Architectural design decisions

Jansen, Antonius Gradus Johannes

CHAPTER 6

EVALUATION OF TOOL SUPPORT FOR ARCHITECTURAL EVOLUTION

Published as: Anton Jansen, Jan Bosch, Evaluation of Tool Support for Architectural Evolution, Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), pp. 375-378, September 2004

Abstract

Evolution of software architectures is, unlike architectural design, an area that only few tools have covered. We claim this is due to the lack of support for an important concept of architectural evolution: the notion of architectural design decisions.

The absence of this concept in architectural evolution leads to several problems. In order to address these problems, we present a set of requirements that tools should support for architectural evolution. We evaluate existing software architecture tools against these architectural requirements. The results are analyzed and an outline for future research directions for architectural evolution tool support is presented.

6.1 Introduction

Software architecture [119, 138] has become a generally accepted concept in research as well as industry. The importance of stressing the components and their connectors of a software system is generally recognized and has lead to better control over the design, development and evolution of large and increasingly dynamic software systems.

Although the achievements of the software architecture research community are admirable, architectural evolution still proves to be problematic. Most research and tool development in the area of software architecture has focused on the careful design, description and assessment of software architecture. Although some attention has been paid to evolution of software architecture, the key challenge of the software architecture community has been that software architectures need to be designed carefully because changing the software architecture of a system after its initial design is typically very costly. Many publications refer to changes with architectural impact as the main consequence to avoid.

Interestingly, software architecture research, almost exclusively, focuses on this aspect of the problem. Very little research addresses the flip side of the problem, i.e. how can we design, represent and manage software architectures in such a way that the effort required for changes to the software architecture of existing systems can be substantially reduced. As we know that software architectures will change, independently of how carefully we design them, this aspect of the problem is of particular interest.

To reduce the effort in changing the architecture of existing software systems it is necessary to understand why this is so difficult. Our studies into design erosion and analysis of this problem, i.e. [168] and [78], have led us to believe that the key problem is knowledge vaporization. Virtually all knowledge and information concerning the results of domain analysis, architectural styles used in the system, selected design patterns and all other design decisions taken during the architectural design of the system are embedded and implicitly present in the resulting software architecture, but lack a first-class representation.

The design decisions are cross-cutting and intertwining at the level at which we currently describe software architectures, i.e. components and connectors. The consequence is twofold. First, the knowledge of the design decisions that lead to the architecture is quickly lost. Second, changes to the software architecture during system evolution easily cause violation of earlier design decisions, causing increased design erosion [72, 119, 168].

We believe that architectural design decisions are a key concept in software architectures, especially during evolution. Consequently, capturing design decisions is of great importance as it addresses some fundamental problems associated with architectural evolution. Effective support for architectural design decisions requires tools support, but this currently largely lacking. However, existing tools do provide parts of the necessary solution. The aim of this paper is therefore to evaluate existing software tools with respect to their ability to represent aspects of architectural design decisions.

The contribution of this paper is threefold. First, it develops the notion of explicit architectural design decisions as a fundamental concept during architectural evolution. Second, it states a set of requirements that tooling needs to satisfy in order to adequately support evolution of architectural design decisions. Finally, an analysis is presented of existing tool approaches with respect to the stated requirements.

The remainder of this paper is organized as follows. The concept of architectural design decisions and the problems associated with the architectural evolution are explained in more depth in section 6.2. In section 6.3, the requirements for tool support of architectural design decisions are formulated. The section after that presents the tools and the evaluation on the requirements stated in the proceeding section. A discussion of the evaluation results is presented in section 6.5. The paper concludes with future work and conclusions in section 6.6.

6.2 Architectural Design Decisions

Architectural evolution is closely related to architectural design decisions. In our experience, architectural evolution is fundamentally the addition, change and removal of architectural design decisions. Classifications such as [14, 27, 109] can all be viewed as the consequences of architectural design decisions.

Due to new and changed requirements, new design decisions need to be taken or existing design decisions need to be reconsidered. Consequently, the architectural design decision is the central concept in the evolution of software architectures. We define an architectural design decision as:

> A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.

This definition of architectural design decisions uses the following elements:

- **Architectural change** Describes the addition, subtraction, and modification of the architectural entities used in the architecture.
- **Rationale** Design decisions are often made with a reason, these reasons behind an architectural design decision are part of the rationale of an architectural design decision.
- **Design rules** Architectural design decision can prescribe rules for architectural change.

- **Design constraints** Besides design rules, certain design possibilities can be invalidated by an architectural design decision. Design constraints therefore describe the opposite side of design rules, i.e. they prohibit certain architectural changes.
- **Additional requirements** Design decisions can give rise to additional requirements. Since designing is often an exploratory process, new requirements will be discovered once a design decision has been made and the architectural change is developed further.

An architectural design decision is therefore the result of a design process during the initial design or the evolution of a software system. This process can result in changes to architectural entities, e.g components & connectors, that make up the software architecture or design rules and constraints being imposed on the architecture (and on the resulting system). Furthermore, additional required functionality (a form of constraining) can be demanded from the architectural entities as a result of a design decision. For instance, when a design decision is taken to use an object-oriented database, all components and objects that require persistence need to support the interface demanded by the database management system.

Architectural design decisions may be concerned with the application domain of the system, the architectural styles and patterns used in the system, COTS components and other infrastructure selections as well as other aspects needed to satisfy all requirements.

However, current approaches to the description of software architectures leave the notion of architectural design decisions implicit and provide no mechanism for representing this concept. During the system's evolution, the software architect is forced to reconstruct the design decisions underlying the current architecture that are relevant for the required changes. This leads to a number of problems that are insufficiently addressed:

- **Lack of first class representation** Architecture design decisions lack a first class representation in the software architecture. Once a number of design decisions are taken, the effect of individual decisions becomes implicitly present, but almost impossible to identify in the resulting architecture. Consequently, the knowledge about the what and how of the software architecture is soon lost [22, 89] Some architecture design methods [29, 67] stress the importance of documenting architecture design decisions, but experience shows that this documentation often is difficult to interpret and use by individuals not involved in the initial design of the system.

- **Design decisions cross-cutting and intertwined** Architecture design decisions typically cross-cut the architecture, i.e. affect multiple components and connectors, and often become intimately intertwined with other design decisions.
- **High cost of change** A consequent problem is that a software architecture, once implemented in the software system, is sometimes so expensive to change that changing the architecture is not economical viable. Due to the lack of first-class representation and the intertwining with other design decisions, changing or removing existing design decisions is very difficult and affects many places in the system.
- **Design rules and constraints violated** During the evolution of software systems, designers, and even architects, may easily violate the design rules and constraints imposed on the architecture by earlier design decisions.
- **Obsolete design decisions not removed** Removing obsolete architecture design decisions from an implemented architecture is typically avoided, or performed only partially, because of (1) effort required, (2) perceived lack of benefit and (3) concerns about the consequences, due to the lack of the necessary knowledge about the design decisions. The consequence, however, is the rapid erosion of the software system, resulting in high maintenance cost and, ultimately, the early retirement of the system [72, 119, 168].

To solve the aforementioned problems, an important role is laid down for the notion of an architectural design decision. Only with proper tool support can this notion be realized. Consequently, requirements are needed for such tools in order to support this fundamental concept of architectural evolution. In the following section, the requirements for architectural design decisions are presented. The requirements in turn are used in section 6.4 to evaluate which parts of architectural design decisions are already supported in existing tools.

6.3 Requirements

In this section, the requirements are presented that need to be satisfied to support architectural evolution in the form of architectural design decisions. The individual requirements have been grouped together into three groups: architecture, architectural design decisions, and architectural change.

Architectural design decisions deal with the evolution of software architecture. Consequently, tools should satisfy requirements regarding software architectures. The concept of architectural design decisions introduces some requirements as well, which are covered in the group architectural design decisions.

Architectural change is an important part of an architectural design decision. Requirements for the changing influence architectural design decisions have on the architecture are covered by the architectural change group of requirements.

The requirements for tools to provide architectural design decision support are the following:

6.3.1 Architecture

1. **First class architectural concepts** For software architecture to be of use, it is required to have a shared domain model between the stakeholders. The domain model should provide a common ground for the stakeholders involved about the concepts used in the software architecture. However, a shared domain model alone is not enough; the architectural concepts also need to be first class citizens. As software architecture deals with abstractions, it is very important to define these abstractions in a first class way. The way in which abstraction choices are made is very subjective and this greatly influences the resulting architecture.

   Expressing these abstractions in a consistent and uniform way is therefore essential for software architectures. Only when the architectural concepts become first class can abstraction choices be explicitly communicated. Consequently, misinterpretations about the used abstractions are reduced. This effect not only eases negotiations about the entities of an architecture, but also communication and reasoning about the evolution of the architecture becomes easier.

2. **Clear, bilateral relationship between architecture and realization** In the view of evolution, it is important to have a bilateral relationship between the software architecture and the realization [106]. During evolution changes in the architecture will have an effect on the realization of the system and vice versa. Knowing the relationship between the architectural entities and their realization is essential for reasoning about the effects of change (caused by architectural design decisions) have on the architecture or realization.

3. **Support multiple views** Software architectures have different views for different concerns [29, 67, 92]. Architectural design decisions often influence multiple concerns simultaneously, because they try to strike a balance in the effects they have on different concerns in their changes. Consequently, for architectural evolution it is important to know these different concerns. Hence, multiple views on the architecture should be supported.

6.3.2 Architectural design decisions

4. **First class architectural design decisions** Architectural design decisions in our vision are the architectural changes evolving the architecture, as already stated in the introduction (see section 6.1). A first class representation of design decisions is required to make evolution explicit. First class design decisions can be communicated, related and reasoned about. For example, an important relationship is the dependency between design decisions. If a design decision is undone, it is important to know which other design decisions might be invalidated. The dependency relationship can provide this information. However, this can only be achieved if there is a first class notion of an architectural design decision in the first place.

5. **Under-specification and incompleteness** An important property of design decisions is the ability to specify incompleteness. The proposed solutions in a design decision can only give a general and incomplete description of how the problem at hand could be solved. Consequently, design decisions are often made on the basis of incomplete information. Knowing which part of the chosen solution is not known is important meta-knowledge. Since designing is (partly) an exploratory process, these incomplete parts are often the sources for additional design decisions and invalidations of implicit assumptions earlier made. Hence, they are important for an architect to evolve an architecture.

6.3.3 Architectural change

6. **Explicit architectural changes** A well defined relationship between the proposed solutions of an architectural decision and the architectural entities involved is essential. The proposed solutions describe design alternatives for solving the problem of a design decision.The decision part of a design decision decides which of the proposed solutions will be realized. The realization of these solutions will change the architecture. The resulting architectural changes define great parts of the semantics of the solution. As the exact semantics cannot be known in advance without much effort. Consequently, the architectural change required to realize the solution is an inherent part of the solution. Hence, an explicit representation of these architectural changes is required to express this important part of design decisions. Architectural changes therefore form the bridge between the first class architectural entities and the rationale part of architectural design decisions.

7. **Support for modification, subtraction, and addition type changes** The characteristic types of change of evolution often distinguished are the corrective, perfective, and adaptive types [51]. However, this classification focuses on the reasons behind the change, not on the effect the changes have on the system. Software architectures primarily deal with the global structure of the system. Consequently, architectural evolution has its main focus on the structural type of changes:modification, subtraction, and addition [109, 115]. Replacement is not a basic change type, as it can be accomplished using subtraction followed by addition in one atomic action. Tools need to support these three basic change types on architectural entities if they want to support architectural evolution. Otherwise, not all types of changes to the architecture can be managed by the tools.

6.4 Evaluation

In this section, six tools are evaluated against the requirements stated in section 6.3. For each tool, an overview description is provided, followed by a short description of the support the tool provides for each requirement. The evaluation presented will be used in section 6.5 to discuss the general tool performance on the three groups of requirements.

6.4.1 ArchStudio 3

6.4.1.1 Description

ArchStudio 3 [106, 115] is an architecture-driven software development environment, i.e. software development from the perspective of software architecture. ArchStudio supports a particular architectural style: the C2 architectural style. C2 [154] supports the typical architectural concepts as components, connectors and messages. The C2 architectural style is a hierarchical network of concurrent components linked together by connectors (or message routing devices) in a layered way. C2 requires that all communication between C2 components is achieved through message passing.

ArchStudio uses the C2 architecture expressed in xADL[37] as a way to communicate with external tools. The sister application Ménage [161], used for software product family architectures (SPF) [19], has an architectural change management support tool called Mae [161]. The Mae tool is a change management tool for the

C2 specifications supporting the evolution of the architecture definition by revision management.

6.4.1.2 Evaluation

- **Architecture**
  - R1 The underlying C2 part of ArchStudio supports first class architectural concepts as architecture, configuration, components, connectors and messages.
  - R2 ArchStudio only supports a one-way, relationship from the software architecture to the realization (Java).
  - R3 The architectural concepts supported by ArchStudio are all part of the Component & Connector view [29] on the software architecture. ArchStudio does not support other architectural views.
- **Architectural design decisions**
  - R4 The concept of an architectural design decision is not supported by ArchStudio.
  - R5 Since the concept of an architectural design decision is not supported, there is no support for under-specification and incompleteness. However, ArchStudio does support inconsistent architectural models, which is part of the required incompleteness.
- **Change**
  - R6 Explicit architectural changes are only partially supported by ArchStudio. The tool supports the basic operations and does not have a first class representation of a change in itself. However, Mae, the change management tool, which is no part of but related to ArchStudio, has support for the notion of revisions. Consequently, explicit architectural changes are only supported by an external tool (Mae) and not by ArchStudio itself.
  - R7 ArchStudio supports the basic operations of change, apart from the modification type. However, replacement as in an atomic subtraction and addition operation is not supported. Mae claims to support replacement of components by encapsulating the separate versions in one component and allowing run-time change between them.

6.4.2 ArchJava

6.4.2.1 Description

ArchJava [2, 3] is an extension to Java that aims to unify software architectural concepts with the implementation. It uses a type system to ensure that the implementation conforms to an architecture and focusses especially on communication integrity. Communication integrity defines that the components in a program can only communicate along declared communication channels in the architecture. ArchJava enforces this communication integrity for control flow: a component may only invoke an operation of another component if it is connected to the other component in the architecture.

6.4.2.2 Evaluation

- **Architecture**
  - R1 ArchJava supports the architectural concepts of connectors, components, configuration, and ports.
  - R2 The supported architectural concepts are implemented directly as first class entities defined as an extension to Java. Consequently, there is no division between the architecture and its realization.
  - R3 Only the architectural concepts of the component & connector view [29] are supported by ArchJava.
- **Architectural design decisions**
  - R4 The concept of an architectural design decision is not supported by ArchJava.
  - R5 Since the concept of an architectural design decision is not supported, the language of ArchJava does not support under-specification and incompleteness.
- **Change**
  - R6 Architectural change is not explicitly supported in ArchJava.
  - R7 ArchJava does not support the basic change types for evolution. An exception is formed by the ability to add a connector. A special connect statement allows new connections to be created at run-time between the components, but only if their type is a subtype of an earlier defined type.

6.4.3 AcmeStudio

6.4.3.1 Description

AcmeStudio is the tool used as a front end for Acme [50], which is an architectural description language. The development of Acme started back in 1995 as an ADL interchange language but has evolved to an ADL itself. Acme currently makes use of xADL [37]. AcmeStudio is a graphical editor for Acme architectural designs, which allows editing of designs in existing styles, or creating new styles and types using visualization conventions that can be style dependent. The integrated Armani constraint checker [148] is used to check the architectural design rules. The tool is implemented as an Eclipse plug-in for portability and extensibility.

6.4.3.2 Evaluation

- **Architecture**
  - R1 Acme, as used by AcmeStudio, supports the architectural concepts of components, connectors, configuration, and ports.
  - R2 AcmeStudio has a code generating ability to Java and C++, resulting in a one-directional relationship from the architecture to the realization.
  - R3 AcmeStudio only supports the Component & Connector view [29]. Although the tool does support multiple views on the architecture (called "representations"), only one type is implemented.
- **Architectural design decisions**
  - R4 The concept of architectural design decisions is not supported by AcmeStudio.
  - R5 Since the concept of an architectural design decision is not supported, underspecification and incompleteness are not explicitly supported.
- **Change**
  - R6 Architectural change is not supported.
  - R7 Supporting mechanisms for the change types are not available.

6.4.4 SOFA

6.4.4.1 Description

SOFA (SOFtware Appliances) [121] is a component model in which applications are viewed as a hierarchy of nested components. In SOFA, the architecture of a

component describes the structure of the component by instantiating direct subcomponents and specifying the subcomponents' interconnections via connectors. The use of connectors in SOFA is to provide separation between the application logic and the necessary interaction semantics that cover deployment dependent details [41]. To describe its components and architectures, SOFA uses the Component Definition Language (CDL). The CDL is loosely based on the Interface Description Language (IDL) as used in CORBA [114].

6.4.4.2 Evaluation

- **Architecture**
  - R1 In the accompanied CDL, SOFA has a first class representations for architecture, modules, and components (called frames [121]).
  - R2 SOFA creates the architectural infrastructure. The implementation is still defined in separate Java classes. Consequently, SOFA uses a one-way code generation approach.
  - R3 Although SOFA defines modules and component concerns, these concerns cannot be used orthogonal in SOFA. Furthermore, SOFA only has a textual interface to the Component Definition Language presenting one view to the component model. Hence, multiple views are not supported in SOFA.
- **Architectural design decisions**
  - R4 The concept of (architectural) design decisions is not supported in SOFA.
  - R5 SOFA does not satisfy the requirement for incompleteness and under-specification.
- **Change**
  - R6 Architectural change is not supported as a first class entity. However, explicit versioning is part of the component definition language model defining the component model.
  - R7 SOFA has support for the replacement of components at run-time. However, the basic change types are not supported.

6.4.5 Compendium

6.4.5.1 Description

Compendium [8, 135, 181] is (partly) a knowledge system based on gIBIS [32], which in turn uses the decision model IBIS[97]. A knowledge system, which uses a decision model, tries to capture the rationale behind the decisions made in a decision process. Architectural design decisions can be seen as a very specific kind of

decisions made in a specific process (architectural design phase), which makes this tool a candidate for evaluation.

Compendium centers on capturing design rationale created in face-to-face meetings of design groups, potentially the most pervasive knowledge-based activity in working life, but also one of the hardest to support well. The tool provides a methodological framework, for collective sense-making and group memory. Compendium excels in enabling groups to collectively elicit, organize and validate information and make decisions about them. In order to integrate this with pre/post-meeting design activities and artifacts, the created reasoning diagrams can be transformed into other document formats for further computation and analysis. The domain independence of Compendium's reasoning mapping technique is the tool strength and weakness.

6.4.5.2 Evaluation

- **Architecture**
  - R1 Compendium has no notion of first class architectural concepts.
  - R2 A relationship between architecture and realization can only be defined by relating the artifacts in Compendium. Consequently, there is only an indirect relationship between the artifacts of the architecture and realization.
  - R3 As Compendium has no notion of architectural concerns, multiple views are not supported.
- **Architectural design decisions**
  - R4 Compendium has a first class notion of a design decision (in the form of an issue). Furthermore, it supports rationale capturing about design decisions.
  - R5 The argument and position elements of the IBIS model [97] allows for under-specification of the design decisions and the increasing refinement of them.
- **Change**
  - R6 Compendium does not support architectural change, let alone a first class representation of it. It does support versioning of an artifact describing the architecture. Through this, Compendium could track architectural change.
  - R7 As the notion of a software architecture is not known in Compendium, the basic changes type are not supported.

6.4.6 Archium

Please note that this evaluation of Archium was not part of the original publication of this chapter, but was part of the publication chapter 5. However, this evaluation was left out in the final publication due to space constraints. To be complete in our evaluation, we present this short evaluation of Archium.

6.4.6.1 Description

For an in-depth description of Archium, we refer to chapter 4 on page 79 and chapter 5 on page 101 of this thesis.

6.4.6.2 Evaluation

- **Architecture**
  - R1 The Archium tool supports first class architectural concepts of the component & connector view, like components, connectors, ports etc.
  - R2 The relationship between the architecture and realization is very closely related in Archium, as they have been integrated with each other. Consequently, a bilateral relationship exists between the two
  - R3 Archium does support multiple views in the form of the component & connector view [29], and an architectural decision dependency view. Figures 5.5 on page 115 and 5.1 on page 109 give concrete examples of these views. However, important views like module and deployment views [29] are to be added later on.
- **Architectural design decisions**
  - R4 The Archium tool includes a first class representation of architectural decisions.
  - R5 Under-specification and incompleteness are only partially supported in the tool in the form of stubs and optional rationale elements. However, the Archium tool lacks the capabilities to explicitly express the incompleteness of its architectural decisions and architectural elements. For example, if an architectural decision requires more investigation and is put aside for the moment, Archium cannot express this state of an architectural decision.
- **Change**
  - R6 In various ways, explicit architectural change can be expressed in the Archium tool. Changes to components, connectors and configurations all can be expressed explicitly.

Table 6.1: Evaluation result of the examined tools

Approach	Architecture			A.D.D.		Change	
	R1	R2	R3	R4	R5	R6	R7
ArchJava	++	++	-	- -	- -	- -	-
ArchStudio	+	+/-	-	- -	- -	+/-	+/-
AcmeStudio	+	+/-	+/-	- -	- -	- -	- -
SOFA	+	+/-	- -	- -	- -	+/-	+
Compendium	- -	- -	- -	+	++	- -	- -
Archium	++	++	+/-	++	+/-	++	+

R7 Furthermore, the Archium tool supports all the three basic change types (i.e. addition, subtraction, and modification of architectural entities) for these explicit architectural changes. However, this support is limited to the architectural entities. An external version management system should be used to track changes to the architectural decisions themselves.

6.5 Discussion

In the previous section, six tools were evaluated with respect to their support for architectural evolution. Table 6.1 provides an overview of the results of the evaluation, by presenting the tools against the requirements. For each requirements group (architecture, architectural design decisions, architectural change), the results are discussed in more detail.

6.5.1 Software Architecture

A clear bilateral relationship between architecture and realization proves to be troublesome for most tools. Most of them (ArchStudio, AcmeStudio, SOFA) use a generative approach. The architecture is defined first in the tool, which than generates the implementation classes once. For evolution this means that two models have to be maintained, one for the realization, and one for the architecture. ArchJava and Archium are the approaches in the evaluation that do not suffer from this co-evolution problem. Since these approaches mix the architecture first-class with the rest of the realization.

Software engineers are notorious in neglecting maintenance of separate models, because they find it very hard to do and do not see clear benefits in maintaining them.

In many organizations, it is not uncommon to use an explicit architecture specification only during the initial design phase. The explicit architecture is primarily used to explore ideas for potential solutions. Once the (automated) translation to a detailed design and implementation is made, the architecture description is no longer updated. Consequently, the description of the architecture slowly starts diverting from the architecture of the realization, which degrades the usefulness of the description of the architecture. In the view of architectural evolution, this is definitely a subject for further research.

The idea of separate view points to an architecture [29, 67] is neither supported, nor implemented in any of the evaluated tools besides Archium. All the evaluated tools apart from Compendium and Archium concentrate on a single view: the Component & Connector (C&C) view [29]. This is probably due to the fact that the C&C view is conceptually the closest to the result the architect want to achieve. The other views, in the view of the tools, more or less "result" from this view as specific realization choices are made by the tool when generating the skeleton for the implementation. Compendium does not support architectural views at all. Archium covers both an architectural design decision view and a component & connector view, but still lacks important other views, especially those of the module and deployment viewtype [29].

6.5.2 Architectural Design Decisions

Apart from Compendium and Archium, the concept of (architectural) design decisions is not supported. Compendium does support design decisions, as issues elements from the IBIS model. The tool supports arguments and positions of stakeholders regarding the current problem that can be solved (partially) by a design decision. A major drawback of Compendium (and other knowledge systems) is that it does not explicitly support architectural concepts. The tool only allows for references to be made to artifacts describing the context of the problem (i.e. the architecture) and artifacts describing potential solutions. Apart from Archium, there are currently (as of 2004) no artifacts for software architectures that can express design solutions to a design problem in an unambiguous way.

The idea of (architectural) design decisions resulting from a design decision process, as used in the research community of knowledge systems, is not treated explicitly in the software architecture community. Consequently, the design rationale of an evolving software architecture is not captured in these tools. Finally, architectural design decisions are implicit, which results in the problems already stated in the introduction (see section 6.1).

6.5.3 Architectural change

Architectural change is an important part of architectural design decisions, as this change primarily affects the architecture and evolves the system. The architectural change forms the bridge between the other aspects of an architectural design decision and the architecture.

Explicit architectural change is at the heart of Archium and is therefore well supported. Of the other evaluated tools, only SOFA and ArchStudio provide some support for this change and the required change types. SOFA has explicit version management in its component language and does support the basic change types to some extent. However, the *change* of the first class entities is not explicit and grouped. Hence, SOFA cannot relate the change to the evolution of the system.

ArchStudio, with the help of Mae, has some support for architectural change. Revisions of components and connectors can be stored and retrieved with the help of the Mae tool. However, the change itself is not explicit in Mae, i.e. the change of the revisions is not grouped in an explicit entity. Furthermore, Mae does not support all the change types. Especially, the modification change seems to be lacking.

Change management tools [174], which are not evaluated, could also be used to track architectural change. For example, they could track the changes Archium cannot capture. However, these approaches require a system model that describes the entities that can change and their relationships. However, these concepts for architectural change are not explicitly defined for these tools. Consequently, tracking architectural change with these kind of tools is troublesome.

In general, architectural tool support (apart from Archium) does not view evolution as an inherent part and separate dimension of a software architecture. The focus of the majority of tools is on defining the architecture in the right way. The change of the architecture is often overlooked and not implemented and left over to change management tools. Consequently, tool support for architectural evolution is currently only partially realized.

Archium has addressed some of the issues other tools have struggled with. Compared to other approaches, Archium performs quite well according to the defined criteria, but needs improvements to alleviate its shortcomings. Most significantly, the tool in its current form is not very suitable for the early design phases. This is because the architect generally uses natural language in these early phases to express the architecture, as opposed to the formalized architectural description in the Archium tool.

6.6 Conclusion

The notion of software architecture has become an increasingly important concept in software engineering research and practice. The rationale for this is that an explicit software architecture facilitates the control over the design and development evolution of large and complex software systems.

Evolution of software systems and their associated software architecture are not equally well supported by existing state-of-the-art approaches. Although basic analysis of architecture evolution can be performed, the evolution of the key architecture concepts during evolution are not captured. The claim of this paper is that this is due to the lack of support for the notion of architectural design decisions. In our experience, architecture evolution is fundamentally the addition, change and removal of architectural design decisions.

The lack of support for architectural design decisions in software architecture representations and associated tool support leads to several problems, including high cost of change of already implemented design decisions, the easy violation of design rules and constraints and the cross-cutting and intertwining nature of design decisions.

In this paper, we have presented a set of requirements that tools for architecture evolution should support. The requirements have been arranged into three groups: architecture, architectural design decisions, and architectural change requirements. For the software architecture, the requirements are: support for architectural concepts, a clear bilateral relationship to the realization, and support for multiple architectural views on a system. The architectural design decisions requirements include a first class representation of this concept and support for dealing with the incompleteness and under-specification of these decisions. The architectural change requirement consists of support for explicit architectural change and the ability of the tool to support the fundamental different types of change (i.e. modification, subtraction, and addition) on the architectural entities.

Overviewing the results of the evaluation, it is concluded that there exists a gap between tools for capturing rationale, architectural change, and software architectures. This gap can only be closed if architectural evolution becomes an inherent dimension of the description of a software architecture. We believe that architectural design decisions are the missing concept in this perspective and that tools need to support this concept in order to support architectural evolution.

In our future work, we will try to make the concept of architectural design decisions more explicit. The integration of architectural design decisions, the resulting

architectural change, and the relationship to the software architecture will be important areas of work. Hopefully, with the right decisions we will evolve and mature the concept of architectural design decisions to tool support, making architectural evolution an inherent part of software architectures.

"The Wheel of Time turns, and Ages come and pass, leaving memories that become legend. Legends fades to myth, and even myth is long forgotten when the Age that gave it birth comes again. In one Age, called the Third Age by some, an Age yet to come, an Age long past, a wind rose in the Mountains of Mist. The wind was not the beginning. There are neither beginnings nor endings to the turning of the Wheel of Time. But it was ***a*** *beginning"*

– The Wheel of Time series, Robert Jordan