hoek, I. Mistrík, J. Whitehead (eds.) Collaborative Software Engineering (CoSE), pp. 343-368, Springer-Verlag, (2010)
11. Liang, P., Avgeriou, P.: Tools and Technologies for Architecture Knowledge Management. In Software Architecture Knowledge Management: Theory and Practice, 91–111. Springer, (2009)
12. Zimmermann, O., Doubrovski, V., Grundler, J., Hogg, K.: Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), ACM Press, (2005)
13. Hentrich, C., Zdun, U.: Patterns for Process-Oriented Integration in Service-Oriented Architectures. In Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006), Irsee, Germany, pages 1-45, July, (2006)
14. Tang, A., Avgeriou, P., Jansen, A., Capilla, R.., Babar, M.A.: A Comparative Study of Architecture Knowledge Management Tools, Journal of Systems and Software 83(3), 352-370, Elsevier, (2010)
15. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture, IEEE Software, vol. 22, (2005)
16. Kruchten, P., Lago, P., Vliet, H. van: Building up and reasoning about architectural knowledge. In: Hofmeister, C. (Ed.), Proceedings of Second International Conference on the Quality of Software Architectures (QoSA 2006), Springer LNCS 4214, (2006)
17. Navarro, E., Cuesta, C.E.: Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach. European Worksop on Software Architecture, Springer-Verlag LNCS 5292, 114-130, (2008)
18. Lee, K., Kang, K.C.: Feature Dependency Analysis for Product Line Component Design, International Conference on Software Reuse, LNCS 3107 Springer-Verlag, pp. 69-85, (2004)
19. Capilla, R., Babar, M.A.: On the Role of Architectural Design Decisions in Software Product Line Engineering. (ECSA'08), Springer-Verlag LNCS 5292, 241-255, (2008)
20. Hallsteeinsen, S., Hinchey, M., Park S., Schmid, K.: Dynamic Software Product Lines, IEEE Computer 41(4), 93-95, (2008)
21. Goedicke, M., Köllmann, C., Zdun, U.: Designing Runtime Variation Points in Product Line Architectures: three cases. Science of Computer Programming 53(3), 353-380, (2004)
22. Lago, P., Muccini, H., Vliet, H. van: A scoped approach to traceability management. Journal of Systems and Software, 82(1), 168-182, (2009)
23. Sangal, N., Jordan, E. Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. OOPSLA'05, 167-176, (2005)
24. Mäder, P., Gotel, O, Philippow, I.: Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations. In Proceedings ECMFA-FA 2009, LNCS 5562, 174-189, Springer, (2009)
25. Cleland-Huang, J., Chang, C.: Event-Based Traceability for Managing Evolutionary Change. IEEE Transactions on Software Engineering, Vol. 29, No. 9, September (2003)