Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations

Ioanna Lytra, Huy Tran, and Uwe Zdun

Faculty of Computer Science
Software Architecture Group
University of Vienna, Austria
Email: {ioanna.lytra,huy.tran,uwe.zdun}@univie.ac.at

Abstract. In recent years, the software architecture community has proposed to use architectural design decisions (ADDs) for capturing the design rationale and the architectural knowledge (AK). As software systems evolve both ADDs and architectural designs need to be documented and maintained. This is a tedious and time-consuming task because of the lack of systematic and automated support for bridging between ADDs and designs. As a result, decisions and designs become inconsistent over time. We propose to alleviate this problem by introducing an AK transformation language supporting reusable AK transformations from pattern-based ADDs to component-and-connector models. In addition, we devise reusable consistency checking rules for verifying the consistency between decisions and designs. Through the use of model-driven transformations, as well as reusable, pattern-based decision models, we ensure the reusability of our approach. We apply our approach in an industrial case study and show that it offers high reusability, is largely automated and scalable, and can deal with the complexity of large numbers of recurring decisions.

1 Introduction

Today, software architectures are usually described in various architectural views [7,3]. The component-and-connector (C&C) model of an architecture is a view that is often considered to contain the most significant architectural information [3]. Although C&C models offer a natural representation of software systems to software architects and designers, they fail to model the design rationale of the architecture and support the sharing of this knowledge among stakeholders. In recent years, software architecture is no longer solely regarded as the solution structure, but also as the set of architectural design decisions (ADDs) that led to that structure [8]. The actual solution structure, or architectural design, is merely a reflection of those design decisions. Architectural design views [11] document the design rationale of the architecture and contribute to the gathering of Architectural Knowledge (AK) and its sharing among different stakeholders. For organizing and documenting AK, various tools and methods that use AK templates [23], ontologies [13] or meta-models [25] have been proposed in the literature. To minimize the effort of documenting architectural decisions, approaches for

reusable architectural decision modeling [25] and using design patterns as a basis for documenting reusable ADDs (as [6]) have been proposed.

Unfortunately, in practice, the ADDs frequently are neither maintained nor synchronized over time with the corresponding C&C diagrams (or other design views) [11]. Thus, ADDs and design views drift apart as software systems evolve. This leads to a potential loss of architectural knowledge, a phenomenon which is known as architectural knowledge vaporization [8,6]. The main reason for the resulting inconsistency and lack of traceability between ADDs and design views is that there is no formal mapping between them. As a consequence, there is no automation for the translation between ADDs and design views, and in practice keeping them synchronized is a tedious manual task that depends highly on the architects' experience and interpretation. Making matters worse, the actual documentation of ADDs is also a tedious and time-consuming task, especially for similar ADDs that occur repeatedly throughout a design.

Our previous work [16] has partially solved this problem by addressing the bridging between the ADDs and designs. It introduced a formal mapping model between different ADD types, on the one hand, and elements and properties of C&C models, on the other hand. Based on this formal mapping model, preliminary component models and OCL-like constraints for consistency checking can be derived. Yet, so far this mapping model had to be manually created and modified. Therefore, the approach is not efficient for handling large numbers of ADDs and/or complex design models. Moreover, in reality there are often several recurring ADDs which can be applied in different contexts and for different elements or properties of the ADDs and designs. As a result, maximizing the reusability of such recurring decisions would significantly enhance the productivity in creating and maintaining the formal mappings between the decisions and the designs. This has not been addressed in our previous work [16] as the formal mapping and the derived constraints can not directly be reused because they are bound to specific elements or properties of the ADDs and designs.

We present in this paper a novel approach aiming to address the aforementioned challenges. In particular, our approach introduces an architectural knowledge transformation language that supports the specification of primitive and complex actions whose enactment leads to automatic updating of design models (i.e., C&C diagrams) based on changes in the ADD view. The transformation languages can be used to formulate the expected outcomes of a certain decision ranging from individual actions, such as creating new elements, grouping a number of elements, and deleting or updating existing elements in the C&C diagrams, to composite actions (e.g., for capturing reusable pattern-based ADDs) that might contain many primitive and/or other composite actions. These actions are designed to support the reusability of specifications in our AK transformation language, as existing actions can be efficiently reused and adapted for different designs where similar architectural decisions are taken. Each action also triggers the instantiation of corresponding constraint checking rule(s).

After the instantiation of the actions for concrete ADDs, we exploit template-based generation rules and model-driven techniques for automatically enacting the actions and generating consistency checking rules automatically. The linking of reusable ADDs to reusable actions and consistency rules (templates) offers higher reusability and automation and results in less complexity and modeling effort for the software architect. The reusability is achieved here (1) through the automatic derivation of parts of the C&C di-

agrams and consistency checking rules using model-driven templates and (2) by reusing common abstractions shared between common design patterns (see [24]).

To demonstrate our approach, we have implemented a prototype based on two existing tools from our previous work: ADvISE[1] – a tool for assisting architectural decision making for reusable ADDs, and VbMF[2] – a tool for describing architectural view models and performing model-driven code generation. Our approach presented in this paper will act as a bridge between ADvISE and VbMF. The prototypical implementation of our approach has been evaluated in scenarios extracted from an industrial case study to show that it is feasible and scalable for large numbers of ADDs.

The remainder of the paper is structured as follows. First, in Section 2 we explain the background for ADvISE and VbMF. In Section 3 we give an overview of our approach and describe the details about the reusable AK transformations and consistency checking rules. The application of our approach in the industrial case study and the evaluation of the reusability, complexity and modeling effort are presented in Section 4. We compare to related work in Section 5 and summarize key contributions in Section 6.

2 Background

In this section we briefly present ADvISE and VbMF, the two tools we integrate for demonstrating our approach.

2.1 Architectural Design Decision Support Framework

The Architectural Design Decision Support Framework (ADvISE) is an Eclipse-based tool that supports the modeling of reusable ADDs using Questions, Options and Criteria (QOC) [17] and the decision making under uncertainty. In particular, it assists the architectural decision making process by introducing for each design issue a set of questions along with potential options, answers and pattern-based solutions, as well as dependencies and constraints between them.

The advantage of the reusable ADD models is that they need to be created only once for a recurring design situation. In similar application contexts, corresponding questionnaires can be automatically instantiated and used for making concrete decisions. Based on the outcomes of the questionnaires answered by software architects through the decision making process, ADvISE can automatically generate architectural decision documentations. Our approach in this paper additionally introduces an architectural knowledge transformation framework (see Section 3) that supports the specification of reusable actions and the association of these actions with the elements of the aforementioned ADD models for automatically transforming ADDs into the underlying design models and generating constraints for consistency checking between them.

---

[1] `http://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_(ADvISE)`
[2] `http://swa.univie.ac.at/View-based_Modeling_Framework`

2.2 View-based Modeling Framework

The View-based Modeling Framework (VbMF) is also an Eclipse-based tool that implements a model-driven, architectural view model. That is, it leverages the notion of view models for describing various concerns of the software systems at different abstraction levels and model-driven development techniques for generating code and configurations from those view models [22]. Among other views, VbMF provides a high-level service component view model –similar to a typical C&C model such as UML component model– for representing essential architectural design elements such as components, ports, connectors, and properties that are independent from the underlying platforms and technologies. Technology- and platform-specific information will be described separately in the low-level view models that refine and enrich the high-level counterparts.

In this paper, we mainly use the high-level service component view model of VbMF (or in short form, the VbMF C&C view) for describing the architectural design of a software system. The advantage of using VbMF is that we can leverage the existing view model integration and transformation mechanisms of VbMF which are based on Eclipse technologies and therefore can be integrated well with ADvISE.

3 Reusable AK Transformations and Consistency Checking Rules

To illustrate the "big picture" of our approach we depict in Fig. 1 an overview of the tools and artifacts along with their interconnections. The artifacts that are automatically derived using model-driven techniques are indicated with dark-gray color.

The *ADD Model Editor* in ADvISE is a tool that is used to create the reusable ADD models (i.e., the artifact *Reusable ADDs* in the figure). It is created only once per application domain. From it, ADvISE can automatically generate *Questionnaires* for making *Actual ADDs*. They are made, possibly multiple times if multiple ADDs are derived from the same reusable ADD model, using the *Questionnaire Editor* tool. For using and manipulating *C&C Diagrams* VbMF provides the *C&C View Editor*. Our approach supports generating the first instance of the C&C Diagrams automatically from the ADD model using *Transformation Actions*. It can further execute automatic *Transformation Actions* on existing view models. In VbMF *C&C Diagrams* can be manually manipulated. To ensure that changes in the C&C models do not violate the ADDs, *Consistency Checking Rules* are generated from the *Transformation Actions*, which are automatically enacted on the *C&C Diagrams* upon changes.

To achieve this in a reusable fashion, the *Reusable ADDs* are formally mapped to *AK Transformation Language Templates*, which are edited with the *AK Transformation Language Editor*. This way, for *Actual ADDs* we can instantiate the corresponding *Transformation Actions* and *Consistency Checking Rules*. Using model-driven techniques they are automatically enacted on the corresponding VbMF *C&C Diagram*.

The binding of templates is realized using Apache Velocity Engine[3]. The parsing and execution of the transformation actions are implemented using Xtend[4], a statically-typed language built on top of Java.

---

[3] http://velocity.apache.org
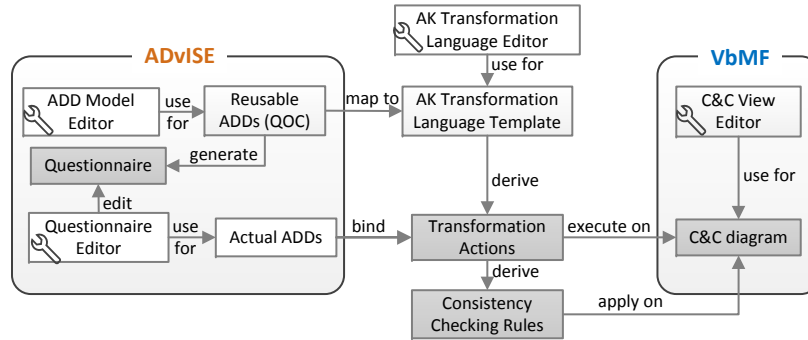[4] http://www.eclipse.org/xtend

Fig. 1. Approach Overview

The AK transformation language and its enactment engine play an important role in our approach for enabling the automated and reusable transformation of ADDs into the design models. In the subsequent parts of this section, we explain the language in detail and its illustrative usage in realistic development circumstances.

3.1 Architectural Knowledge Transformation Language

Essentially, the goal of the AK transformation language is to express the actions that create or update the underlying architectural models (e.g., C&C models) according to the intentions of the software architects reflected by the design decisions. Unlike general model transformation languages (as ATL) our domain-specific language (DSL) is intended to provide simple and comprehensible architecture-specific transformation actions, as well as the structures for grouping, extending and inheriting these actions. Listing 1 presents a formal definition of the AK transformation language in terms of a EBNF like syntax developed using the Xtext DSL framework[5]. Please note that the square brackets in Xtext enable cross-references to other models, in our case, to the elements of the VbMF C&C view. We also use Xtext to generate an Eclipse-based textual editor that can support several useful features such as syntax highlighting, content assist and auto-completion, validation and quick fixes, automated external cross-references resolutions, and so on.

The core of the language consists of basic actions that can be used to create or alter individual elements of the architectural models, for instance, creating a new component, deleting an existing connector, or updating a port. In addition, we introduce special structures, such as `Group`, `Loop`, and `Compound` to support the compositions and extensions of the predefined actions. A `Group` (defined by the grammar rules in lines 41–42) indicates the grouping of a finite set of components that are sub-components of a particular component. For efficiently handling the iteration and application of similar actions to a finite set of elements of the design model, a `Loop` (see line 43–44) can be used. A `Compound` (see line 45–46) represents a structure that embraces multiple actions and even other `Compounds`. Through an extension, a `Compound` can inherit the definition

---

[5] http://www.eclipse.org/Xtext/

of existing `Compounds` and extend the inherited behavior with additional actions. The semantics of a `Compound` is an atomic (i.e., all-or-nothing) sequential execution of its inherited compounds and constituting actions.

```
1  Action:
2    Add | Delete | Update | Group | Loop;
3  Add:
4    AddComponent | AddConnector | AddPort | AddProperty | AddStereotype | AddPrimitive;
5  AddComponent:
6    "add component" name=STRING;
7  AddConnector:
8    "add connector" name=STRING "from" source=[component::Port|FQN] "to" target=[component::
        Port|FQN];
9  AddPort:
10   "add port" name=STRING "kind=" kind=PortKind "to" component=[component::Component|FQN];
11 AddStereotype:
12   "add stereotype" "<<" text=STRING ">>" "to" target=[core::Element|FQN];
13 AddProperty:
14   "add property" name=STRING "type=" type=STRING "value=" value=STRING "to" target=[core::
        Element|FQN];
15 AddPrimitive:
16   "add compound" primitive=[Compound|FQN] name=STRING "("(args+=ID|LIST)+")";
17 Delete:
18   DeleteComponent | DeleteConnector | DeletePort | DeleteProperty | DeleteStereotype;
19 DeleteComponent:
20   "delete component" component=[component::Component|FQN];
21 DeleteConnector:
22   "delete connector" conn=[component::Connector|FQN];
23 DeletePort:
24   "delete port" port=[component::Port|FQN];
25 DeleteProperty:
26   "delete property" property=[component::Property|FQN];
27 DeleteStereotype:
28   "delete stereotype" stereotype=[component::Stereotype|FQN];
29 Update:
30   UpdateComponent | UpdateConnector | UpdatePort | UpdateProperty | UpdateStereotype;
31 UpdateComponent:
32   "update component" component=[component::Component|FQN] "name=" newName=STRING;
33 UpdateConnector:
34   "update connector" conn=[component::Connector|FQN] "name=" newName=STRING;
35 UpdatePort:
36   "update port" port=[component::Port|FQN] ("name=" newName=STRING)? ("kind=" newKind=
        PortKind)?;
37 UpdateProperty:
38   "update property" prop=[component::Property|FQN] ("name=" newName=STRING)? ("type=" newType
        =STRING "value=" newValue=STRING)?;
39 UpdateStereotype:
40   "update stereotype" stereotype=[component::Stereotype|FQN] "text=" newText=STRING;
41 Group:
42   "group" component=[component::Component|FQN] "container" container=[component::Component|
        FQN];
43 Loop:
44   "for" "(" element=ID ":" (params+=FQN)+")" (actions+=Action)+ "end";
45 Compound:
46   "compound" name=ID ("extends" (parent+=[Compound|FQN])*)? spec=Spec;
47 Spec: "("(args+=ID)+")" "{" (actions+=Action)* "}";
48 Import: "import" importedNamespace=FqnWildcard;

50 enum PortKind: provided="PROVIDED"|required="REQUIRED";
51 FQN returns ecore::EString: ID ("." ID)*;
52 FqnWildcard: FQN ".*"?;
53 LIST: ID","ID(","ID)*;
```

Listing 1. Grammar of the AK transformation language

The core actions of the AK transformation language presented in Listing 1 mainly aim at expressing particular changes to individual elements of the corresponding VbMF C&C view. Using the ADvISE tooling, we can associate the options and answers of

a certain ADD model with one or many transformation actions in template form, in order to enable the automation of creating and/or updating of the architectural C&C models. Once the generated questionnaires from the ADD model are answered resulting in concrete decisions, the related actions will be bound to concrete elements of the underlying architectural models.

For instance, suppose in a simple case that a selection of an option in an ADD model leads to the definition of the type of a component which will be indicated by introducing a stereotype. In the example in Listing 2 a new stereotype with the value of the template variability `TypeOfComponent` as its name is created and attached to the component denoted by the template variability `A`.

```
add stereotype <<"${TypeOfComponent}">> to ${A}
```

Listing 2. Example of a parametrized transformation action for adding a stereotype

The binding of the template variables during decision making will result in an executable transformation action, such as the one in the example in Listing 3.

```
add stereotype <<"Remote Proxy">> to example.ServiceProxy
```

Listing 3. Example of a transformation action for adding a stereotype

The execution of the transformation action updates the corresponding C&C diagram as it can be seen in Fig. 2.
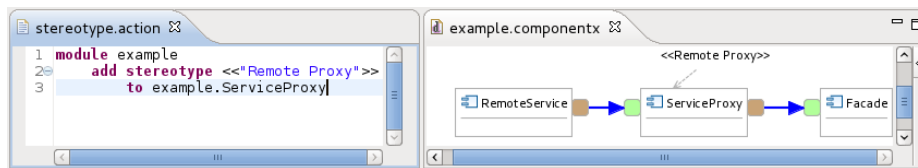
Fig. 2. AK Transformation Language Editor and C&C View Editor

3.2 Recurring Pattern Primitives as Reusable AK Transformations

In the course of decision making, software architects often leverage several recurring architectural elements and structures such as proxies, adapters, gateways, layers, and so forth. The idea of proposing primitives as fundamental elements for describing such recurring design patterns and architectural styles has been investigated by various studies. For example, Zdun and Avgeriou described architectural patterns through a number of recurring architectural primitives in the component-and-connector view using UML profiles [24]. Mehta and Medvidovic developed a framework for defining abstract primitives shared by all architectural styles for composing their elements [18]. In this section, we will describe how to define recurring architectural primitives for modeling certain patterns or styles as action sets, as an example of how to realize reusable AK transformations with our language.

In particular, the expressiveness of our AK transformation language and the support for compositions and extensions through the composite structures mentioned above enable us to define such *recurring architectural primitives* in a reusable and extensible way. In our approach, we specify such recurring primitives using parameterized *action sets* that are based on compounds and can be inherited and extended further. Each action set represents one primitive abstraction that can be used to realize a number of patterns that use this particular primitive as part of their solution (as defined in [24]). The action sets are used via their name and appropriate parameters. In the action set specifications we use variable access in the form `${p-name}` to refer to the parameter *p-name*. When actual ADDs are made, the compound parameters are replaced, which also leads to the variable binding of their primitive actions.

In Listing 4, we present the *indirection* compound. Indirection happens when one or more related "proxy" components receive a message on behalf of one or more "target" components, forward the message to these "targets" and receive results from these "targets" also through the "proxy" components [24]. Proxies and adapters are examples of indirection. The parameters `cv`, `A`, and `B` refer to the target component view, the target component, and the client respectively. The variable `n` will be bounded to the name of the compound "instance" (e.g., Proxy, Adapter, etc.).

```
compound indirection (cv A B) {
  add component "${A}${n}"
  add port "${A}${n}_I1" kind=PROVIDED to ${cv}.${A}${n}
  add port "${A}${n}_I2" kind=REQUIRED to ${cv}.${A}${n}
  add port "${A}_I" kind=REQUIRED to ${cv}.${A}
  add port "${B}_I" kind=PROVIDED to ${cv}.${B}
  add connector "${A}_I_${A}${n}_I1" from ${cv}.${A}.${A}_I to ${cv}.${A}${n}.${A}${n}_I1
  add connector "${A}${n}_I2_${B}_I" from ${cv}.${A}${n}.${A}${n}_I2 to ${cv}.${B}.${B}_I
  add stereotype <<"${n}">> to ${cv}.${A}${n}
}
```

Listing 4. Indirection compound action specification

An example of using the indirection compound is presented in Listing 5.

```
add compound indirection "Proxy" (example Facade Service)
```

Listing 5. Usage example of indirection compound action

This compound action will create a proxy for invoking the component "Service" from the component "Facade". This will happen by replacing the variables `n`, `cv`, `A` and `B` with "Proxy", the C&C view name "example" and the components "Service" and "Facade", respectively. The transformation of the C&C view includes the creation of a component, two connectors, the corresponding ports and a stereotype. The execution of the compound transformation action triggers the execution of its containing actions. In our example, the enactment of Listing 5 will trigger the execution of the primitive actions in Listing 6.

```
  add component "ServiceProxy"
  add port "ServiceProxy_I1" kind=PROVIDED to example.ServiceProxy
  add port "ServiceProxy_I2" kind=REQUIRED to example.ServiceProxy
  ...
```

Listing 6. Binding of primitive actions of indirection compound action

A compound can extend other compounds, and therefore, inherits the corresponding action sets of these compounds, thus increasing reusability of the AK transformations.

3.3 Generation of Consistency Checking Rules

Consistency checking is an important mechanism to ensure the integrity of the design models under consideration. For this, we developed a set of predefined parameterized constraint templates that are related to the basic actions of the AK transformation language shown in Listing 1. As a result, the instantiation and binding of the parameterized constraint templates for each action are performed automatically at the same time and in the same manner as the actions, without requiring any additional effort from the developers and architects. In addition, further constraint templates can be easily formulated in an OCL-like syntax supported by the Eclipse Xpand model validation library [6] and connected to the relevant actions. Again, constraint templates need to be defined only once at the model-level and can then be reused for concrete instantiations of the ADD model. For instance, the following parameterized transformation action of Listing 7 will create a component with name `A`.

```
add component "${A}"
```

Listing 7. Example of a parametrized transformation action to add a component

The resulting C&C model can be checked for its consistency against the related decision `ADD` by the predefined constraint template of Listing 8, which checks that the added component is present in the C&C model.

```
context component::ComponentView ERROR "ADD ${ADD}: Component ${A} does not exist":
  element.typeSelect(component::Component).exists(c|c.name == "${A}");
```

Listing 8. Example of a parametrized consistency checking rule

We have designed and developed respective constraint templates for each AK transformation language element and each architectural primitive defined above. Similar to the AK transformation language, the `${...}` syntax in the constraint rule templates allows to access a variable that is instantiated and bound to particular values of the related actions and models. The outcomes of the instantiation and binding of the parameterized constraint templates are concrete constraints that can be enacted by our model-driven tools. The combination of transformation actions with automatically generated constraints that check that the transformation's semantics are not violated in the C&C diagram, enables us to allow developers and architects to manually change the C&C model. If a manual change violates an architectural decision that has triggered transformation actions, the corresponding constraint checking will signal an error.

4 Case Study and Evaluation

4.1 Case Study

We illustrate the applicability of our approach in the context of an industrial case study on service-based platform integration in the area of industry automation. In our case study, three heterogeneous platforms, a Warehouse Management System—WMS (storage of goods or storage bins into racks via conveyor systems), a Yard Management

---

System—YMS (scheduling, coordination, loading and unloading of trucks), and an Enterprise Resource Planning System—ERP (overall commissioning and handling of goods on an abstract level beyond real storage places) need to provide domain-specific services in an integrated manner. For this, an intermediate integration layer will provide services to operator applications developed on top of it. The integration layer must handle various integration aspects including interface adaptation between the platforms, integration of service-based and non-service-based solutions, routing, enriching, aggregation, splitting, etc. of messages and events, synchronization and concurrency issues, adaptation, and monitoring of events.

ADD Options	AK Transformation Actions

Type of Integrating Component
 	
– None
– Same Interface
– Different Interface
	```add port "${PS}_p1" kind=PROVIDED to ${cv}.${PS}```
	```add port "${IC}_p1" kind=REQUIRED to ${cv}.${IC}```
	
	```#if(${TypeOfComponent} == "None")```
	``` add connector "${IC}_${PS}" from ${cv}.${IC}.${IC}_p1 to ${cv}.${PS}.${PS}_p1```
	``` add stereotype <<"Direct call">> to ${cv}.${IC}_${PS}```
	```#elseif(${TypeOfComponent} == "Same Interface")```
	``` add compound indirection "Proxy" (${PS} ${IC})```
	```#elseif(${TypeOfComponent} == "Different Interface")```
	``` add compound indirection "Adapter" (${PS} ${IC})```
	```#end```

Type of Proxy – Local – Remote	```add stereotype <<"${TypeOfProxy} Proxy">> to ${cv}.${PS}Proxy```
Type of Adapter – Local – Remote	```add stereotype <<"${TypeOfAdapter} Adapter">> to ${cv}.${PS}Adapter```

Heterogeneous systems
 	
– No
– Yes
	```#if(${HeterogeneousSystems} == "Yes")```
	``` add compound integrationAdapter "Integration Adapter" (${PS} ${IC})```
	```#end```

Interchangeability – No – Yes	```add property "${PS}Adapter_Interchangeability" type="Interchangeability" value="${Interchangeability}" to ${cv}.${PS}Adapter```
Adaptation Parameters (String)	```add property "${PS}Adapter_params" type="Parameters" value="${Parameters}" to ${cv}.${PS}Adapter```

Table 1. An excerpt of the service-based platform integration ADD model and its corresponding AK transformation actions

To handle these integration aspects in the platform integration domain, in our previous work, we have introduced an ADD model for resolving architectural design issues related to integration and adaptation, interface design, communication style, and communication flow [15]. We present in Table 1 an excerpt of the ADD model of the platform integration scenario consisting of questions and different alternative options (or answers). This would be normally modeled using ADvISE. Note that the dependencies and constraints between the questions, decisions and options are not present in Table 1 for simplicity reasons. This example assists the decision making on the type of integrat-

ing component between a platform service PS of one of the three platforms in our case study (WMS, YMS and ERP) and a component of the integration layer IC (cv refers to the target C&C view). Along with the ADD model excerpt we present its associated primitive actions and compound actions based on pattern primitives in pattern form, as defined in Section 3. It consists of 6 questions, uses 8 primitive actions and 2 compound actions (integrationAdapter once and indirection twice) and is related to 3 patterns: Proxy (local or remote), Adapter (local or remote) and Integration Adapter. We defined in total 6 basic compounds (indirection, shield, grouping, callback, transformer and router) that are used to describe 21 design patterns in our decision model [15]. The definitions of the compounds are omitted because of the space limitation.

The integration of the Velocity template language with our AK transformation language allows us not only to use placeholders (${...}) but also statements (if, foreach, etc.) which begin with the # character and are parsed by the template engine, but ignored by the AK transformation language editor.

To give an example of the binding of the transformation actions, suppose that the architect opts for a remote proxy as an integrating component between the YMS service *TruckMgmnt* and the integration layer component *OperatorFacade*. The actual ADDs will be reflected in the corresponding C&C view by executing the transformation actions of Listing 9.

```
add port "TruckMgmnt_p1" kind=PROVIDED to example.TruckMgmnt
add port "OperatorFacade_p1" kind=REQUIRED to example.OperatorFacade
add compound indirection "Proxy" (TruckMgmnt OperatorFacade)
add stereotype <<"Remote Proxy">> to example.TruckMgmntProxy
```

Listing 9. Transformation actions example from case study

4.2 Generalizability

Our approach is generic to a large extent. The transformation actions and constraint templates constitute reusable AK assets that can be customized and re-used in various reusable decisions. These templates can be applied for any existing ADD model or ADD documentation because the essential concepts and elements of these models and those in the ADvISE ADD model are almost equivalent. In most cases, the binding between the template variables and the elements of ADD models might need human intervention. That is, in order to properly associate a reusable parameterized action template containing some input parameters with a certain ADD, we need to align the parameters with the corresponding values in the ADD.

The C&C view that is created or updated by enacting the transformation actions contains all the information captured by the corresponding ADDs derived from the ADD meta-model. Nevertheless, the AK transformation language is generic and can be applied to similar C&C models or architectural views on different scenarios as well. Please note that the VbMF C&C view contains very similar elements as other typical C&C views. Therefore, our approach is also applicable for most of existing component models such as UML component diagram with marginal effort for adapting the actions to accommodate new elements. This effort will be added to the effort for editing the AK transformation language templates and constraint templates.

4.3 Reusability

Regardless of the initial efforts for creating the reusable AK transformations, architects will benefit from reduced total efforts in case of recurring ADDs and AK transformations. In our approach, reusability is achieved at various levels. First of all, the AK transformations are edited only once for each ADD model and are afterwards instantiated when actual ADDs are made. This kind of reuse is possible by taking advantage of the benefits of model-driven techniques and template engines. In addition, the use of compound actions that can be extended and inherited increases reusability. Finally, the use of the AK transformation language hides the complex model actions which are embedded in its enactment engine.

4.4 Modeling Effort and Scalability

We present in this section a quantitative evaluation on the modeling effort of using our approach. In particular, we document the number of actions (primitive and compound), primitive actions and model actions that are needed per number of recurring ADDs and for four different ADDs that have been already documented in Section 4.1. For the definition of the action templates 4, 4, 6 and 5 actions had to be edited manually for the reusable decisions Direct Calls, Proxy, Adapter and Integration Adapter respectively. With the use of compound actions we reduced the number of required actions in the last three cases, where 7, 6 and 12 actions were contained in the compound actions *add compound indirection* and *add compound integrationAdapter (extends indirection)*. The number of the actions that are directly applied on the C&C model are 13, 35, 32 and 42 respectively for the four ADDs, which means that without the use of the Action Transformation Language the modeling effort would increase significantly.

This benefit is dramatically increased in case ADDs can be reused. For example, in our case study, the integration of the WMS system currently requires some 35 proxies and adapters, meaning that very similar decisions need to be taken over and over again and, as a consequence, they need to be modeled in C&C diagrams over and over again. Table 2 shows this dramatic increase for the aforementioned decisions, in case of a specific decision outcome being selected 1, 5, 10, 20, 50, and 100 times. Clearly, primitive actions already scale much better in terms of modeling effort than manual change actions in models; reusable actions with compounds offer an additional level of support. In particular, in the cases we study, the modeling effort would increase up to 240% if the compound actions would be replaced by primitive actions and up to 740% if instead of the AK Transformation Language single model actions would be used.

We estimated the scalability of our approach by measuring the performance for binding the action templates variables and transforming the actions into C&C views. We opted to conduct our measurements on a normal desktop machine, as our approach will usually need to run on the local machines of the software architects and designers. The machine for testing had an Intel Quad Core i5 2.53GHz with 8GB of memory running Java VM 1.6 and Eclipse Indigo on Debian Linux. Each measurement is performed 100 times and the resulting time, in milliseconds, is calculated on average. We report only the average, as the deviations calculated were small. Table 3 presents the time needed for the binding of the action template variables and for the transformation of the actions into the C&C views per number of actions, respectively.

Table 2. Modeling Effort for Reusable ADDs

| | | Reusability of ADDs | | | | | | Average increase of modeling effort |
		1	5	10	20	50	100	
Direct Calls	Actions (with compounds)	4	20	40	80	200	400	-
	Primitive Actions	4	20	40	80	200	400	0%
	Model Actions	13	65	130	260	650	1300	225%
Proxy	Actions (with compounds)	4	20	40	80	200	400	-
	Primitive Actions	11	55	110	220	550	1100	175%
	Model Actions	35	175	350	700	1750	3500	775%
Adapter	Actions (with compounds)	6	30	60	120	300	600	-
	Primitive Actions	13	65	130	260	650	1300	117%
	Model Actions	32	160	320	640	1600	3200	433%
Integration Adapter	Actions (with compounds)	5	25	50	100	250	500	-
	Primitive Actions	17	85	170	340	850	1700	240%
	Model Actions	42	210	420	840	2100	4200	740%

Table 3. Performance Measurement

Primitive Actions	5	10	20	50	100	200	500	1000	5000
Binding Time (in msec)	2	3	4	5	6	8	13	21	77
Transformation Time (in msec)	96	102	111	125	147	210	331	671	2748

We can see that the binding and the transformation time increase in a linear manner with respect to the number of actions and remain considerably low even for a big number of actions. In particular, the binding and transformation for 100 actions are accomplished in roughly 6 and 150 ms, for 1000 actions in approximately 20 and 670 ms, and for 5000 actions in about 80 and 2700 ms, respectively. Thus, our approach scales well enough for being integrated in the typical development flow of developers and architects on a typical work station, even for ADDs that create or update large C&C models.

5   Related Work

The documentation of the design rationale, as well as the gathering of Architectural Knowledge (AK) have promoted ADDs to first class citizens in software architecture. For this, many approaches based on decision-capturing templates [23], on ontologies for architectural decisions [13] and decision meta-models [25] have been proposed in the literature. Also, a considerable amount of tools have been developed to ease capturing, managing and sharing of ADDs [21]. These approaches mainly target reasoning on software architectures, capturing and reusing of AK and do not tackle the maintenance and consistency of ADDs with architectural views.

The generation of architectural design views from specifications or other architectural views has been studied extensively in the literature. Pérez-Martínez and Sierra-Alonso [20] use model-to-model transformations to generate component-and-connector architecture models from classes and packages analysis models by using OCL mapping rules. In a different approach [14] variability elements from the problem space are connected to architecture elements in the solution space using a Variability Modeling Language (VML) that provides primitives for referencing and invoking decisions which result in fine-grained or coarse-grained compositions of variable and common

core architectural elements. This approach supports rather the composition than the generation of software architectures as it requires that all architectural elements are predefined. Consistency checking between the different models or the documentation of design rationale are not considered in any of the approaches.

A considerable amount of research has been conducted in relating requirements with software architectures. For example, Kaindl et al. [9] suggest that with the use of model-driven approaches we can map requirements to architectural design and Grunbacher et al. [5] introduce the mapping from requirements to intermediate models that are closer to software architecture. A different approach by van Lamsweerde et al. [12] derives software architectures from the formal specifications of a system goal model (KAOS) using transformation rules and refines the architectures incrementally using patterns that satisfy quality of service goals like availability and fault tolerance. In the aforementioned approaches, although the transformations are done automatically, the mapping has to be done manually and is not reusable. Another disadvantage compared to our approach is that the rationale that led from the requirements to the architectural views is not documented. In our work we assume that architectural decision making follows the collection of requirements and precedes the design of software architectures and set our focus on the linking of reusable ADDs to C&C models.

Our approach is not the first one to relate ADDs to software architectures. The problem of inconsistencies between ADDs and software architectures that cause design knowledge vaporization has been discussed before by Choi et al. [2]. For this, they propose to make ADDs more explicit by introducing a meta-model for relating decisions with architectural elements and a decision constraint graph for representing decision relationships and studying decision change impact analysis. Compared to our approach, this approach demands that most of the work is done manually: decision making, architectural design and change propagation during software evolution. STREAM-ADD [4] also relates architectural decisions documented in decision templates with requirements and architectural models generated from these requirements. This approach focusses rather on the integration of systematic documentation of structural and technological decisions with requirements and architectural models than on the consistency checking between decisions and designs.

Traceability links between decision models and architecture models have been used extensively in the literature. Capilla et al. [1] introduce fine-grained traceability links between design decisions and other software artifacts. Knemann and Zimmermann [10] establish links between design decisions and design models in model-based software development in order to support architectural knowledge documentation and reuse, as well as to check consistency. Mirakhorli and Cleland-Huang [19] introduce the TTIM approach that provides a reusable infrastructure for tracing architecture tactics to designs used to trace from tactic-related design decisions to architecture components in which a decision is realized. Also, most of the approaches require significant amount of manual work for the establishment of the traceability links, which can be in our approach automated for recurring ADDs from the mapping of the ADDs to transformation actions and to constraints at template level. Apart from that, none of these approaches target the reusability of these links between ADDs and architectural views, nor do they tackle the complexity of big numbers of reusable ADDs.

6 Conclusions

We present a novel approach that provides reusable and extensible transformation actions and consistency checking rules for (semi-)automatically mapping of the design rationale and knowledge reflected by ADDs onto architectural component models. In particular, our approach introduces an AK transformation language for specifying reusable actions that need to be enacted to automatically create or update the underlying architectural models with respect to particular ADDs. The transformation language provides basic actions for updating individual model elements, as well as expressive composite structures for describing actions applied in a set of elements such as compounds and loops. This enables us, for instance, to define recurring architectural primitives, e.g., to realize reusable specifications for architectural patterns or styles in the transformation language. In addition, our approach supports the specification and automatic generation of consistency checking rules to make sure no manual changes of the component models violate the ADDs. The application of our approach in an industrial case study shows that our approach is applicable in a realistic scenario. Our evaluation illustrates the benefits of our approach in terms of potential modeling effort reduction, as well as its scalability in a typical work environment, even for large model sizes. As discussed, the use of a template engine and model-driven techniques, as well as the support for inheritance and extension in the transformation language significantly enhance its reusability and extensibility. In our future work, we plan to study repair actions for resolving inconsistencies between reusable ADDs and component views, as well as the possibility for bidirectional transformations, i.e., also from component views onto decisions.

References

1. Capilla, R., Zimmermann, O., Zdun, U., Avgeriou, P., Küster, J.M.: An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle. In: 5th European Conf. in Software Architecture (ECSA), Essen, Germany. pp. 303–318. Springer (2011)
2. Choi, Y., Choi, H., Oh, M.: An architectural design decision-centric approach to architectural evolution. In: 11th Int'l Conf. on Advanced Communication Technology (ICACT), Gangwon-Do, South Korea. pp. 417–422. IEEE Press (2009)
3. Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R.: Documenting Software Architectures: Views and Beyond. Pearson Education (2002)
4. Dermeval, D., Pimentel, J., Silva, C.T.L.L., Castro, J., Santos, E., Guedes, G., Lucena, M., Finkelstein, A.: STREAM-ADD - Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process. In: 36th Annual IEEE Computer Software and Applications Conf. (COMPSAC), Izmir, Turkey. pp. 602–611. IEEE Comp. Soc. (2012)
5. Grunbacher, P., Egyed, A., Medvidovic, N.: Reconciling Software Requirements and Architectures with Intermediate Models. Softw. Syst. Model. 3(3), 235–253 (2003)
6. Harrison, N.B., Avgeriou, P., Zdun, U.: Using Patterns to Capture Architectural Decisions. IEEE Softw. 24(4), 38–45 (2007)
7. ISO: ISO/IEC CD1 42010, Systems and software engineering — Architecture description (2010)

8. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: 5th Working IEEE/IFIP Conf. on Software Architecture (WICSA), Pittsburgh, PA, USA. pp. 109–120. IEEE Comp. Soc. (2005)
9. Kaindl, H., Falb, J.: Can We Transform Requirements into Architecture? In: 3rd Int'l Conf. on Software Engineering Advances (ICSEA), Sliema, Malta. pp. 91–96. IEEE (2008)
10. Könemann, P., Zimmermann, O.: Linking Design Decisions to Design Models in Model-Based Software Development. In: 4th European Conf. in Software Architecture (ECSA), Copenhagen, Denmark. pp. 246–262. Springer (2010)
11. Kruchten, P., Capilla, R., Dueñas, J.C.: The Decision View's Role in Software Architecture Practice. IEEE Softw. 26(2), 36–42 (Mar 2009)
12. van Lamsweerde, A.: From System Goals to Software Architecture. In: Bernardo, M., Inverardi, P. (eds.) School on Formal Methods. vol. LNCS 2804, pp. 25–43. Springer (2003)
13. Lee, L., Kruchten, P.: Capturing Software Architectural Design Decisions. In: 2007 Canadian Conf. on Electrical and Computer Engineering. pp. 686–689. IEEE Comp. Soc. (2007)
14. Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language Support for Managing Variability in Architectural Models. In: Software Composition. pp. 36–51 (2008)
15. Lytra, I., Sobernig, S., Zdun, U.: Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In: Joint 10th Working IEEE/IFIP Conf. on Software Architecture & 6th European Conf. on Software Architecture (WICSA/ECSA), Helsinki, Finland. IEEE Comp. Soc. (2012)
16. Lytra, I., Tran, H., Zdun, U.: Constraint-based consistency checking between design decisions and component models for supporting software architecture evolution. In: 16th European Conf. on Software Maintenance and Reengineering (CSMR), Szeged, Hungary. pp. 287–296. Springer (2012)
17. MacLean, A., Young, R., Bellotti, V., Moran, T.: Questions, Options, and Criteria: Elements of Design Space Analysis. Human-Computer Interaction 6, 201–2502 (1991)
18. Mehta, N.R., Medvidovic, N.: Composing architectural styles from architectural primitives. In: 9th European Software Engineering Conf. held jointly with 11th ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering (ESEC/FSE-11), Helsinki, Finland. pp. 347–350. ACM (2003)
19. Mirakhorli, M., Cleland-Huang, J.: Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In: 27th IEEE Int'l Conf. on Software Maintenance (ICSM), Williamsburg, VA, USA. pp. 123–132. IEEE (2011)
20. Pérez-Martínez, J.E., Sierra-Alonso, A.: From Analysis Model to Software Architecture: A PIM2PIM Mapping. In: Model Driven Architecture - Foundations and Applications (ECMDA-FA), Biblao, Spain. pp. 25–39 (2006)
21. Shahin, M., Liang, P., Khayyambashi, M.R.: Architectural design decision: Existing models and tools. In: Joint Working IEEE/IFIP Conf. on Software Architecture and European Conf. on Software Architecture (WICSA/ECSA), Cambridge, UK. pp. 293–296. IEEE (2009)
22. Tran, H., Zdun, U., Dustdar, S.: View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. In: Int'l Conf. Business Process and Services Computing (BPSC). pp. 105–124. Lecture Notes in Informatics (LNI) (2007)
23. Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Softw. 22(2), 19–27 (2005)
24. Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. In: 20th ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). pp. 133–146. ACM Press (2005)
25. Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., Schuster, N.: Reusable architectural decision models for enterprise application development. In: 3rd Int'l Conf. on Quality of Software Architectures (QoSA), Medford, MA, USA. pp. 15–32. Springer (2007)