Supporting Architectural Design Decisions Evolution through Model Driven Engineering

Ivano Malavolta¹, Henry Muccini¹, V. Smrithi Rekha^{1,2}

¹ University of L'Aquila, Dipartimento di Informatica, Italy

² Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham, India
ivano.malavolta@univaq.it, henry.muccini@univaq.it,
smrithirekha@gmail.com

Abstract. Architectural design decisions (i.e., those decisions made when architecting software systems) are considered an essential piece of knowledge to be carefully documented and maintained. As any other artifact, architectural design decisions may evolve, having an impact on other design decisions, or on related artifacts (like requirements and architectural elements). It is therefore important to document and analyze the impact of an evolving decision on other related decisions or artifacts. In this work we propose an approach based on a notation-independent metamodel that becomes a means for systematically defining traceability links, enabling inter-decision and extra-decision evolution impact analysis. The purpose of such an analysis is to check the presence of inconsistencies that may occur during evolution. An Eclipse plugin has been realized to implement the approach.

1 Introduction

A new perspective on Software Architecture (SA) views an SA as a composition of a set of design decisions [1], taken to resolve architectural design issues. In this light, and as currently recognized by software architects, architectural design decisions (ADDs) have to be considered first class entities when architecting software systems [2, 1, 3]. Therefore, they have to be explicitly documented and maintained. Since ADDs strongly depend on elicited requirements and highly contribute to shape the architecture of complex software systems, it is highly recommended to support ADDs evolution and to explicitly document and analyze the impact of design decisions' changes to other decisions and to related artifacts [4–6]. While a few approaches and tools have been proposed for documenting and managing ADDs evolution, they are typically tied to some specific architecture description language or ADD notation, and they only partially document the impact of a decision change to other decisions and related artifacts.

In this paper we propose an approach to support ADD evolution. It provides a generic, ADD notation-independent approach that starting from a model based specification of ADDs, enables to identify the impact of a changing design decision on other ADDs (referred as *inter-decisions* analysis) and the impact of the same on requirements and architectural descriptions (named *extra-decisions* analysis).

Our contribution is mainly in the following lines: *a*) We have proposed a meta-model for evolving ADDs. The metamodel is generic and flexible enough to manage

requirements and architectures described in any notation; *b*) the proposed metamodel enables explicit representation of relationships among ADDs *c*) the metamodel enables bi-directional traceability links between ADDs, requirements and architectural elements which will help in analyzing the impact of evolution on the corresponding artifacts; *d*) we present an approach that enables inter-decision and extra-decision validation to check the presence of inconsistencies that may occur during evolution.

The structure of the paper is as follows. Section 2 provides a background on SA evolution and highlights the importance of explicitly supporting ADD evolution and motivates our work. Section 3 describes our approach. Section 4 presents a prototype implementation of what is presented before. Section 5 summarizes related works, while Section 6 concludes the paper.

2 Background and Motivation

All software systems are subject to constant evolution, which is often triggered by external stimuli. As a result, all related software artifacts must evolve to handle foreseen or unforeseen changes [7]. By looking for example at the Rational Unified Process (RUP) [8], we observe that the architecting process is spread over several iterations and each iteration results in an *executable architectural prototype*. This results in an evolving architecture, being refined through many incremental iterations. Many times, requirements and architectural design co-evolve owing to exploration, negotiation and decision-making [9]. Since ADDs are related to each other in different ways and are related to other artifacts (e.g., ADDs directly trace to requirements *upstream* and to design and implementation *downstream* [2]), such an incremental refinement process requires to constantly add or modify decisions, that will in turn impact other decisions, requirements and architectural components. Practically, architects face difficulties when managing evolution: if the relationships among different decisions and the mapping between ADDs, requirements and architectural artifacts are not clear, changes to one decision may conflict with other decisions, bringing the system to an inconsistent state.

In order to partly cope up with such scenarios, several tools and approaches are available to capture ADDs and rationale (Archium [1] and ADDSS [10], for example). Most currently available tools capture the following concepts: *issues* or *problems*, *alternatives* or *options*, *assumptions*, *constraints*, *design decisions*, *rationale*, and relationships among ADDs. Though these tools and methods effectively capture architectural knowledge, very few tools have been designed to account for evolution of architectural decisions. The lack of support or availability of only minimal support for evolution by current tools and Architecture Description Languages has been discussed in [6, 7, 11].

3 Supporting Design Decisions Evolution

This section presents our approach to support design decisions evolution analysis through bidirectional traceability between decisions and all the involved artifacts.

In this respect, it is fundamental to analyze how design decisions evolve, and the actual impact of a design decision evolution on other related artifacts like requirements specifications and software architecture descriptions.

Few factors that may impact the evolution of ADD include:

1. Natural evolution that happens when the system is being developed incrementally
2. Evolution of requirements during the development of the system
3. Once the system has been deployed, Addition/Deletion/Changes to existing:
 - (a) Requirements
 - (b) Issues
 - (c) Alternatives (e.g., technology alternatives) for an issue under consideration
 - (d) Stakeholder concerns

These stimuli may directly impact the ADD model as well as its related artifacts. Figure 1 gives an overview of the approach we propose to support architectural design decisions evolution.

Fig. 1. Overview of the proposed approach

The **ADD model** is the most important artifact since it represents all the design decisions identified during the design process; it also classifies design decisions depending on their type and their status, and keeps track of the various relationships between them (Section 3.1 gives an overview on how to specify design decisions). Design decisions are linked to (i) **Requirements specification** and (ii) a set of **SA descriptions** resulting from the chosen design decisions (only one SA description is chosen, all the others are alternatives).

Traceability is realized as a set of tracing links from requirements to ADDs and from ADDs to SA descriptions (see the dotted lines in Figure 1). Tracing links are contained into special models (**wmreq** and **wmSA** in Figure 1) called *weaving models*; they will be described in detail in Section 3.2. A peculiar aspect of our approach is that tracing links are represented as models; this aspect is perfectly in line with MDE and represents an interesting added value for our approach. In particular, the use of weaving

models allows designers to keep the requirements and architectural models free from any traceability metadata. This makes the proposed approach **ADL (and requirements notation) independent**, allowing designers to use their preferred notation to describe the system according to their specific concerns. Moreover, the use of weaving models opens up for an **accurate evolution impact analysis** since they provide traceability information both at the model-level and at the model-element level. Intuitively, a designer not only can specify that a design decision dd_x pertains to a whole architectural model am , but also that dd_x pertains only to components c_1 and c_2 in am .

In this work we consider *evolution* to be a set of incremental changes of a model in response to changing external factors [12], as it is considered in the model management research area. Clearly, changes in an ADD model element can have an impact either on elements (e.g., ADDs conflicting with a deleted decision become valid alternatives), or also to external related artifacts (e.g., a component realizing a rejected decision must be checked and probably modified). Based on this, supposing that the current ADD model has evolved (see the left-lower part of the ADD model in Figure 1), the problem of analysing the impact of such an evolution can be divided into three main steps (they are also shown in Figure 1):

1. *evolution identification*: all the evolved ADD elements (the highlighted part of the ADD model in Figure 1) must be timely identified and must be presented to the designer;
2. *inter-decisions analysis*: the portion of ADD model which depends on the initial evolved ADD elements identified and their consistency must be checked;
3. *extra-decisions analysis*: all requirements and architectural elements depending on the evolved ADD elements are identified and checked against consistency.

It is important to check the ADD model and its related artifacts after evolution since evolution may introduce contradictions between ADDs (or partially defined elements, and so on) that must be carefully checked by the tool and solved by the designer (this aspect will be clarified in section 3.3).

Next sections describe in detail how our proposed approach allows designers (i) to clearly specify design decisions and their relationships between each other (see Section 3.1), (ii) to trace design decisions towards other system life-cycle artifacts (see Section 3.2), and (iii) to estimate the impact of design decisions evolution (see Section 3.3).

3.1 ADDMM: a Metamodel for Supporting Design Decisions Evolution

In the proposed approach design decisions are represented within a Design Decision(DD) model. A DD model represents all the design decisions and keeps track of all the design reasoning history of the system. For the sake of generality, instead of being coupled to a specific notation for ADDs (and its corresponding tool), we propose *ADDMM*: a notation- and tool-independent metamodel for representing design decisions and their evolution.

The *ADDMM* metamodel has been defined by analyzing the current state of the art ADD representation ([2], [13], [14] to include a few). We designed *ADDMM* so that it (i) presents the most common aspects and relationships of design decisions, and (ii)

contains concepts to define the evolution of an ADD (like its state, history and scope). It is important to note that, even if we customized the proposed approach to *ADDMM*, it is possible to adapt the proposed approach to work with other metamodels for ADDs (the visualization and traceability parts of the approach remain unchanged, whereas the evolution analysis part must be adapted to the new metamodel since it directly refers to elements in *ADDMM*).

Fig. 2. ADDMM: a metamodel for evolving design decisions

Figure 2 graphically shows the metamodel for design decisions models. Naturally, *DesignDecision* is the the main concept of the metamodel. It contains attributes like *description*, *state* that represents the current state of the decision (e.g., idea, tentative, decided, rejected, etc.), *timestamp* that specifies when the design decision has been created, *history* that keeps track of the evolution history of the decision (mainly, author, timestamp and status of each past version of the decision, and so on); it should be noted that the *history* attribute is conceived as a means for software architects to quickly access information about the evolution of a design decision over time, other mechanisms are used to manage different (evolved) versions of a design decision model (see Section 3.3). The *evolved* attribute is used as a marker to identify which design decisions have been subject to an evolution, it will be used during the evolution impact analysis, which is in charge of managing traceability between evolutions of architectural design decisions models (see Section 3.3).

Each design decision can be related to some other design decisions. Examples of relationships include: *constrains*, *enables*, *conflictsWith*, and so on (please refer to [2] for a comprehensive description of all the types of relationships). The *comprises* relationship exists when an high-level decision is composed of a set of more specific decisions; differently from all the other relationships (which are represented as simple reference), it is represented as a composition reference because the lifetime of comprised decisions must correspond to the lifetime of the comprising decision.

The *ADMM* metamodel distinguishes between two types of design decision:

1. *InternalDecision*: the design decision has a direct impact only on other design decisions; it is not related to any design artifact outside the ADD model. It can affect business and methodological parts of the system, like development process, organization issues, and so on.
2. *ExternalDecision*: the design decision has a direct impact to elements in other related artifacts (like a requirement, a set of design components, and so on).

The distinction between internal and external design decisions is useful during evolution impact analysis since it allows the tool to tune the scope of the analysis depending on which design decisions have been evolved. Designers must associate a *Rationale* to each design decision in order to keep track of the reasons behind the decision. A design decision can reference a set of *Stakeholders* (that can be either responsible for or interested in it). Stakeholders can have also a set of *Concerns* identifying the key design issues related to a design decision. A design decision may either raise or pertain to a specific concern. Basically, concerns can be seen as the means to evaluate a set of possible design decisions in order to choose among a set of alternatives. For example, typical concerns can be “low user effort”, “security”, “simplicity”, “low cost solution”, and so on. In order to keep design decisions cognitively manageable and well scoped, they can be organized into *categories*.

Providing a concrete syntax for ADD models is out of the scope of this work, so we have defined a simplified graph-based representation that renders ADDs as nodes of the graph and their relationships as edges between nodes. An example of this pseudo-notation is shown in Section 4.2.

3.2 Tracing Design Decisions to/from other Artifacts

As introduced in the beginning of Section 3, in our approach tracing links are contained into special models (technically called weaving models). This allows us (i) to keep the ADD model, architecture descriptions and requirements clearly separated and (ii) to be independent from any architectural language or requirements specification language. A weaving model can be seen as a means for setting fine-grained relationships between models and executing operations on them. The types of relations that can be expressed in our weaving models are shown in Figure 3.

The *TracingModel* metaclass represents the root of each design decision weaving model. It has a reference to the decision model, and a reference to the artifact to be linked (i.e., *decisionModel* and *artifact*, respectively) and contains a set of *Links* (*links* in Figure 3) which can be given in any order. Each *Link* represents a fine-grained traceability link between one or more ADDs and one or more element of the

linked artifact; the “1..*” cardinality of these two relationships means that in this context traceability is a many-to-many relationship. The *kind* attribute specifies the type of the current link, it may be:

1. *tracing*: if the linked artifact elements trace to/from the linked ADDs;
2. *conflict*: if the linked artifact elements are not compliant with the linked ADDs;
3. *undefined*: if there is the need to link artifact elements and ADDs, but at the moment there is no way to establish whether it is a tracing or a conflict link.

Fig. 3. Design decision weaving metamodel

It is important to note that by using our approach, designers are not forced to manually create weaving models, rather designers define traceability links by simply dragging and dropping the elements to be linked across the artifacts; then the corresponding weaving models are automatically created by the tool.

3.3 Evolution Impact Analysis

In Section 3 we explained how an ADD may evolve, and the basic steps of our approach in analysing the evolution impact. In this section we detail each step of the analysis.

Identification of the evolved ADD elements. The main goal of this step is to identify the evolved ADD elements that are relevant for the evolution impact analysis. The outcome of this step is *E*: the set of ADDs that must be checked by the designer, e.g., ADDs that have been either added, modified (e.g., status change), or that depend on other evolved elements (e.g., added issue or concern). In order to keep the approach well focussed on the changes that are relevant for evolution impact analysis, only some kinds of change within the ADD model are considered during analysis. For example, a change of the rationale of an ADD or a change of the description of an issue are not very much related to evolution impact analysis; the same holds for categories, stakeholders, and concerns. Referring to the *ADDMM* metamodel, the kind of changes relevant for our evolution impact analysis are:

- Addition/deletion of an issue

- Addition/deletion of a concern
- Addition/modification/deletion of a design decision

Figure 4 shows how the E set is identified. The E set is computed by comparing the current version of the ADD model ($v_{current}$ in figure) with its latest stable version (v_{old} in figure); the current prototype automatically keeps track of the last two versions of the ADD model, see Section 4.3. The comparison between the two version of the ADD model is performed by reusing an already existing model differencing tool, which produces as output a difference model (*diff model* in figure). Such a model enables timely identification of added, deleted or modified elements within the whole ADD model. The Evolution Manager component analyzes the *diff model*, filters out all the changes that are not relevant for our evolution impact analysis, and updates the current version of the ADD model by marking all design decisions in E as *evolved*. These design decisions will be the focus of next analysis steps.

Fig. 4. Identification of the initial set of evolved design decisions

Inter-decisions analysis. At this point, the approach establishes the ADDs in the model that depend on the ones in E , and adds them to E . The E set is populated with these extra design decisions as follows:

$$computeImpact(E) = E \cup \bigcup_{e \in E} computeImpact(dependsOn(e))$$

where $dependsOn(e)$ is the set of all design decisions depending on e . This dependency relationship is calculated as the union of all the other references of a specific design decision (e.g., *constraints*, *conflictsWith*, etc.).

The updated E set now represents all ADDs that must be checked with respect to the performed evolution. All the ADDs included into the E set are presented to the designer, so that she can focus and reason only on those impacted by the evolution. As stated at the beginning of Section 3, it is important to *validate* the current version of the ADD model; to this purpose, we defined a set of constraints that are used to semantically check the ADD model. These constraints are defined in the Object Constraint Language (OCL), an OMG standard language used to describe expressions on models. Listing 1.1 shows an example of such constraints: it considers each design decision in the ADD model (line 1 in the listing), and checks if the design decision (either directly or indirectly) conflicts with itself (line 3 in the listing).


```

1 context DesignDecision
2   inv conflictsWithSelf :
3     not self.allConflicts()->exists(e | e = self)

```

Listing 1.1. Example of OCL constraint validating Design Decisions

Our approach defines different constraints like this, for example there is a constraint checking if there is some issue which is not addresses by any decision, there are constraints checking decisions with inconsistent statuses (e.g., a rejected decision which constraints an accepted one), and so on. Due to space limitations, we do not describe those constraints in this paper.

At this point, the ADD model will be corrected by the designer, and checked again in an iterative fashion, until the ADD model reaches a stable correct status. So, at the end of this step the ADD model is correct and its design decisions are consistent with each other.

Extra-decisions analysis. The scope of this part of the analysis is the whole set of involved artifacts (i.e., requirements and architectural descriptions). The main goal of this analysis is to identify which elements in other artifacts are impacted by the evolved design decision model. So, the weaving links connected to at least a design decision in the E set are shown to the designer in a different way. It is important to note that this analysis step is skipped if all design decisions in E are internal decisions (see metamodel in Section 3.1) since in this case there is no evolved design decision that is related to any external artifact.

Validation is an important issue also in this step; however, unlike inter-decisions validation, extra-decisions validation must consider the both the ADD model and other different artifacts (and their relationships) at the same time. This is achieved by specifying OCL constraints on weaving models, rather than on the ADD model itself. For example, Listing 1.2 shows an excerpt of OCL constraint that considers each *Link* contained into a weaving model between an SA description and a DD model (line 1), and checks if it is related to a *rejected* design decision (line 4 in the listing); this situation may be risky because the state of the decision may have changed to *rejected*, but the architectural element linked to that decision has not been updated accordingly.

```

1 context Link
2   inv rejectedDecisions : if(self.kind = LinkKind::tracing)
3     then
4       self.designDecision->forAll(e | e.state != 'rejected')
5     else true
6   endif

```

Listing 1.2. Example of OCL constraint validating related artifacts

The result of this validation gives designers insights and accurate information about which element (being it an architectural element, or a requirement) is not aligned with the evolved design decisions. For example, there may be:

- an architectural component that is in conflict with a chosen decision,
- a requirement which is traced to a rejected decision,
- an architectural element which is traced to two mutually exclusive design decisions.

Such validation helps the designers to ensure the normal functioning of the system even during impairments caused by changing decisions because any conflicting decision is immediately flagged. It is important to note that, unlike many other approaches for design decision analysis, our approach allows to trace the evolution not only to the various impacted artifacts (for example, to a specific architecture description), but also to fine-grained elements of the artifacts (like specific architectural components, interfaces, and so on). It is up to the designer now to analyze the information provided by the analysis step and to correct the involved artifacts according to the evolved design decisions. Clearly, a change in one of the artifacts may imply a change back to the ADD model, triggering again a new design decisions evolution: this results in an iterative process in which design decisions evolution is iteratively evaluated and reflected to the involved artifacts. This process is part of the typical architectural reasoning activity.

4 Prototype Implementation

The current version of the prototype realizing the approach has been implemented as an Eclipse³ plugin that relies on the Atlas Model Management Architecture (AMMA) [15]. We chose AMMA since it best fits our technical needs like (ADL and requirements) metamodel independence, high flexibility of the traceability metamodel, integration to Eclipse and its modeling facilities [16]. The next sections will detail the technologies we used in each part of the proposed approach.

4.1 Models and Metamodels Implementation

Models and metamodels are expressed via Ecore, the metamodeling language of the Eclipse Modeling Framework (EMF⁴), and serialized into standard XMI files. The technology we use for creating weaving models is the Atlas Model Weaver (AMW) [17]. In order to have a homogeneous framework we make the assumption that all heterogeneous artifacts that we use are models and that each model conforms to its metamodel. This enables the management of the tracing and validation of the involved artifacts since their elements are fully defined and encoded in the models. The assumption also seems reasonable in light of the recent Doc2Model (Document to Model⁵) Eclipse project for parsing structured documents to produce EMF models. We see this assumption as reasonable considering the benefits it offers.

4.2 Seamless Visualization of Architectural Artifacts

As previously stated, it is fundamental to clearly visualize the ADD model together with its related artifacts. So, while designing the tool we had two orthogonal concerns: (i) to show the ADD model together with other models in a seamless way, and (ii) to provide a means to easily define tracing links between all those models. In this work we propose a solution in which ADD models and their related artifacts can be displayed

³ <http://eclipse.org>

⁴ <http://www.eclipse.org/modeling/emf>

⁵ <http://eclipse.org/proposals/doc2model>

and edited simultaneously on different 3D planes and 3D links can be used to display inter-model connections. Figure 5 gives an overview on how the current version of the prototype renders ADDs and the other artifacts to designers. More specifically, it shows ADDs by means of the graphical notation we defined for the *ADDMM* metamodel, requirements as a UML use case diagram and the architecture of the system as a UML component diagram. It is important to note that our approach is not dependent on UML; UML has been used only for illustrative purposes.

Fig. 5. Visualization of ADDs and their relationships to other artifacts

Requirements, ADDs and architecture descriptions are rendered on three separated planes in a 3D environment. ADDs are in the middle plane and in general are shown as white boxes. In Figure 5 there are some coloured decisions indicating the results of the evolution analysis: red ADDs are the evolved ones, and yellow ADDs are those depending on the evolved ones. The prototype automatically updates the ADD model by coloring the ADDs according to the results of the evolution analysis. The other planes are dedicated to requirements and architecture descriptions. It is important to note that designers may add other artifacts via additional planes; this can be done by simply dragging and dropping a (suitably adapted) diagram into the 3D environment.

Tracing links are visualized as connections between the various elements of the models displayed on these planes. Also, the colours of the connections play an important role: red connections indicate violations of some rule of the extra-decision analysis presented in Section 3.3, whereas the tracing links that passed the analysis are rendered as green.

From a technological point of view, the 3D environment and the visualization of the involved models on planes rely on GEF3D⁶, an Eclipse modeling environment sup-

⁶ <http://www.eclipse.org/gef3d/>

porting 3D diagrams editing. GEF3D allows designers to create 3D diagrams, 2D diagrams and combine 3D with 2D diagrams (this is what our prototype currently does). An important feature of GEF3D is that existing Eclipse-based 2D editors can be easily embedded into 3D editors; so designers can basically reuse their preferred notations in our approach with minimal effort. Issues like 3D camera management, 3D rendering of the models, adaptation of existing notations to the 3D environment are transparently supported by GEF3D.

4.3 Evolution Analysis Implementation

The current version of the tool proposed three buttons in the GUI to execute each step of the evolution impact analysis: one to identify the evolved ADD elements, and other two buttons in the GUI to perform inter- and extra- decisions analyses, respectively. Behind the lines, the three steps of the evolution analysis implementation are realized as a combination of model-to-model transformations, a model comparison technology and OCL constraints. The next sections will give some implementation detail of each step of the evolution impact analysis.

Identification of evolved decisions. By referring to Figure 4, the identification of the evolved ADD elements is a two-steps process: comparison of the ADD model with its latest stable version and evolved decisions identification.

For what concerns the comparison step, we reuse the EMF Compare⁷ tool. We decided to use EMF Compare because it is well integrated in Eclipse, and provides a set of APIs to interact with its facilities. Among other features, EMF Compare includes a generic comparison engine and creates the corresponding delta; this delta may itself be represented as a model. We use this model as a means to represent differences between the current ADD model and its latest stable version (it is called *diff model* in Figure 4). Clearly, in our approach the intermediate *diff model* is kept in memory, and discarded after the results of the analysis are shown in the initial ADD model.

Evolved design decisions represent the *E* set described in Section 3.3 and are identified in the model by their *evolved* attribute set to *true*. The *Evolution Manager* component in Figure 4 is implemented as a model-to-model transformation that takes as input the ADD model and the *diff model*, selects design decisions are related to relevant evolved ADD elements, and updates the ADD model by setting to *true* the "evolved" attribute of all evolved decisions. This transformation is specified using the Atlas Transformation Language (ATL) [17], an hybrid model transformation language with declarative and imperative constructs.

Inter- and extra- decisions analyses implementation. In the current version of the prototype, inter- and extra- decisions analyses are executed by pressing two dedicated buttons in the GUI, one for each kind of analysis; this allows designers to easily check the current state of design decisions without interrupting their reasoning process. The results of the validation will be shown in the standard "Error Log" view of the Eclipse framework (see Figure 6 for a screenshot of the prototype showing a list of issues).

⁷ <http://www.eclipse.org/emf/compare/>

The identification of the element impacted by a design decision evolution is performed by means of a set of transformations similar to the one realizing the *Evolution Manager* component. The only difference is that in inter-decisions analysis, the E set is expanded with design decisions dependent on some decisions already in E , whereas in extra-decisions analysis, the transformation marks weaving links

Message	Plug-in	Date
D1 is self conflicting (cycle: D1-D3-D4-D1)	it.univaq.intraddanalysis	28/04/11 15:29
D5 does not contain any rationale	it.univaq.intraddanalysis	28/04/11 15:29
D3 is external, but it is not linked to any external artifact	it.univaq.extraddanalysis	28/04/11 15:29
D6 is approved, but it conflicts with external artifacts elements	it.univaq.extraddanalysis	28/04/11 15:29

Fig. 6. The results of inter- and extra- decisions analyses

For what concerns the validation parts of the analyses, they are implemented as a set of OCL constraints specified using the Topcased-VF OCL evaluator⁸. Moreover, tracing links from/to design decisions to other related artifacts are rendered as red or grey, depending on the results of the validation; i.e., if some tracing link does not pass the inter-decisions analysis, it is rendered as red in order to help designers to have a quick idea of the problems affecting current design artifacts. (see Figure 5 for a screenshot of the prototype showing this feature).

5 Related Work

The evolution of software systems was recognized as early as 1969 [18]. Since then, lot of efforts have been directed towards managing evolution in a effective manner. ISO/IEC 42010 standard on Architecture Description [14], provides direct correspondence between ADDs and Architecture Description elements which we have represented as tracing links.

Jansen and Bosch have proposed in [1] a tool that provides an architecture description language, a notation for representing design decisions, and a mechanism for integrating them. Similarly to our approach, their main goal is to keep ADDs as first-class entities and to strongly connect them to the architecture. However, they focus less on the various relationships that may exist either among ADDs or between ADDs and other artifacts. We believe that this plays a fundamental role in characterizing the evolution. Also, their approach depends on the Archium ADL itself, whereas our approach is ADL-independent.

Capilla et al. have proposed a metamodel for architecting and evolving architectural knowledge in [11]. The metamodel comprises three parts (the project model, the decision model and the architecture model) and they provide an *integrated view* combining

⁸ <http://gforge.enseeiht.fr/projects/topcased-vf>

requirements, ADDs and software architecture into a single process. Even if we share with Capilla et al. the same idea of an integrated view including requirements, ADDs and architecture, we focus more on characterizing how design decisions may evolve and on what is the impact of such an evolution either on other ADDs or other related artifacts. Furthermore, our approach allows designers to simultaneously visualize and better reason about ADDs and their relationships to other artifacts.

Authors of [5] propose a metamodel for ADDs and establish a 1:1, 1:n and n:1 relationship between ADDs and the elements. To handle evolution, they have used a decision constraint graph and have also provide a change impact analysis using an example. However, differently from our approach, their ADD model includes very limited relationships among ADDs, it does not relate ADDs to requirements, and is not independent from the language to design the architecture.

Summarizing, the main differences between the approaches described above and our work are that (i) we use a generic and notation-independent metamodel to represent ADDs, (ii) our approach ensures that requirements and architecture can be represented using any preferred language, and (iii) our approach allows designers to simultaneously visualize and reason about ADDs and their relationships to other artifacts.

6 Conclusion and Future Work

Design decisions must be considered first-class elements when architecting software systems, and they have to be explicitly documented and supported. The fact that software architecture is a continuously evolving entity is not new, so explicit support and managing of the evolution of design decisions is becoming a necessity. The strong relationship that design decisions share with requirements and other artifacts needs to be taken into consideration while supporting evolution.

In this work we propose a model-driven approach for supporting the evolution of design decisions. Within the approach, our main contributions are (i) a metamodel for representing evolving design decisions; (ii) a means to create bidirectional traceability links between ADDs, requirements and artifacts which will help in analysing the impact of evolution on the artifacts; (iii) a technique to identify evolved ADDs and to check the impact of such an evolution both on other decisions and on other related artifacts.

As future work, we are investigating to make the proposed approach totally independent from the metamodel used to represent ADDs. This can be done by allowing designers to create models representing only the evolution of ADDs and then providing a mechanism to bind those models on the various models used to represent ADDs. Furthermore, it would be interesting to enhance the evolution analysis part with some mechanism to (semi)automatically solve the identified problems (e.g., by applying some resolution pattern to each constraint running during the analysis).

An interesting and important increment to our work will be to add quality aspects to the model. Though currently we take into account stakeholder concerns, we have not explicitly accounted for non-functional aspects like scalability, reliability, dependability etc in our model. By including these, we believe that our model will be better capable of handling evolution thereby ensuring that the system retains functionality even while undergoing changes.

Finally, the completeness of the tracing relationship types, and of the set of constraints executed during the evolution analysis will be experimented on a concrete, industrial case study. This will also help in understanding possible usability and performance issues related to the tool implementing the approach.

References

1. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: WICSA 2005. (2005)
2. Kruchten, P.: An Ontology of Architectural Design Decisions in Software Intensive Systems. In: 2nd Groningen Workshop Software Variability. (2004) 54–61
3. Potts, C., Bruns, G.: Recording the reasons for design decisions. In: 10th International Conference on Software Engineering ICSE 1988. (1988) 418–427
4. Capilla, R., Nava, F., Tang, A.: Attributes for characterizing the evolution of architectural design decisions. In: Software Evolvability, 2007 Third International IEEE Workshop on. (2007) 15 –22
5. Choi, Y., Choi, H., Oh, M.: An architectural design decision-centric approach to architectural evolution. In: Advanced Communication Technology, 2009. ICACT 2009. 11th International Conference on. Volume 01. (2009) 417 –422
6. Gilson, F., Englebert, V.: An architectural design decision-centric approach to architectural evolution. In: 5th Workshop on Variability Modeling of Software-Intensive Systems. (2011)
7. Mens, T., Magee, J., Rumpe, B.: Evolving software architecture descriptions of critical systems. *Computer* **43** (2010) 42–48
8. Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P.: A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software* **80**(1) (2007) 106 – 126
9. Tang, A., Liang, P., Clerc, V., van Vliet, H.: Supporting co-evolving architectural requirements and design through traceability and reasoning. In: *Relating Software Requirements and Architectures*. Springer (2011)
10. Capilla, R., Nava, F., Perez, S., Duenas, J.C.: A web-based tool for managing architectural design decisions. *ACM SIGSOFT Software Engineering Notes* **31**(5) (2006)
11. Rafael Capilla, Francisco Nava, J.C.D.: Modeling and documenting the evolution of architectural design decisions. In: Second Workshop on SHaring and Reusing architectural Knowledge Architecture. (2007) 9
12. Leventovszky, T., Rumpe, B., Schätz, B., Sprinkle, J.: Model evolution and management. In: *Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems*, Berlin, Heidelberg, Springer-Verlag (2010) 241–270
13. Eklund, U., Arts, T.: A classification of value for software architecture decisions. In: *Proceedings of the 4th European conference on Software architecture*. ECSA'10, Berlin, Heidelberg, Springer-Verlag (2010) 368–375
14. ISO: Final committee draft of Systems and Software Engineering – Architectural Description (ISO/IECFCD 42010). Working doc.: ISO/IEC JTC 1/SC 7 N 000, IEEE (2009)
15. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: *LNCS*, Vol. 3599. (2005)
16. Bézivin, J., Bouzitouna, S., Del Fabro, M., Gervais, M.P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.F.: A canonical scheme for model composition. (2006)
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: *Proceedings of the MTP, Workshop at MoDELS 2005*. (2006)
18. Lehman, M.: The programming process. IBM Research Report (1969)