Harmonizing Architectural Decisions with Component View Models using Reusable Architectural Knowledge Transformations and Constraints

Ioanna Lytra, Huy Tran, Uwe Zdun

Research Group Software Architecture
University of Vienna, Austria
Email: firstname.lastname@univie.ac.at

Abstract

Architectural design decisions (ADDs) have been used in recent years for capturing design rationale and documenting architectural knowledge (AK). However, various architectural design views still provide the most common means for describing and communicating architectural design. The evolution of software systems requires that both ADDs and architectural design views are documented and maintained, which is a tedious and time-consuming task in the long run. Also, in lack of a systematic and automated support for bridging between ADDs and architectural design views, decisions and designs tend to become inconsistent over time. In our proposal, we introduce a reusable AK transformation language for supporting the automated transformation of reusable AK knowledge to component-and-connector models, the architectural design view used most commonly today. In addition, reusable consistency checking rules verify the consistency between decisions and designs. We evaluate our approach in an industrial case study and show that it offers high reusability, provides automation, and can, in principle, deal with large numbers of recurring decisions.

Keywords: architectural decisions; architectural design; architectural knowledge; AK transformation language; consistency checking

1. Introduction

From the various architectural views [8, 14, 42] used to document software architectures the component-and-connector (C&C) view is often considered the one that contains the most significant architectural information [8]. In many enterprises today, software architecture is mainly documented using component-and-connector diagrams, usually in an informal or semi-formal fashion (e.g., as box-and-line diagrams). However, architectural documentations based only on components and connectors have many disadvantages, such as limited reusability of and reasoning about architectural knowledge (AK), and lack of sharing support of this knowledge among stakeholders [3]. Therefore, the software architecture community has proposed a new perspective on software architecture through the explicit documentation of architectural design decisions (ADDs) [16]. The actual solution structure, or architectural design, is merely a reflection of those design decisions.

Several approaches have been proposed for capturing architectural design decisions. Akerman and Tyree defined a rich decision capturing template [42]. Kruchten et al. presented an ontology for architectural decisions that defines types of architectural decisions, dependencies between them, and a decision lifecycle [21]. Zimmermann et al. suggested a meta-model for decision capturing and modeling [45]. To minimize the effort of documenting architectural decisions, approaches for reusable architectural decision modeling [46] and using design patterns as a basis for documenting reusable ADDs (see, e.g., [12]) have been proposed.

ADDs play a crucial role not only during architectural design but also during development, evolution, reuse, and integration of software architectures [16]. In practice, the ADDs frequently are neither maintained nor synchronized over time with the corresponding C&C diagrams (or other design views), that is decisions and designs drift apart over time [16]. This leads potentially to the loss of architectural knowledge, a phenomenon which is known as *architectural knowledge vaporization* [12, 16]. Lacking of an adequate harmonization between software architectures and design decisions often leads to more severe consequences [7].

Until now, the establishment and preservation of consistency between decisions and designs have not been addressed or supported systematically. That is, so far there is no formal mapping or automated translation between ADDs and design views. Thus, the task of harmonizing decisions and designs remains ad hoc and tedious. Making matters worse, the actual documentation of ADDs is also time consuming [46], especially for kinds of ADDs that need to be made repeatedly throughout a software design process, such as many of the ADDs documented in [23, 24].

In our previous work [25], we partially addressed the problem of bridging ADDs and designs. This approach 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 for each ADD separately, making this approach inefficient for large numbers of ADDs and/or complex design models. Moreover, in reality, several ADDs can be reused in different software design contexts and domains [45]. Thus, taking advantage of the reusability of such recurring decisions would significantly enhance the productivity in creating and maintaining the formal mappings between the decisions and the designs. Unlike our previous work [25], we now set the focus on reusable architectural knowledge.

The approach presented in this paper aims at addressing the aforementioned challenges. In particular, our proposal introduces an architectural knowledge transformation language. This domain-specific language supports the specification of primitive and complex actions whose enactment leads to automatic updates of design models (i.e., C&C diagrams) based on the corresponding documented ADDs. The outcomes of a certain decision can be expressed either by executing individual actions, such as the creation of new elements or the deletion, modification, or grouping of existing elements in the C&C diagrams, or by executing composite actions (e.g., for capturing reusable pattern-based ADDs) that can be formally modeled through certain architectural primitives [44] or other composite actions. To ensure consistency between ADDs and C&C views, constraints are automatically generated from the execution of transformation actions. That is, the constraints ensure, for instance, that manual changes in C&C views do not violate the ADDs.

In our approach, we exploit template-based generation rules and model-driven techniques for automatically instantiating and enacting the actions, as well as generating corresponding constraints. The linking of reusable ADDs to reusable actions and constraints (in template form) offers higher reusability and automation and results in less complexity and modeling effort for software architects. The reusability is achieved here (1) through the automatic derivation of parts of the C&C diagrams and consistency checking rules using model-driven templates and (2) by reusing common abstractions shared among common design patterns (see [44]).

In order to demonstrate our approach, we integrated two 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. Using this prototypical implementation we evaluated our proposal in the context of an industrial case study from the warehouse automation area, in terms of reusability and modeling effort.

Our approach solely addresses C&C views at the moment, but it is possible to integrate other architectural design aspects with architectural decisions. The incorporation of other views is supported in VbMF via its view integration techniques [40]. Illustrative examples of different aspects that are integrated using VbMF include data [29], human [13], event and runtime monitoring [33], and compliance [41]. Investigating architecture-specific information leveraging the integration of different VbMF views is beyond the scope of this paper and part of our future work.

This article is an extension of our conference publication [26] with following significant additions and improvements: (a) a more complete list of compound transformation actions and reusable constraints has been included along with further elaborations, (b) the case study and evaluations have been extended, and (c) the related work has been discussed further and compared to our contributions.

The remainder of the paper is structured as follows. First, in Section 2 we introduce some basic concepts and present briefly the ADvISE and VbMF tools and their meta-models. In Section 3 we describe the details of our approach, namely we present the reusable AK transformations and consistency checking rules along with some illustrative examples. We apply our approach in an industrial case study in Section 4 and discuss the evaluation results in Section 5. Finally, we compare to the related works in Section 6 and summarize the key contributions of our work in Section 7.

2. Background

The main focus of our study is the harmonization of architectural design decisions and architectural models. This involves the tasks of making and documenting design decisions and creating and manipulating architectural models. Thus, in this section we briefly present ADvISE and VbMF, the two tools that support

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

[2] `http://swa.univie.ac.at/View-based_Modeling_Framework`

these tasks and are leveraged to illustrate our approach. Before that, we introduce some key concepts that we will use throughout the paper.

2.1. Basic Concepts

According to the ISO/IEC/IEEE 42010 standard [14] an ADD affects various architectural elements, pertains to or raises concerns, and is justified by architecture rationale. ADDs capture knowledge that may concern a software system as a whole, or one or more components of a software architecture. Rather than documenting the structure of software systems (e.g., components and connectors) ADDs entail the design rationale that led to that structure (e.g., justification about the ADDs that were made and the architectural alternatives not chosen).

For capturing reusable ADDs, architectural decision modeling has been introduced in the existing literature (refer to [36] for a comparison of existing architectural decision models and tools). The advantage of these architectural decision models is that they are reusable and can thus provide guidance for architectural decision making activities, whenever recurring design issues emerge. Reusable architectural models share common concepts with patterns (see [12]) which give proven solutions to recurring problems that arise in particular contexts and domains [35]. The relationship between architectural patterns and reusable decision models is eventually synergetic [45], for instance, reusable decision models can be used for pattern selection. In our approach, we use mainly pattern-based reusable decision models to capture ADDs and relate them to reusable AK transformations.

A richer taxonomy in the areas of software architecture and architectural design can be found elsewhere, for instance in [16, 19, 38, 42].

2.2. Architectural Design Decision Support Framework

The Architectural Design Decision Support Framework (ADvISE) provides tool support for modeling of reusable ADDs using Questions, Options, and Criteria (QOC) [27] and architectural decision making. In particular, ADvISE assists the architectural decision making process by introducing for each design issue a set of questions along with potential options related to specific criteria, answers, and pattern-based solutions. In some cases a question may expect a free-text answer. The provided tooling also supports the specification of dependencies and constraints among decisions, questions, and solutions. These dependencies and constraints are then used by ADvISE to determine the logical flow of making decisions for particular architectural design problems.

ADvISE introduces a reusable ADD meta-model (see Figure 1) for the design space of certain application domains (e.g., service-oriented integration of industrial automation platforms in our case study below), consisting of decisions, questions, options, criteria, solutions, design patterns, and relationships among them. Examples of relationships are that a question triggers a next decision or an option is incompatible with another option. For each option, related criteria are positively or negatively evaluated and the selection of a specific option may lead to a suggested solution (pattern-based or not). 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 resolve potential constraints and dependencies, recommend best-fitting design solutions and patterns, and generate ADD documentations.

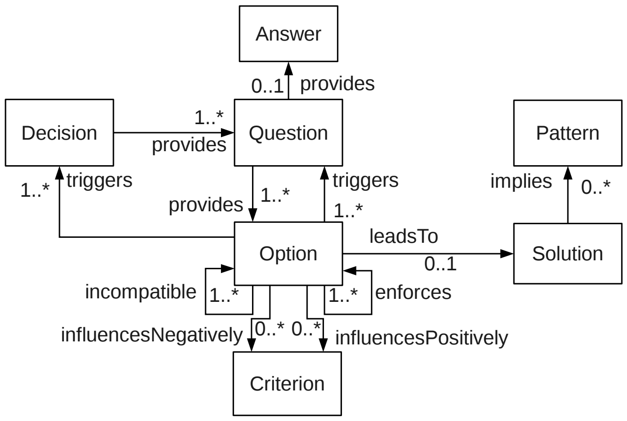

Figure 1: Reusable ADD meta-model

Our approach in this paper aims at devising an architectural knowledge transformation framework (c.f. 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 actual design decisions into corresponding design models and generating constraints for consistency checking between them.

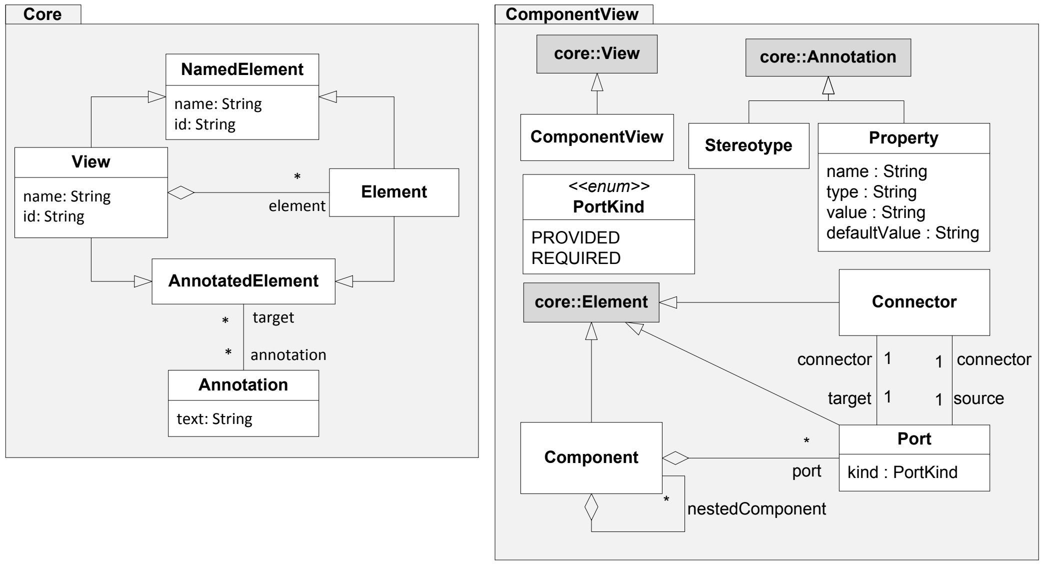

Figure 2: VbMF's Core and Component Model

2.3. View-based Modeling Framework

The View-based Modeling Framework (VbMF) [39] implements a model-driven, architectural view model. That is, it leverages the notion of view models for describing various aspects of the software systems at different abstraction levels and model-driven development techniques for generating code and configurations out of the view models [39]. The Core model shown in Figure 2 is the basis for defining various view models as well as integrating the view models using name-based matching techniques [40].

Among other views, VbMF provides a high-level component view model (see Figure 2)—similar to a typical C&C model such as UML component model—for representing essential architectural design elements such as components, ports, connectors, annotations, 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 use the high-level 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. Nevertheless, we note that the Core model is introduced in VbMF mainly for integrating various view models, which is not the main focus of this approach. Therefore, any other C&C models, for instance, UML component model [10], can be easily adapted to be used in our approach with reasonable extra effort.

3. Reusable AK Transformations

3.1. Approach Overview

We depict in Figure 3 an overview of our approach in terms of the involved stakeholders, tool suites, tools and artifacts along with their relationships. Tools are indicated with rounded rectangles while manually edited and automatically generated artifacts are denoted by rectangles having light-gray and dark-gray background, respectively.

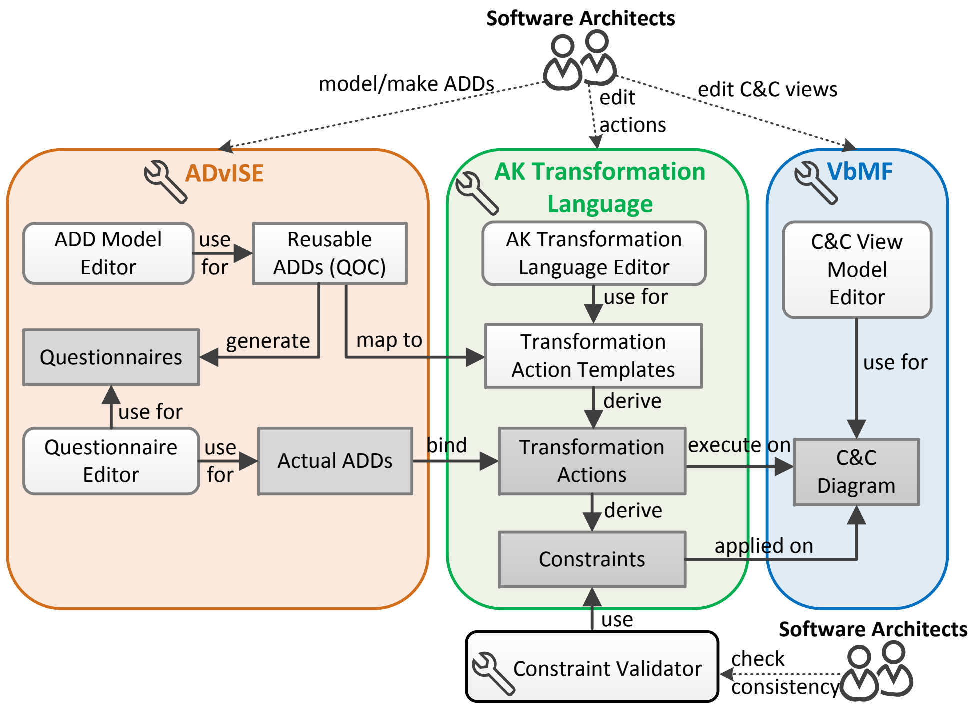

Figure 3: Approach Overview

The *ADD Model Editor* is a graphical editor provided by ADvISE and can be used by software architects to create the reusable ADD models (i.e., *Reusable*

ADDs in Figure 3). This step is intended to be performed by senior software architects with significant expertise and experience in software architecture design, possibly together with domain experts. The advantage of the reusable ADD models is that they have to be created only once per application domain. Based on these ADD models, ADvISE will automatically generate *Questionnaires* that can be used for making actual architectural decisions (see *Actual ADDs*). With the support of the *Questionnaire Editor* tool, software architects—now in the role of end-users—can navigate the provided design space (e.g., by selecting question options, viewing follow-on questions and decisions, etc.) in order to make and document actual design decisions. Questionnaires containing ADDs instantiated from the same reusable ADD model can be generated and answered (but also modified) multiple times in similar design situations.

For manipulating *C&C Diagrams*, VbMF provides a graphical *C&C View Model Editor*. In order to assist software architects and developers in *greenfield* scenarios (i.e., where a completely new C&C model needs to be derived from decisions that have been made) our approach supports generating initial instances of the *C&C Diagrams* automatically from ADDs through the execution of *Transformation Actions*. *Transformation Actions* can be also applied for updating existing view models. In a typical development scenario, design models can be changed independently of the documented decisions. This is also the case in VbMF where *C&C Diagrams* can be manually manipulated. In this case, in order to ensure that changes in the C&C models do not invalidate existing ADDs, our approach allows the generation of *Constraints* based on the corresponding *Transformation Actions*. The constraint validation is achieved using an external *Constraint Validator*, which is integrated with our AK transformation language tooling. The validation of the *C&C Diagrams* against these constraints can be enacted manually by the software architects or automatically upon changes.

To achieve the generation of transformation actions and constraints in a reusable fashion, the *Reusable ADDs* are formally mapped to *Transformation Language Templates*, which can be edited with the *AK Transformation Language Editor*. This way, for *Actual ADDs* we can instantiate the corresponding *Transformation Actions* and subsequently the *Constraints*. Using model-driven techniques, the transformation actions are automatically enacted on the corresponding VbMF C&C diagram.

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 design models. In the subsequent parts of this section, we elaborate the language and its illustrative usage in realistic development circumstances.

3.2. Approach Steps

We illustrate in Figure 4 the steps of our proposal that shall be performed by software architects, as well as the tools that will be used in each step and phase. In the first phase of the preparation of the reusable ADDs and their mappings to C&C views, the software architect uses the *ADD Model Editor* to create the reusable ADD models (i.e., define questions, options and criteria that will provide architectural decision guidance through corresponding questionnaires) based on the ADD meta-model. Then the software architect creates the transformation action templates that will transform actual ADDs into C&C view elements and consistency checking rules between decisions and designs respectively. We note that the aforementioned steps (1–2, Figure 4(a)) need to be done only once to define an architectural design space for particular application domains or contexts and are supposed to be performed by experienced software architects, as mentioned before. Given such a design space, the following steps (1–4, Figure 4(b)) will be carried out by software architects at design time who will make and document appropriate architectural design decisions and create the corresponding design models.

The software architect follows ADvISE's guidance to make and document ADDs which will trigger the instantiation of the corresponding transformation actions and constraints. The transformation actions will then be enacted using our tools to generate initial C&C views. Finally, the software architect will be able to view and edit the C&C view(s) with the VbMF editor and validate the aforementioned constraints applied on the C&C view(s). These steps are supposed to be performed repeatedly in one or different projects and are based on the configuration that has been done by the experienced software architects in the Steps 1 and 2 of Figure 4(a). In the following subsections, we focus on the main contribution of this paper, namely, the AK Transformation Language; further instructions how to use the rest of the tools can be found at the corresponding websites. While AK Transformation Language constitutes the central element of our approach and needs to be managed by the software architects, the knowledge of a template language is essential for editing the transformation action templates which correspond to the reusable ADDs, as well as the knowledge of an OCL-like language in which the constraints are edited.

3.3. Architectural Knowledge (AK) Transformation Language

In the following subsections we present the main goal and central concepts of the AK transformation language and discuss some usage examples.

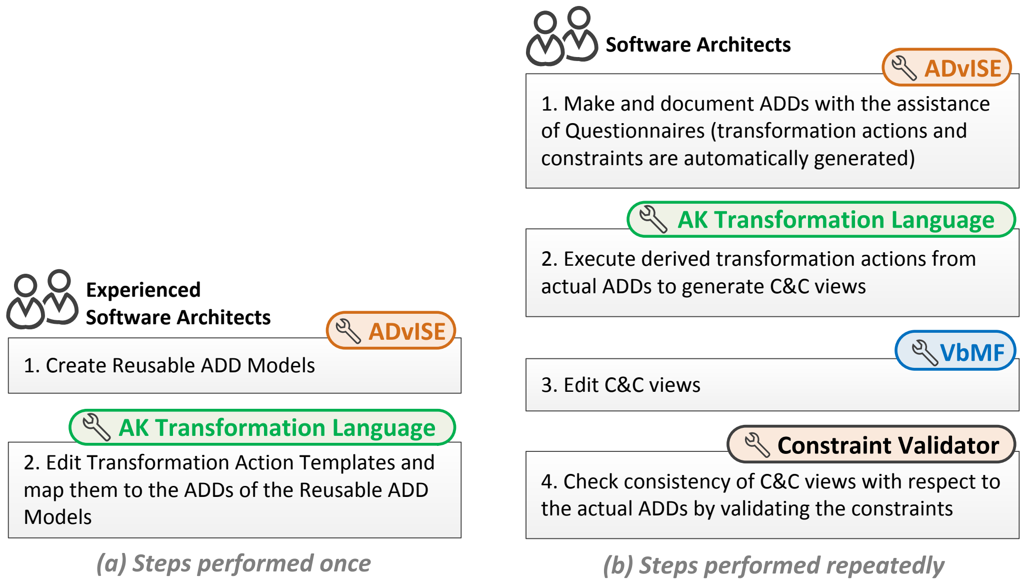
Experienced Software Architects
ADvISE
1. Create Reusable ADD Models
AK Transformation Language
2. Edit Transformation Action Templates and map them to the ADDs of the Reusable ADD Models
(a) Steps performed once
Software Architects
ADvISE
1. Make and document ADDs with the assistance of Questionnaires (transformation actions and constraints are automatically generated)
AK Transformation Language
2. Execute derived transformation actions from actual ADDs to generate C&C views
VbMF
3. Edit C&C views
Constraint Validator
4. Check consistency of C&C views with respect to the actual ADDs by validating the constraints
(b) Steps performed repeatedly

Figure 4: Approach Steps Performed by Software Architects

3.3.1. Main Concepts

Essentially, the goal of the AK transformation language is to support the harmonization of ADDs and architectural design models via transformation actions that, when being enacted, shall create or update corresponding design models (e.g., C&C models) with respect to the intentions of the software architects reflected in the design decision models. One simple example is that making an architectural design decision on using a proxy between a client and server will lead to the creation of a new "*proxy component*" along with its properties, as well as appropriate connectors with the existing "*client*" and "*server*" components. In case none of them exists (e.g., in a "*greenfield*" scenario), these components must be created and wired accordingly.

There are many existing languages and techniques, for example, QVT[3], ATL[4], ETL[5], Xtend[6], to name but a few, that target generic and abstract concepts and elements in model transformations. They provide expressive languages and pow-

[3]http://www.omg.org/spec/QVT
[4]http://www.eclipse.org/atl
[5]http://www.eclipse.org/epsilon/doc/etl
[6]http://www.eclipse.org/xtend

erful transformation engines for both endogenous and exogenous model transformations [31].

Yet, these approaches provide no adequate abstractions and concepts for architecture-specific transformations. As a consequence, it will be inefficient to use such full-blown languages for architectural knowledge transformations as software architects must get familiar with several model transformation concepts instead of solely focusing on their particular domain of expertise. Therefore, we aim to bring the bests of both worlds by developing AK transformation language as a domain-specific language (DSL) in order to provide simple but succinct and comprehensible concepts and elements for describing and formulating the transformation of architectural design decisions into design models in a reusable manner.

We use the meta-model of Figure 5 to illustrate the main elements of the AK transformation language. The fundamental concept of our DSL is `Action` that represents the potential effects on an architectural design model such as adding (`Add`), deleting (`Delete`), and updating (`Update`) individual or a set of architectural elements such as components, connectors, ports, and their properties (e.g., `AddComponent`, `DeleteConnector`, `UpdatePort`, etc.). The enactment of an `Action` will lead to changes in the corresponding architectural design model such as the creation of a new component, the deletion of an existing connector, or the modification of a port, and so on. Apart from that, the user of AK transformation language can define a set of components as sub-components of another component (`Group`) and refine a high level component onto one or more low-level components (`Refine`). A `Compound` is, firstly, used to represent a composite construct containing multiple actions. Secondly, it can inherit the definitions of existing `Compounds`, and therefore, reduce redundancy and duplicated efforts. The semantics of a `Compound` ensures atomic (i.e., all-or-nothing) sequential execution of its inherited compounds and constituting actions. In addition, the AK transformation language allows the execution of transformation actions under conditions (`Condition`). Finally, a `ForLoop` and a `WhileLoop` can be used in order to define one or more actions that are performed over a predefined set of design elements or under certain boolean constraints respectively.

As we mentioned above, the AK transformation language aims at aiding software architects in focusing on transforming AK into design models instead of being overwhelmed by model transformation concepts and techniques. Therefore, the language is designed to mainly emphasize on architecture-specific notions. The actions and constructs of the AK transformation language aim at expressing tentative changes to the elements of the corresponding VbMF C&C view and

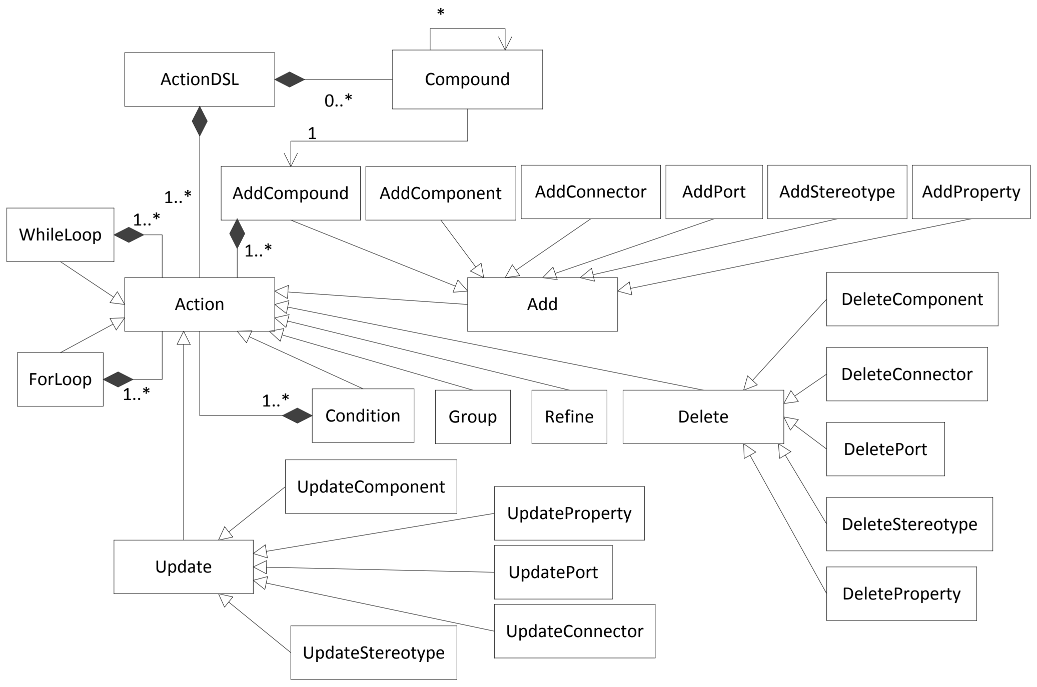

Figure 5: AK Transformation Language meta-model

can be used either independently or along with ADvISE. In the latter case, the software architects, in the first and second steps of creating the design space (see Figure 4(a)), need to relate the options and answers of a certain ADD model with one or many transformation actions in the design decision templates. This enables the automation of creating and/or updating of the architectural C&C models when actual ADDs are made (in the Step 1–4 shown in Figure 4(b)), i.e., the templates are bound to concrete values. That is, once the generated questionnaires from the ADD model are answered resulting in concrete design decisions, the related actions will also be instantiated and bound to the concrete elements of the corresponding architectural models.

In the scope of our work, we developed the AK transformation language using the Eclipse Xtext framework[7]. The biggest and pragmatic advantage of using Xtext framework is its integration with Eclipse's modeling and development environment, which is widely used and supported in practice. Additionally, Xtext

[7]`http://www.eclipse.org/Xtext`

can generate Eclipse-based textual editors that deliver several powerful features such as syntax highlighting, content assistance and auto-completion, validation and quick fixes, automated external cross-references resolutions, and so on. Moreover, Xtext allows for partially dealing with some contextual aspects such as typing (by using Xtext type system framework[8]), uniqueness of identifiers (by using the notion of namespaces and scoping), and visibility (by scoping).

Please refer to Appendix A for a complete specification of the AK Transformation Language using the Xtext grammar.

3.3.2. Usage Examples

Let us use the reusable ADD to introduce a *remote proxy* for calling a remote service from an application's component. This task will require a new component being created and annotated as "Remote Proxy" to denote an invocation of the remote service from the application's component. We illustrate the tentative set of transformation actions in Listing 1, in which the parameters to be replaced are indicated with `${}`. These actions are only valid when they are bound to concrete C&C view elements. Therefore, the use of this set of actions requires that following information is provided: the name of the C&C view model (`cv`), the name of the remote service (`A`) and the name of the component which calls this remote service (`B`).

```
add component "${A}Proxy"
add stereotype <<"Remote Proxy">> to ${cv}.${A}Proxy
add port "${A}Proxy_p1" kind=REQUIRED to ${cv}.${A}Proxy
add port "${A}Proxy_p2" kind=PROVIDED to ${cv}.${A}Proxy
add port "${A}_p" kind=PROVIDED to ${cv}.${A}
add port "${B}_p" kind=REQUIRED to ${cv}.${B}
add connector "${A}Proxy_${A}" from ${cv}.${A}Proxy.${A}Proxy_p1 to ${cv}.${A}.${A}_p
add connector "${B}_${A}Proxy" from ${cv}.${B}.${B}_p to ${cv}.${A}Proxy.${A}Proxy_p2
```

Listing 1: Example of Parameterized Transformation Actions for Creating a Proxy in Template Form

When the software architects make concrete choices in the design decisions, the corresponding architectural models, which either exist or are newly created, are determined. As a result, the parameters of the aforementioned set of transformation actions can be bound to concrete values, for instance, `cv` $\sim$ `model`, `A` $\sim$ `Service`, and `B` $\sim$ `CompA`. The binding (denoted above as $\sim$) will be done automatically by our tools, resulting in the executable transformation actions shown in Listing 2.

[8]https://code.google.com/a/eclipselabs.org/p/xtext-typesystem

```
add component "ServiceProxy"
add stereotype <<"Remote Proxy">> to model.ServiceProxy
add port "ServiceProxy_p1" kind=REQUIRED to model.ServiceProxy
add port "ServiceProxy_p2" kind=PROVIDED to model.ServiceProxy
add port "Service_p" kind=PROVIDED to model.Service
add port "CompA_p" kind=REQUIRED to model.CompA
add connector "ServiceProxy_Service" from model.ServiceProxy.ServiceProxy_p1 to model.
     Service.Service_p
add connector "CompA_ServiceProxy" from model.CompA.CompA_p to model.ServiceProxy.
     ServiceProxy_p2
```

Listing 2: Example of Transformation Actions for Creating a Proxy

The transformation actions presented above add new elements in the C&C view. Nonetheless, updating and removing existing components, connectors, etc. in a C&C view may be necessary when certain ADDs are revised or removed. Listing 3 illustrates three "`Delete`" actions that reverse the effects of the transformation actions shown in Listing 2 and an "`Update`" action that changes the name of the port `Service_p` of the component `Service`.

```
delete component model.ServiceProxy
delete port model.Service.Service_p
delete port model.CompA.CompA_p
...
update port model.Service.Service_p name="IService"
```

Listing 3: Examples of Delete and Update Actions

The use of templates enables the reuse of transformation actions whenever ADDs from the reusable ADD model are made. The enactment of transformation actions will update the corresponding C&C diagram. This is achieved by performing model-to-model transformations from single transformation actions into model actions that modify the underlying C&C view models. Compound actions, conditionals and loops are translated into their containing single actions that are afterwards enacted on the C&C view. Thus, the AK transformation language offers simplicity and reusability by providing adequate abstractions for hiding unnecessary details of model transformation techniques, and therefore, helps software architects to better focus on architecture-specific concepts. In our example, the execution of the actions of Listing 2 will lead to the C&C view of Figure 6.

3.4. Recurring Pattern Primitives as Reusable AK Transformations

In the course of design decision making, software architects often deal with several recurring architectural elements and structures such as proxies, adapters, gateways, layers, and so forth. The idea of proposing primitive abstractions as fundamental elements for describing such recurring design patterns and architectural

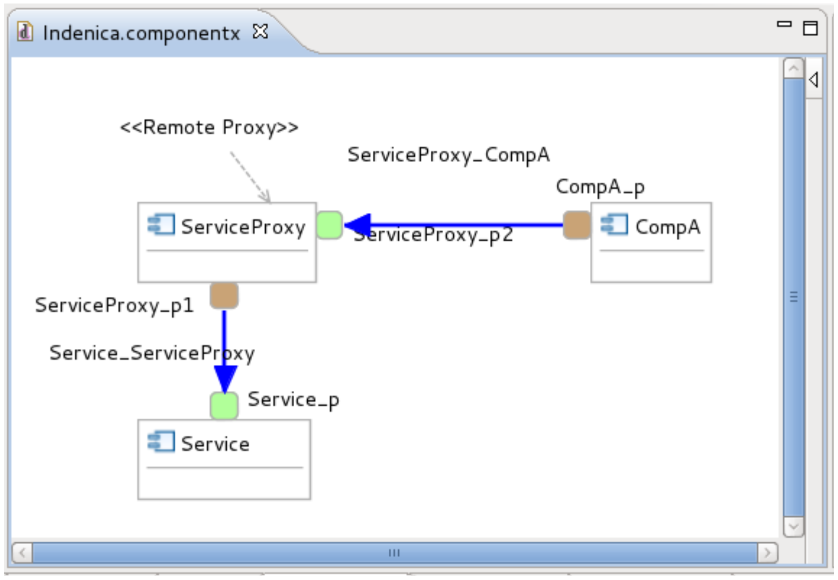

Figure 6: C&C View Editor

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 [44]. Mehta and Medvidovic developed a framework for defining abstract primitives shared by all architectural styles for composing their elements [30]. In our AK transformation language, we provide support for expressing such primitive abstractions. In this section, we will describe how our approach can be used to define and use recurring architectural primitives for modeling certain patterns or styles as compounds.

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 *recurring architectural primitives* in a reusable and extensible way. In our approach, we specify such recurring primitives using parameterized sets of actions that are based on compounds and can be inherited and extended further. Each compound 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 [44]). The compounds are used via their name and appropriate parameters. In a compound specification, we use variable access in the form `${p-name}` to refer to the parameter *p-name*. When concrete compounds are "instantiated" via the command type `add compound`, the compound parameters are replaced with the provided parameter values, which also leads to the variable binding of their primitive actions.

In Listing 4, we illustrate the *indirection* compound. Indirection is applicable in case 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 [44]. 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 bound 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=REQUIRED to ${cv}.${A}${n}
  add port "${A}${n}_I2" kind=PROVIDED to ${cv}.${A}${n}
  add port "${A}_I" kind=PROVIDED to ${cv}.${A}
  add port "${B}_I" kind=REQUIRED to ${cv}.${B}
  add connector "${A}${n}_I1_${A}_I" from ${cv}.${A}${n}.${A}${n}_I1 to ${cv}.${A}.${A}_I
  add connector "${B}_I_${A}${n}_I2" from ${cv}.${B}.${B}_I to ${cv}.${A}${n}.${A}${n}_I2
  add stereotype <<"${n}">> to ${cv}.${A}${n}
}
```

Listing 4: Indirection Compound Action Specification

A usage example of the indirection compound is presented in Listing 5.

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

Listing 5: Example Usage of the Compound “indirection”

This compound action will create a proxy for invoking the component “Service” from the component “Facade”. The command `add compound` contains a reference to a compound definition. When a compound action is executed, the reference to the compound definition is resolved and the actual variables are replaced. In our example, the variables `n`, `cv`, `A`, and `B` will get the values “Proxy”, the C&C view model name “model” 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 model.ServiceProxy
add port "ServiceProxy_I2" kind=REQUIRED to model.ServiceProxy
  ...
```

Listing 6: Binding of Primitive Actions of Indirection Compound Action

A compound can extend other compounds, and therefore, inherit the corresponding sets of actions contained in these compounds. For instance, in Listing 7, we specify a new compound `integrationAdapter` which extends the existing compound `adapter`.

```
compound integrationAdapter extends adapter {...}
```

Listing 7: Example of Compound Extension

The advantage of the compounds is that they specify a group of transformation actions that can be reused with a simple parametrized transformation action "`add compound ...`". In this way, we can increase the reusability of the AK transformations. Therefore, apart from the reuse of the low-level model actions that apply on the C&C views, the reusability of the AK transformations can be further increased by the introduction of compound actions which group other transformation actions.

For the needs of a case study that we discuss in detail in the next section, we have modeled the following six compounds and have reused them in various design situations covering 21 design patterns in total:

Indirection: Indirection happens when a "proxy" component receives a message on behalf of a "target" component and forwards the message to that "target". Afterwards the result is sent back through the "proxy" component again.

Shield: A component can act as a "shield" for a set of components that form a subsystem. In this case, a client can access the members of the subsystem only through this "shield".

Grouping: A group member is part of an abstract or virtual entity. That is, there is no component in the software architecture for representing the group as a whole, but it is made only of its parts.

Callback: A callback denotes an invocation to a component B that is stored as an invocation reference in a component A. Between two components A and B, a set of callbacks can be defined, also usually implemented as methods.

Transformer: A transformer performs transformation as well as enrichment, splitting, aggregation, etc. of data that is sent from component A to component B.

Router: A router routes requests of a component to a set of other components according to specified criteria.

In Table 1 we present the compound specifications along with the patterns that share these common abstractions. For these compound specifications the primitive abstractions modeled in [44] were used as reference with slight modifications. While Zdun and Avgeriou use OCL to document architectural primitives [44], we express these abstractions using compounds. For a detailed description of the patterns for service-based platform integration that are included in Table 1 you can refer to [23, 24].

Compound Name	Compound Specification	Design Patterns

Indirection	`compound indirection (cv A B) {`
	`  add component "${A}${n}"`
	`  add port "${A}${n}_I1" kind=REQUIRED to ${cv}.${A}${n}`
	`  add port "${A}${n}_I2" kind=PROVIDED to ${cv}.${A}${n}`
	`  add port "${A}_I" kind=PROVIDED to ${cv}.${A}`
	`  add port "${B}_I" kind=REQUIRED to ${cv}.${B}`
	`  add connector "${A}${n}_I1_${A}_I" from ${cv}.${A}${n}.`
	`       ${A}${n}_I1 to ${cv}.${A}.${A}_I`
	`  add connector "${B}_I_${A}${n}_I2" from ${cv}.${B}.${B}`
	`       _I to ${cv}.${A}${n}.${A}${n}_I2`
	`  add stereotype <<"${n}">> to ${cv}.${A}${n}`
	`}`
		PROXY, REMOTE PROXY, ADAPTER, REMOTE ADAPTER

Shield	`compound shield (cv T L) {`
	`  add component "${T}"`
	`  add stereotype <<"${n}">> to ${cv}.${T}`
	`  add port "${T}_S" kind=REQUIRED to ${cv}.${T}`
	`  for (c : ${L})`
	`    add port "${c}_S" kind=PROVIDED to ${cv}.${c}`
	`    add connector "${T}_S_${c}_S" from ${cv}.${T}.${T}_S`
	`         to ${cv}.${c}.${c}_S`
	`  end`
	`}`
		FACADE, REMOTE FACADE, GATEWAY, SERVICE INTERFACE

Grouping	`compound grouping (cv T L) {`
	`  add component "${T}"`
	`  add stereotype <<"${n}">> to ${cv}.${T}`
	`  add port "${T}_G" kind=PROVIDED to ${cv}.${T}`
	`  for (c : ${L})`
	`    group ${cv}.${c} container ${cv}.${T}`
	`  end`
	`}`
		SERVICE ABSTRACTION LAYER

Continued on next page

Table 1 – continued from previous page

Compound Name	Compound Specification	Design Patterns
Callback	`compound callback (cv A B) {`	RESULT CALLBACK, REQUEST-REPLY, REQUEST-ACKNOWLEDGE CALLBACK
	`  add port "${A}_E" kind=PROVIDED to ${cv}.${A}`	
	`  add stereotype <<"EventPort">> to ${cv}.${A}.${A}_E`	
	`  add port "${A}_C" kind=REQUIRED to ${cv}.${A}`	
	`  add stereotype <<"CallbackPort">> to ${cv}.${A}.${A}_C`	
	`  add port "${B}_E" kind=REQUIRED to ${cv}.${B}`	
	`  add stereotype <<"EventPort">> to ${cv}.${B}.${B}_E`	
	`  add port "${B}_C" kind=PROVIDED to ${cv}.${B}`	
	`  add stereotype <<"CallbackPort">> to ${cv}.${B}.${B}_C`	
	`  add connector "${B}_E_${A}_E" from ${cv}.${B}.${B}_E to`	
	`        ${cv}.${A}.${A}_E`	
	`  add connector "${A}_C_${B}_C" from ${cv}.${A}.${A}_C to`	
	`        ${cv}.${B}.${B}_C`	
	`  add stereotype <<"${n}">> to ${cv}.${A}_E_${B}_E`	
	`  add stereotype <<"${n}">> to ${cv}.${B}_C_${A}_C`	
	`}`	
Transformer	`compound transformer (cv A B) {`	DATA MAPPER, MESSAGE TRANSLATOR, SPLITTER, AGGREGATOR, CONTENT ENRICHER, CONTENT FILTER
	`  add component "${A}${n}"`	
	`  add port "${A}${n}_T1" kind=REQUIRED to ${cv}.${A}${n}`	
	`  add port "${A}${n}_T2" kind=PROVIDED to ${cv}.${A}${n}`	
	`  add port "${A}_T" kind=PROVIDED to ${cv}.${A}`	
	`  add port "${B}_T" kind=REQUIRED to ${cv}.${B}`	
	`  add connector "${A}${n}_T1_${A}_T" from ${cv}.${A}.${A}`	
	`       ${n}_T1 to ${cv}.${A}.${A}_T`	
	`  add connector "${B}_T_${A}${n}_T2" from ${cv}.${B}.${B}`	
	`       _T to ${cv}.${A}.${A}${n}_T2`	
	`  add stereotype <<"${n}">> to ${cv}.${A}${n}`	
	`}`	
Router	`compound router (cv T A L) {`	PUBLISH-SUBSCRIBER, MESSAGE ROUTER, CONTENT-BASED ROUTER
	`  add component "${T}"`	
	`  add stereotype <<"${n}">> to ${cv}.${T}`	
	`  add port "${T}_R1" kind=PROVIDED to ${cv}.${T}`	
	`  add port "${A}_R" kind=REQUIRED to ${cv}.${A}`	
	`  add connector "${A}_R_${T}_R1" from ${cv}.${A}.${A}_R`	
	`       to ${cv}.${T}.${T}_R1`	
	`  add port "${T}_R2" kind=REQUIRED to ${cv}.${T}`	
	`  for (c : ${L})`	
	`    add port "${c}_R" kind=PROVIDED to ${cv}.${c}`	
	`    add connector "${T}_R2_${c}_R" from ${cv}.${T}.${T}`	
	`         _R2 to ${cv}.${c}.${c}_R`	
	`  end`	
	`}`	

cv: component view n: design pattern T: component name A, B: components L: list of components

Table 1: Reusable AK Transformation Compounds

3.5. Generation of Constraints

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 17. 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 transformation actions, without requiring any additional effort from the developers and architects. Please note that these constraint templates have been developed only for the specific transformation actions to demonstrate how constraint-based consistency checking can be achieved. Further constraint templates for particular circumstances can be formulated assuming a basic knowledge of an OCL-like language. Depending on the reusable ADDs to be related with AK reusable transformations, constraints can be reused or/and adapted from other approaches that use similar constraints in order to express decisions formally at architectural design level [18, 20, 44].

As mentioned before, constraint templates only need to be defined once at the model-level (i.e., Step 1–2 in Figure 4(a)) and can then be reused for concrete instantiations of the ADD model (i.e., Step 3–6 in Figure 4(b)). For instance, let us consider the following transformation action from the previous example of Listing 6:

```
add component "ServiceProxy"
```

Listing 14: Example of a Transformation Action

The resulting C&C model can be checked for its consistency against the related decision (e.g., `ADD1`) by the constraint of Listing 15, which checks that the underlying component is present in the C&C model. Please note that this constraint derives from the corresponding template once the transformation action is executed.

```
context component::ComponentView ERROR "ADD ADD1: Component ServiceProxy does not exist":
  element.typeSelect(component::Component).exists(c|c.name == "ServiceProxy");
```

Listing 15: Example of a Generated Constraint

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 using the constraint validator. 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.

We show in Table 2 an excerpt of the set of reusable constraints in template-based form that correspond to the actions defined in the AK transformation language. The table does not include transformation actions that delete elements of the C&C view (in our proposal, constraints should check the existence and not non-existence of elements) as well as non-primitive transformation actions such as `add compound` (these can be analyzed in primitive actions). The current form of the reusable constraint templates is designed for better understanding and maintaining. The set of constraint templates is not yet complete but rather for illustrating how consistency constraints can be defined in a reusable manner that can be then instantiated after being bound to concrete parameters.

3.6. Tool Implementation

For the sake of demonstration, validation, and evaluation of our approach we have developed a prototype in the form of Eclipse plugins[9]. The AK transformation language (DSL) and its editor were developed with Xtext framework[10]. The Eclipse Modeling Framework (EMF)[11] project is used to create all required models (e.g., architectural decision models) and the code for editing these models. The template binding of transformation actions and constraints in template form is done by the Velocity Template Engine[12]. Finally, the constraints for consistency checking between decisions and designs are formulated in the OCL-like Check language[13].

[9]You can install the prototype in the Eclipse environment from the ADvISE Update Site `http://indenica.swa.univie.ac.at/public/advise` and download the source code from `http://indenica.swa.univie.ac.at/public/advise/ADvISE-src.zip`

[10]`http://www.eclipse.org/Xtext`

[11]`http://www.eclipse.org/modeling/emf`

[12]`http://velocity.apache.org`

[13]`http://www.eclipse.org/modeling/m2t/?project=xpand`

Transformation Action	Reusable constraint in template form
– `add component` – `update component`	`context component::ComponentView ERROR "${ADD}:Component ${component} does not exist": element.typeSelect(component::Component).exists(c\|c.name=="${component}");`
– `add connector` – `update connector`	`context component::ComponentView ERROR "${ADD}:Component ${componentA} and ${componentB} are not connected": element.typeSelect(component::Component).exists(c1\|c1.name=="${componentA}" && element.typeSelect(component::Component).exists(c2|c2.name=="${componentB}" && element.typeSelect(component::Connector).exists(conn|(c1.port.exists(p|conn.source==p) && c2.port.exists(p|conn.target==p)) || (c1.port.exists(p|conn.target==p) && c2.port.exists(p|conn.source==p)))));`
– `add port` – `update port`	`context component::ComponentView ERROR "${ADD}:Port ${port} of kind ${portKind} for component ${component} does not exist": element.typeSelect(component::Component).exists(c\|c.name=="${component}" && c.port.exists(p|p.name=="${port}" && p.kind==component::PortKind::${portKind}));`
– `add property` – `update property`	`context component::ComponentView ERROR "${ADD}:Property ${type}=${value} of ${element} does not exist": annotation.typeSelect(component::Property).exists(p\|element.exists(c\|c.annotation.exists(a\|a==p) && c.name=="${element}" && p.type=="${type}" && p.value=="${value}"));`
– `add stereotype` – `update stereotype`	`context component::ComponentView ERROR "${ADD}: ${element} is not annotated as ${stereotype}": annotation.typeSelect(component::Stereotype).exists(s\|element.exists(c\|c.annotation.exists(a\|a==s) && c.name=="${element}" && s.text=="${stereotype}"));`
– `group component`	`context component::ComponentView ERROR "${ADD}:Component ${container} does not contain ${component}": element.typeSelect(component::Component).exists(c\|c.name=="${container}" && c.nestedComponent.exists(c|c.name=="${component}"));`

Table 2: List of Reusable Constraints in Template Form

4. Case Study

In this Section, we introduce a case study on service-based platform integration in the area of warehouse automation and discuss in the context of this case study the application of our proposal, i.e., the integration of design decisions with C&C views using the AK transformation language. Three heterogeneous service 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. An intermediate platform 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. For instance, as shown in Figure 7, two Proxies (i.e., Proxy1 and Proxy2) and one Adapter will need to be developed at the platform integration layer for communicating with the three heterogeneous platforms.

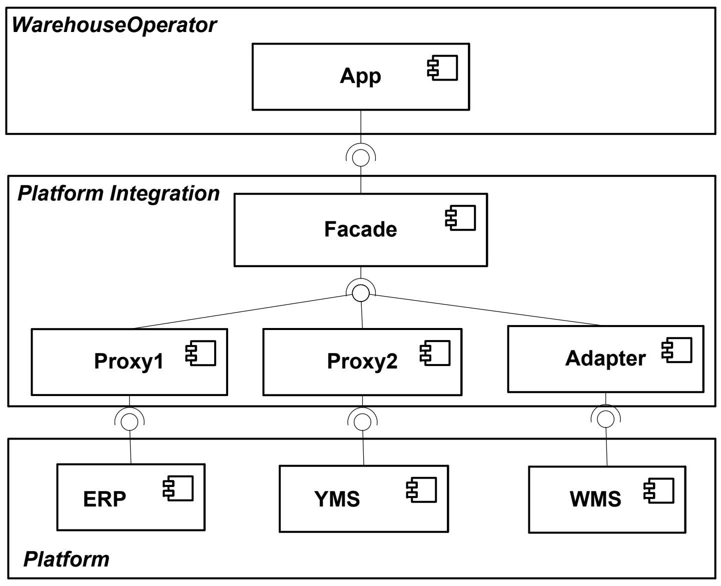

Figure 7: Example from Service-based Platform Integration Case Study

In our previous work, we have introduced an ADD model to handle various integration aspects in the service-based platform integration domain, i.e., integration and adaptation, interface design, communication style, and communication flow [23, 24]. We present in Table 3 an excerpt of the ADD model of the platform integration scenario consisting of questions and different alternative options (or answers) along with the corresponding transformation actions that translate the underlying decisions into elements of the C&C view. This ADD model is modeled using ADvISE. Please note that the dependencies and constraints between the questions, decisions, and options are not present in Table 3 for simplicity reasons.

This example shows the support in the decision making on the type of integrating components 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` (note: `cv` refers to the target C&C view). Every ADD option (or answer) in the ADD model is associated with a set of primitive and compound actions based on pattern primitives in template form as defined in Section 3. The excerpt of the ADD model consists of six questions, uses eight primitive actions and two compound actions (`integrationAdapter` once and `indirection` twice) and is related to three patterns: Proxy (local or remote), Adapter (local or remote) and Integration Adapter.

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.

When actual ADDs are made, the parameters of the transformation actions of Table 3 are bound to concrete values. For instance, let us assume that the software architect opts for a remote proxy as an integrating component between a YMS service, namely, *TruckMgmnt*, and an integration layer's component, namely, *OperatorFacade*. The actual ADDs will be reflected in the corresponding C&C view by executing the derived transformation actions of Listing 16.

```
add compound indirection "Proxy" (example TruckMgmnt OperatorFacade)
add stereotype <<"Remote Proxy">> to example.TruckMgmntProxy
```

Listing 16: Transformation Actions Example from Case Study

In Table 4, we report a set of reusable ADDs that we leveraged in the warehouse case study. We also document on the number of constraints that were generated in order to check the consistency between the ADDs and the resulting C&C view. In total, we documented 24 ADDs and, using our tools, we transformed them into C&C view elements and generated 222 corresponding constraints. We note that the actual benefit of our proposed reusable constraint templates does not merely rely on the generation of concrete constraints but on the ability to be leveraged in the application context of recurring design decisions. For instance,

ADD Options	AK Transformation Actions

Type of Integrating Component
– None
– Same Interface
– Different Interface
	`#if(${TypeOfComponent} == "None")`
	`  add port "${PS}_p1" kind=PROVIDED to ${cv}.${PS}`
	`  add port "${IC}_p1" kind=REQUIRED to ${cv}.${IC}`
	`  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" (${cv} ${PS} ${IC})`
	`#elseif(${TypeOfComponent} == "Different Interface")`
	`  add compound indirection "Adapter" (${cv} ${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(${HeterogeneousSystems} == "Yes")`
	`  add compound integrationAdapter "Integration Adapter" (${cv} ${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 3: An Excerpt of the Service-based Platform Integration ADD Model and its Corresponding AK Transformation Actions

without support from our techniques the software architects must manually make 24 decisions and define (or copy/paste) 222 constraints for integrating each platform service, which is a tedious and error-prone task.

5. Evaluation Results

5.1. Reusability

Notwithstanding 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 the long term. In our approach, reusability can be

ADD	Times Reused	Constraints
Proxy	3	3 x 9
Adapter	1	1 x 11
Result Callback	7	7 x 12
Facade	1	1 x 8
Gateway	2	2 x 6
Publish-Subscriber	7	7 x 8
Data Mapper	2	2 x 8
Content Enricher	1	1 x 8
Total	24	222

Table 4: Reusable ADDs and Generated Constraints in the Warehouse Case Study

achieved at three different levels: 1) First of all, the AK transformation language hides the low-level model actions that are needed to transform the C&C view models. These model actions are embedded in the enactment engine of the DSL; 2) In addition, 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; 3) Finally, the use of compound actions that can be extended and inherited increases reusability, as it allows the introduction of new transformation actions with parameters which group other transformation actions. In the same way, loops also contribute to the reusability of transformation actions.

In order to show the reusability of our approach, 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. Regarding the definition of the action templates of Table 3, the numbers of actions that have to be edited manually for the reusable decisions Direct Calls, Proxy, Adapter, and Integration Adapter, are four, two, four, and three actions, respectively. By defining compound actions, we can reduce the number of required actions in the last three cases where single actions were contained in the compound actions *add compound indirection* and *add compound integrationAdapter (extends indirection).*

Table 5 shows the actions that need to be edited manually per decision—both compound and primitive actions—along with the primitive actions, as well as the model actions in which the primitive actions are translated from the enactment engine of the AK transformation language. The model actions are the actions that eventually apply on the C&C views. The number of the actions that are directly applied on the C&C model are 13, 25, 26 and 42 respectively for the four ADDs, which means that without the use of the AK transformation language the effort for editing and executing these actions would increase significantly. 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.

Reusable ADD	Actions (with compounds)	Primitive Actions	Model Actions
Direct Calls	4	4	13
Proxy	2	9	25
Adapter	4	11	26
Integration Adapter	3	15	42

Table 5: Comparison of Number of Actions for Reusable ADDs

5.2. Modeling Effort

We performed a quantitative evaluation by measuring the modeling effort for editing transformation actions. In particular, we measured the increase of modeling effort if the compound actions in the AK transformation language are replaced by primitive or model actions. The results are presented in Table 6 for the ADDs we have documented in the warehouse case study. We notice that the modeling effort would increase up to 1100% and up to 3700% if the compound actions would be replaced by primitive or model actions respectively (i.e., for Result Callbacks). On average, we have an increase of modeling effort of 549% and 2041% for the aforementioned cases and for the setting we created for the needs of the case study.

ADD (Times reused)	Actions (with compounds)	Primitive Actions	Model Actions	Average increase of modeling effort *for Primitive Actions*	*for Model Actions*
Proxy (3)	6	27	75	350%	1150%
Adapter (1)	4	11	26	175%	550%
Result Callback (7)	7	84	266	1100%	3700%
Facade (1)	3	8	24	167%	700%
Gateway (2)	2	12	30	500%	1400%
Publish-Subscriber (7)	7	56	189	700%	2600%
Data Mapper (2)	2	16	50	700%	2400%
Content Enricher (1)	1	8	25	700%	2400%
Total	32	222	685	549% (on average)	2041% (on average)

Table 6: Modeling Effort for Reusable ADDs

Figure 8 compares the increase of the number of compound, primitive, and model actions on average for the ADDs that were reused in the warehouse case study discussed in this section. We notice that the number of actions edited with the AK transformation language remains low in comparison to the primitive and model actions that need to be executed as the number of reusable ADDs increases.

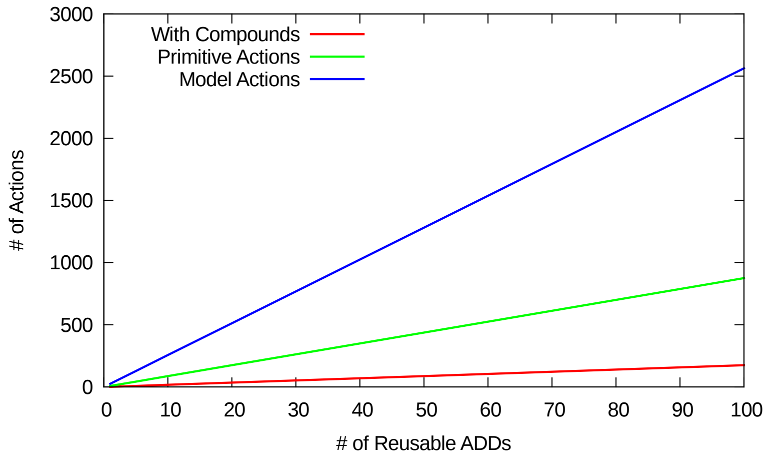

Figure 8: Comparison between Numbers of Actions on average for the Warehouse Case Study

5.3. Discussion and Limitations

Although we have implemented and demonstrated our proposal using two specific tools, we claim that our approach can be generalizable to a certain 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. This is similar to the way we pass parameters when invoking a function in traditional programming languages.

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 to elements of other typical C&C views. Therefore, our approach is also applicable for most of existing component models such as the 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.

In case the underlying component models are different from the model presented in this paper, we can reuse most of the aforementioned AK transformation actions and only need to add additional actions in order to accommodate the specific elements that do not exist in our model. As the commonality between our component model and the widely-used models is substantial, the effort to adapt the AK transformation actions to specific needs is mostly marginal. Moreover, the notion of compound inheritance and extension would help to alleviate such adaptation effort.

Regarding the manual steps of our approach, namely the editing of reusable ADD models and constraint templates with the mappings between them, many of the participating assets can be customized and reused in various design situations. The reusability and automation of our approach is based on reusable and recurring ADDs. Nevertheless, some non-reusable ADDs can also be integrated to the C&C views with small extra efforts.

By implementing our proposal in a real-life design setting—in the context of the warehouse case study—we showed that it offers high reusability and reduces the efforts for documenting and maintaining two important design artifacts, the design decisions and C&C views. However, in the scope of our study, we have not conducted any usability studies within human designers, thus we were not able to assess the effectiveness and usefulness of our approach in the design process and in the long term. Lacking such an empirical evaluation, we are not able, for instance, to predict the time needed by software architects to learn and use the AK transformation language and from what number of reusable decisions the initial modeling effort pays off. Such empirical evidence would also be useful for introducing this or similar methods in the industrial practice. An empirical evaluation of our proposal, nonetheless, is part of our future work.

6. Related Work

Our approach presented in this paper is motivated by the current gaps between architectural design decisions and architectural models that have partially or not been addressed in the literature yet. In this section, we will discuss related techniques and methods, especially existing approaches for architectural decision support and for mapping among software modeling and development artifacts.

6.1. Existing Approaches for Architectural Decision Support

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 [16]. There are numerous attempts on documentation and leveraging of design rationales. Clements et al. suggest a general outline for documenting architectures and guidelines for justifying design decisions [8]. Tyree and Akerman present a rich template for capturing and documenting several aspects of architectural design decisions [42]. Another technique proposed by Lee and Kruchten aims at establishing formalized ontological description of architectural decisions and their relationships [21] whilst Harrison et al. use patterns—which are proven knowledge—for capturing recurring decisions [12]. Zimmermann et al. proposed decision meta-models and guidelines for formulating and documenting design decisions, and illustrated their methods in the context of the SOA design space [45].

Several tools and techniques have been developed to assist software architects in capturing, managing, and sharing of ADDs [36]. In [16], Jansen and Bosch advocate for considering software architecture as a set of design decisions and propose a new approach, namely Archium, for describing ADDs. The SEURAT toolkit developed by Burge and Brown provides means for browsing and analyzing architectural design rationales [4]. Similar tools proposed by Babar and Gorton can also be used for capturing and managing software architecture knowledge [1]. Capilla et al. also emphasize the importance of recording and maintaining architectural design decisions and present a Web-based architecture design decision support system, namely ADDSS, for this purpose [5]. These approaches mainly target reasoning on software architectures, capturing and reusing of AK and have not considered the maintenance and consistency of ADDs with architectural views.

Architectural decisions are the result of making trade-offs for the quality attribute requirements. For example, in the Architecture Tradeoff Analysis Method (ATAM) and Attribute-Driven-Design Method (ADD) [3] the analysis of architectural trade-offs is an important part of the architectural decision making process. Bachmann et al. suggest a reasoning framework with quality attribute knowledge to help architects make trade-offs that impact individual quality attributes in an architecture [2]. These and other approaches for supporting architectural decision making have not been integrated with tools for modeling and documentation of architectural decisions.

Existing methods and techniques, as we discussed above, were grounded on the main idea of considering ADDs first-class citizens in software architecture design. Therefore, these approaches lay a solid foundation on capturing and managing design decisions for several successive studies, including our approach presented in this paper. For instance, the ADD model supported by ADvISE inherits common concepts and elements of existing meta-models and templates for recording design decisions. Nevertheless, our approach goes beyond with not only supporting reusable ADDs but also providing an extensible, reusable AK transformation language for transforming architectural design intentions and knowledge onto architectural models (such as C&C models). The AK transformation language also supports automatically generating constraints for checking the consistency between ADDs and the corresponding architectural models.

6.2. Relating ADDs and Software Architectures

Our approach is not the first one in relating ADDs to software architectures. The problem of not documenting and not maintaining ADDs that cause design knowledge vaporization has been discussed before [7, 16]. For instance, Choi et al. suggest 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 [7]. Compared to our proposal, this approach demands that most of the work is done manually: decision making, architectural design and change propagation during software evolution.

STREAM-ADD [9] also relates architectural decisions documented in decision templates with requirements and architectural models generated from these requirements. This approach focuses 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. Küster has proposed to intertwine ADDs with standard architecture documentation processes by defining architecture-specific decision types along with OCL constraints used for decision conformance checking of component and deployment views [20]. The aforementioned approach uses similar concepts for checking the conformance of ADDs to architectural views, however, comparing to our proposal, it does not support the transformation of ADDs into components and connectors.

Another popular technique for relating ADDs and software architecture aims at establishing trace links between design decisions and architectural elements. Capilla et al. [6] introduce fine-grained traceability links between design decisions and other software artifacts. Könemann and Zimmermann [18] 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 [32] 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. Recently, Malavolta et al. present an approach that enables analysis of change impact of certain change in architectural decisions [28]. This is mainly achieved by manually defining trace links between design decisions and other artifacts. Although this is not the focus of our approach, the AK transformation language could be easily extended to automatically establish trace links between design decisions and consistency checking constraints as well. Also, none of the aforementioned approaches target the reusability of these links between ADDs and architectural views, nor do they tackle the complexity of large numbers of reusable ADDs. Furthermore, our approach can also provide support for automatically transforming architectural design intentions and knowledge to concrete architectural elements and configurations through the AK transformation language.

6.3. Mapping to and Generating Software Architectures

The generation of architectural design views from specifications or other architectural views has been studied extensively in the literature. Pérez-Martínez and Sierra-Alonso use semi-automated model-to-model transformations to generate component-and-connector (C&C) architecture models from classes and packages analysis models by using 32 OCL-based mapping rules [34]. In this approach, architects can use the analysis model, which is a UML class diagram extended with additional profiles for the Unified Development Process [15], to describe architectural design intentions. After that, the elements of the analysis model will be coerced to the concepts of the C&C model according to the mapping rules.

In a different approach [22], 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. Different from our approach, this approach supports rather the composition than the generation of software architectures as it requires that all architectural elements are prescribed. Moreover, 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. show that using model-driven development techniques can help in mapping requirements to architectural design [17]. Grünbacher et al. introduce the mapping from requirements to intermediate models that are closer to software architecture [11]. A different approach presented by van Lamsweerde et al. 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 such as availability and fault-tolerance [43].

Sochos et al. present an approach, namely Feature-Architecture Mapping (FArM), that focuses on providing a strong mapping between features and software product line architectures [37]. The drawback of this approach is to assume a one-to-one mapping, that is, each component of the resulting product line architecture encapsulates the business logic of a feature. In reality, a high-level abstraction concept (such as a requirement or feature) might relate to more than one architectural elements and vice versa.

In the aforementioned approaches, in which requirements are related to software designs, although the transformations are done automatically, the mapping mostly has to be performed manually and is not reusable. We note that, in these approaches, the rationales that led from the requirements to the architectural views are neither considered nor documented. That is, because the requirements belong to the problem space while ADDs to the solution space. 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.

7. 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 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 and reusability. 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 research agenda, we aim to generalize the harmonization of other architectural design views with architectural decisions using similar reusable AK transformations. We also plan to study repair actions for resolving inconsistencies between reusable ADDs and component views as well as investigate the possibility of bidirectional transformations, i.e., also from component views onto decisions. Part of our future work is, finally, to consider other aspects of architectural designs apart from the elements of C&C models.

Appendix A. AK Transformation Language Specification

Listing 17 presents a formal definition of the AK transformation language in terms of the EBNF-like notation provided by Xtext. We note that square brackets denote the cross-references between different models. For instance, the rule "`AddConnector`" defined in Line 37–38 refers to the components' ports at the two ends of the newly added connector. Using AK transformation language simple actions that add, delete, or update components, connectors, and other elements or properties of C&C views can be edited (e.g., `AddComponent`, `UpdateConnector`, etc.). Apart from that, AK transformation language supports the following actions: `Condition`, `ForLoop`, `WhileLoop`, `Compound`, `Group`, and `Refine`. A `ForLoop` (see Line 21–22) can be used in order to define one or more actions that are performed over a predefined set of design elements. The `WhileLoop` will do the same as `ForLoop` but under certain boolean constraints. A `Group` (see Line 91–92) can be used to define the grouping of a finite set of components as sub-components of another component, while `Refine` (see Line 94–95) indicates a mapping of an abstract and high-level component onto one or more low-level components. A Compound (see Line 25–26) is used to represent a composite construct containing multiple actions. It can inherit the definitions of existing Compounds via the keyword "extends", and therefore, reduce redundancy and duplicated efforts.

```
1  grammar at.ac.univie.cs.swa.dsl.action.ActionDSL with org.eclipse.xtext.common.Terminals

3  ActionDSL:
4    {ActionDSL}
5    "module" name=FQN
6    (compounds+=Compound)*
7    (actions+=Action)+;

9  Action:
10   Add | Delete | Update | Group | Refine | Condition | WhileLoop | ForLoop;

12 Condition:
13   "if" "(" condition=BooleanExpression ")"
14   (thenActions+=Action)+
15   (=>"else" (elseActions+=Action)+)?;

17 WhileLoop:
18   {WhileLoop}
19   "while" expression=BooleanExpression "do" (actions+=Action)+ "end";

21 ForLoop:
22   {ForLoop}
23   "for" "("element=ID ":" params=LIST")" (actions+=Action)+ "end";

25 Compound:
26   "compound" name=ID ("extends" (parent+=[Compound|FQN])+)? spec=Spec;

28 Spec:
29   "("(args+=ID)+")" "{" (actions+=Action)* "}";

31 Add:
32   AddComponent | AddConnector | AddPort | AddProperty | AddStereotype | AddCompound;

34 AddComponent:
35   "add component" name=STRING;

37 AddConnector:
38   "add connector" name=STRING "from" source=[component::Port|FQN] "to" target=[component::
         Port|FQN];

40 AddPort:
41   "add port" name=STRING "kind=" kind=PortKind "to" component=[component::Component|FQN];

43 enum PortKind:
44   provided="PROVIDED" | required="REQUIRED";

46 AddStereotype:
47   "add stereotype" "<<" text=STRING ">>" "to" target=[core::Element|FQN];

49 AddProperty:
  "add property" name=STRING "type=" type=STRING "value=" value=STRING "to" target=[core::
      Element|FQN];

AddCompound:
  "add compound" compound=[Compound|FQN] name=STRING "("(args+=ID)+")";

Delete:
  DeleteComponent | DeleteConnector | DeletePort | DeleteProperty | DeleteStereotype;

DeleteComponent:
  "delete component" component=[component::Component|FQN];

DeleteConnector:
  "delete connector" conn=[component::Connector|FQN];

DeletePort:
  "delete port" port=[component::Port|FQN];

DeleteProperty:
  "delete property" property=[component::Property|FQN];

DeleteStereotype:
  "delete stereotype" stereotype=[component::Stereotype|FQN];

Update:
  UpdateComponent | UpdateConnector | UpdatePort | UpdateProperty | UpdateStereotype;

UpdateComponent:
  "update component" component=[component::Component|FQN] "name=" newName=STRING;

UpdateConnector:
  "update connector" conn=[component::Connector|FQN] "name=" newName=STRING;

UpdatePort:
  "update port" port=[component::Port|FQN] ("name=" newName=STRING)? ("kind=" newKind=
      PortKind)?;

UpdateProperty:
  "update property" prop=[component::Property|FQN] ("name=" newName=STRING)? ("type="
      newType=STRING "value=" newValue=STRING)?;

UpdateStereotype:
  "update stereotype" stereotype=[component::Stereotype|FQN] "text=" newText=STRING;

Group:
  "group" component=[component::Component|FQN] "container" container=[component::Component|
      FQN];

Refine:
  "refine" component=[component::Component|FQN] "in" (refinedComp+=STRING)+;
```
Listing 17: AK transformation language specification

References

[1] M. A. Babar and I. Gorton. A Tool for Managing Software Architecture Knowledge. In *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, SHARK-ADI'07, pages 11–, Washington, DC, USA, 2007. IEEE Computer Society.

[2] F. Bachmann, L. Bass, M. Klein, and C. Shelton. Designing Software Architectures to Achieve Quality Attribute Requirements. *Software, IEEE Proceedings*, 152(4):153–165, Aug. 2005.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, volume 2. Addison-Wesley Professional, 2003.

[4] J. E. Burge and D. C. Brown. An Integrated Approach for Software Design Checking Using Design Rationale. In *1st Int'l Conf. on Design Computing and Cognition (DCC'04)*, pages 557–576. Kluwer Academic Press, 2004.

[5] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Software Engineering Notes*, 31(5), Sept. 2006.

[6] R. Capilla, O. Zimmermann, U. Zdun, P. Avgeriou, and J. M. Küster. 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*, pages 303–318. Springer, 2011.

[7] Y. Choi, H. Choi, and M. Oh. An architectural design decision-centric approach to architectural evolution. In *11th Int'l Conf. on Advanced Communication Technology (ICACT), Gangwon-Do, South Korea*, pages 417–422. IEEE Press, 2009.

[8] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.

[9] D. Dermeval, J. Pimentel, C. T. L. L. Silva, J. Castro, E. Santos, G. Guedes, M. Lucena, and A. Finkelstein. 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*, pages 602–611. IEEE Computer Society, 2012.

[10] O. M. Group. Uml 2.4.1 superstructure specification. `http://www.omg.org/spec/UML/2.4.1`. Last accessed: 2014-06-10.

[11] P. Grünbacher, A. Egyed, and N. Medvidovic. Reconciling Software Requirements and Architectures with Intermediate Models. *Softw. Syst. Model.*, 3(3):235–253, 2003.

[12] N. B. Harrison, P. Avgeriou, and U. Zdun. Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4):38–45, 2007.

[13] T. Holmes, H. Tran, U. Zdun, and S. Dustdar. Modeling Human Aspects of Business Processes - A View-Based, Model-Driven Approach. In *European Conf. Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 246–261, Berlin, Germany, 2008. Springer-Verlag.

[14] ISO/IEC/IEEE. Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, pages 1–46, 1 2011.

[15] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1999.

[16] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *5th Working IEEE/IFIP Conf. on Software Architecture (WICSA), Pittsburgh, PA, USA*, pages 109–120. IEEE Computer Society, 2005.

[17] H. Kaindl and J. Falb. Can We Transform Requirements into Architecture? In *3rd Int'l Conf. on Software Engineering Advances (ICSEA), Sliema, Malta*, pages 91–96. IEEE Computer Society, 2008.

[18] P. Könemann and O. Zimmermann. Linking Design Decisions to Design Models in Model-Based Software Development. In *4th European Conf. in Software Architecture (ECSA), Copenhagen, Denmark*, pages 246–262. Springer, 2010.

[19] P. Kruchten, R. Capilla, and J. C. Dueñas. The Decision View's Role in Software Architecture Practice. *IEEE Software*, 26(2):36–42, Mar. 2009.

[20] M. Küster. Architecture-Centric Modeling of Design Decisions for Validation and Traceability. In *Proceedings of the 7th European Conf. on Software Architecture (ECSA)*, pages 184–191, Berlin, Heidelberg, 2013. Springer.

[21] L. Lee and P. Kruchten. Capturing Software Architectural Design Decisions. In *2007 Canadian Conf. on Electrical and Computer Engineering*, pages 686–689. IEEE Computer Society, 2007.

[22] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language Support for Managing Variability in Architectural Models. In *Software Composition*, pages 36–51, 2008.

[23] I. Lytra, S. Sobernig, H. Tran, and U. Zdun. A Pattern Language for Service-Based Platform Integration and Adaptation. In *17th Annual European Conf. on Pattern Languages of Programs*, pages 111–120. Hillside, Jul. 2012.

[24] I. Lytra, S. Sobernig, and U. Zdun. 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 Computer Society, 2012.

[25] I. Lytra, H. Tran, and U. Zdun. 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*, pages 287–296. Springer, 2012.

[26] I. Lytra, H. Tran, and U. Zdun. Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations. In *European Conf. on Software Architecture (ECSA)*, LNCS 7957, pages 224–239. Springer, July 2013.

[27] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human-Computer Interaction*, 6:201–250, 1991.

[28] I. Malavolta, H. Muccini, and V. S. Rekha. Supporting Architectural Design Decisions Evolution through Model Driven Engineering. In *Proceedings of the 3rd Int'l Conf. on Software Engineering for Resilient Systems*, SERENE'11, pages 63–77, Berlin, Heidelberg, 2011. Springer-Verlag.

[29] C. Mayr, U. Zdun, and S. Dustdar. Model-Driven Integration and Management of Data Access Objects in Process-Driven SOAs. In *First European Conf., ServiceWave 2008, Proceedings*, LNCS 5377, pages 62–73, Madrid, Spain, 2008. Springer-Verlag.

[30] N. R. Mehta and N. Medvidovic. 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*, pages 347–350. ACM, 2003.

[31] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, Mar. 2006.

[32] M. Mirakhorli and J. Cleland-Huang. 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*, pages 123–132. IEEE Computer Society, 2011.

[33] E. Mulo, U. Zdun, and S. Dustdar. An Event View Model and DSL for Engineering an Event-based SOA Monitoring Infrastructure. In *Proceedings of the Fourth ACM Int'l Conf. on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010*, pages 62–72, 2010.

[34] J. E. Pérez-Martínez and A. Sierra-Alonso. From Analysis Model to Software Architecture: A PIM2PIM Mapping. In *Model Driven Architecture - Foundations and Applications (ECMDA-FA), Biblao, Spain*, pages 25–39, 2006.

[35] D. C. Schmidt and F. Buschmann. Patterns, Frameworks, and Middleware: Their Synergistic Relationships. In *25th Int'l Conf. on Software Engineering (ICSE)*, pages 694–704, 2003.

[36] M. Shahin, P. Liang, and M. R. Khayyambashi. 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*, pages 293–296. IEEE Computer Society, 2009.

[37] P. Sochos, M. Riebisch, and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Prod-

uct Lines. In *IEEE Int'l Conf. on the Engineering of Computer-Based Systems*, pages 308–318, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[38] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[39] H. Tran, U. Zdun, and S. Dustdar. 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)*, pages 105–124. Lecture Notes in Informatics (LNI), 2007.

[40] H. Tran, U. Zdun, and S. Dustdar. Name-based view integration for enhancing the reusability in process-driven SOAs. *Int'l Journal of Business Process Integration and Management*, 5(3):229–239, 2011.

[41] H. Tran, U. Zdun, T. Holmes, E. Oberortner, E. Mulo, and S. Dustdar. Compliance in service-oriented architectures: A model-driven and view-based approach. *Information and Software Technology*, 54(6):531–552, June 2012.

[42] J. Tyree and A. Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22(2):19–27, 2005.

[43] A. van Lamsweerde. From System Goals to Software Architecture. In M. Bernardo and P. Inverardi, editors, *School on Formal Methods*, volume LNCS 2804, pages 25–43. Springer, 2003.

[44] U. Zdun and P. Avgeriou. Modeling Architectural Patterns Using Architectural Primitives. In *20th ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 133–146. ACM, 2005.

[45] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. Reusable Architectural Decision Models for Enterprise Application Development. In *3rd Int'l Conf. on Quality of Software Architectures (QoSA), Medford, MA, USA*, pages 15–32. Springer, 2007.

[46] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann. Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. In *7th Working IEEE/IFIP Conf. on Software Architecture (WICSA), Vancouver, BC, Canada*, pages 157–166. IEEE Computer Society, 2008.