Inconsistency Management Between Architectural Decisions and Designs Using Constraints and Model Fixes

Ioanna Lytra
Software Architecture Research Group
University of Vienna, AT
ioanna.lytra@univie.ac.at

Uwe Zdun
Software Architecture Research Group
University of Vienna, AT
uwe.zdun@univie.ac.at

***Abstract*—The software architecture community has proposed to document the design rationale of software architectures by means of architectural design decisions (ADDs). The constant evolution of software systems requires that both architectural designs and corresponding ADDs are continuously documented and synchronized. However, in practice, designs and ADDs become inconsistent over time. Usually, the potential inconsistencies need to be detected and resolved manually. We propose to alleviate this problem by providing semi-automated support for detecting and handling these inconsistencies. For this, we use constraints for consistency checking between reusable ADDs and component-and-connector (C&C) models. These constraints apply on the C&C models and their invalidation is resolved by software architects either 1) by executing automatically suggested model fixes on the C&C models, or 2) by reconsidering ADDs and subsequently updating the corresponding C&C diagrams, in order to align designs to decisions. We demonstrate our approach in the context of a case study and evaluate its efficiency and scalability.**

I. Introduction

To describe software architectures various architectural views [1–3] for different stakeholders' needs are used. Among these views, the component-and-connector (C&C) model is often considered the one that contains the most significant architectural information [1]. In recent years, software architecture is no longer solely regarded as the solution structure (as modeled in the C&C view), but also as the set of architectural design decisions (ADDs) that led to that structure [4]. The main idea is that by documenting ADDs we can capture and preserve the architectural knowledge and the design rationale.

In practice, ADD documentations often do not get maintained over time [5]. One of the main reasons for this is that documenting ADDs is a tedious and time-consuming task, especially for repeated ADDs [5, 6]. When ADDs are captured, they often do not get synchronized with the corresponding design views, and thus ADDs and designs drift apart as software systems evolve [4]. Until now and to the best of our knowledge, no approach for automated unidirectional or bidirectional translation between ADDs and design views has been proposed. Thus, keeping ADDs and design views consistent and synchronized is a tedious and error-prone manual task.

As models are often used for describing ADDs and architectural views, the problem of keeping ADDs and design views synchronized can be seen as an *inconsistency management problem*. Spanoudakis and Zisman [7] see inconsistency management as a multi-step process composed of 1) the detection of inconsistencies, 2) the diagnosis of the cause of inconsistencies, and 3) the handling of inconsistencies. The management of model inconsistencies (with an emphasis on UML models) has been addressed in various research approaches (see [8–14]). The existing approaches focus on maintaining the well-formedness of models according to specific consistency constraints (as in [10, 11]) and are to a large extent based on predefined repair actions for violations of predefined consistency checking rules [9, 11, 12]. Other approaches address the enforcing of consistency or even bidirectional transformations between architectural views at different levels of abstraction (as in [15]).

The existing approaches can not deal with the handling of inconsistencies between ADDs and architectural views, as each and every decision leads to decision-specific inconsistencies, different to the inconsistencies that might be caused by another decision. In addition, for each caused inconsistency the intention of the software architect has to be considered. In contrast, the other approaches aim to generally resolve inconsistency management problems (e.g. for all models based on the UML meta-model) providing automated fixes. For instance, an inconsistency caused by deleting a component from the C&C model can be resolved either by restoring this component (a strategy akin to the existing approaches listed above), renaming an existing component, or asking the software architect to reconsider the corresponding ADD.

In our previous work [16], we addressed the bridging between ADDs and C&C views by introducing 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, *transformation actions* that apply on the C&C models, as well as *OCL-like constraints* for consistency checking between the ADDs and the C&C models, can be automatically derived using model-driven techniques. For recurring decisions, these mappings can be made reusable, thus reducing implementation time

and effort [17]. Although this approach assists in consistency checking between ADDs and C&C models and the automatic generation of C&C models from ADDs, it does not provide any support for handling potential inconsistencies, whenever decisions and designs evolve. In addition, this approach has not considered reusable ADDs which is the focus of the current approach.

We present in this paper a novel approach aiming to address the semi-automated handling of inconsistencies between reusable ADDs and C&C views along with adequate tool support. Initial architectural designs can be automatically generated from ADDs using transformation actions that act on the C&C models. Constraints with corresponding model fixes can be automatically generated from the underlying ADDs. The validation of these constraints highlights inconsistencies between the ADDs and C&C views and suggests possible fixes that entail either modifications of the view elements or required reconsiderations of existing ADDs.

To demonstrate our approach, we develop a prototype for inconsistency management between decisions modeled in ADvISE[1] – a tool for architectural decision modeling and making support – and C&C views modeled with VbMF[2] – a tool for describing architectural view models (both are tools from our previous work). We use our prototypical implementation in the context of an industrial case study and show that our proposal is efficient and scalable for large numbers of inconsistencies.

The remainder of the paper is structured as follows. First, in Section II we present briefly the tools ADvISE and VbMF. In Section III we discuss the details of our approach giving also illustrative examples. The application of our approach in an industrial case study and its evaluation are presented in Section IV. We compare to related work in Section V and summarize our key contributions in Section VI.

II. Background

In this section we briefly present ADvISE and VbMF, the two tools we integrate for demonstrating our approach for inconsistency management between decisions and designs.

A. Architectural Design Decision Support Framework

The Architectural Design Decision Support Framework (ADvISE) is an Eclipse-based tool that supports the modeling of reusable ADDs using Questions, Options and Criteria (QOC) [18] for decision making. In particular, it assists the architectural decision making process by introducing for a group of design issues a set of questions along with potential options, answers and related (often design pattern based) solutions, as well as dependencies and constraints between them. The advantage of the reusable ADD models is that they need to be created only once for a recurring design situation. In similar application contexts, corresponding questionnaires can be automatically instantiated and used for making concrete decisions, from which architectural decision documentations are generated. For demonstrating our approach for inconsistency management between decisions and designs we use ADDs modeled in ADvISE.

B. View-based Modeling Framework

The View-based Modeling Framework (VbMF) is also an Eclipse-based tool that implements a model-driven, architectural view model. That is, it leverages the notion of view models for describing various concerns of the software systems at different abstraction levels and model-driven development techniques for generating code and configurations from those view models [19]. Among other views, VbMF provides a high-level service component view model – similar to a typical UML component model – for representing essential architectural design elements such as components, ports, connectors, and properties, independently from the underlying platforms and technologies. Technology- and platform-specific information will be described separately in the low-level view models that refine and enrich the high-level counterparts. In this paper, we mainly use the high-level service component view model of VbMF (or in short form, the VbMF C&C view) for describing the architectural design of a software system.

III. Inconsistency Management between ADDs and C&C views

A. Approach Overview

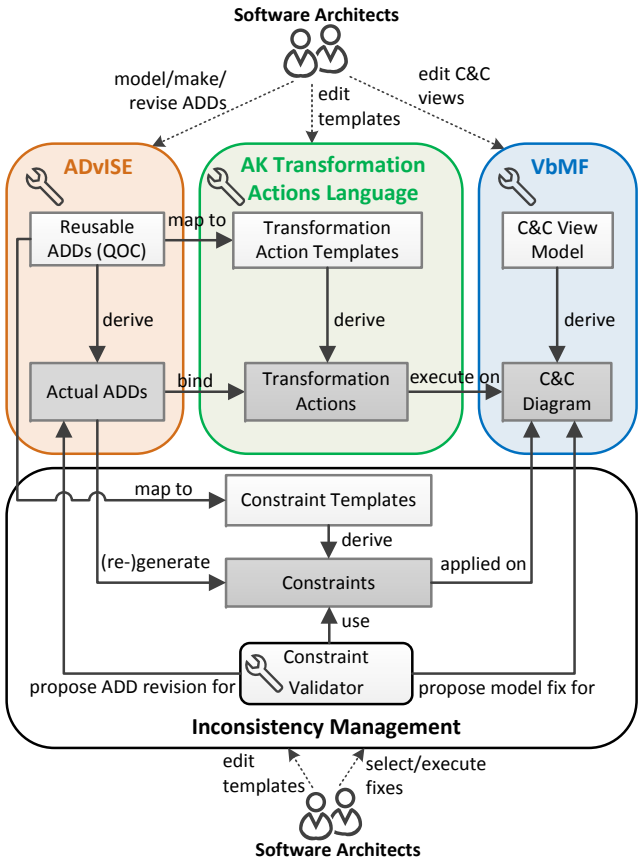

Figure 1. Approach Overview

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

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

The overview of our approach in Figure 1 shows the two existing tools for modeling ADDs and design views – ADvISE and VbMF respectively – as well as their integration through the Architectural Knowledge (AK) Transformation Language that transforms decisions to designs. Our main contribution in this paper is the Inconsistency Management implementation which integrates with both, the ADvISE and VbMF tools. Dark-gray color is used to highlight artifacts that are automatically derived using model-driven techniques, whereas light-gray color highlights manually created artifacts. Tools and toolsuites are depicted in boxes with rounded edges. Please note that ADvISE and VbMF contain multiple tools (e.g., textual and graphical editors, etc.), which for simplicity reasons are not included in this "big picture".

Reusable ADD models based on Questions, Options and Criteria (QOC) (i.e., the artifact *Reusable ADDs (QOC)*) are created using the ADvISE tool for recurring design issues. From these models, ADvISE can automatically generate questionnaires for assisting the making of *Actual ADDs*, which are made possibly multiple times if multiple Questionnaires are derived from the same reusable ADD model. The reusable ADD models are edited by software architects. For using and manipulating *C&C Diagrams* VbMF provides a graphical editor for the *C&C View Model*. Instances of C&C Diagrams can be automatically generated from ADDs using *Transformation Actions*. In VbMF *C&C Diagrams* can be manually manipulated with a graphical C&C view editor.

To check if C&C diagrams still comply to the existing ADDs, OCL-like *Constraints* are generated using predefined mappings between the reusable ADDs and constraint templates. The *Constraint Validator* uses these constraints to check the consistency between actual ADDs and the corresponding C&C diagram. Their invalidation triggers model fixes for the C&C diagrams or recommendations for changing existing ADDs. The selection of the fixes is left to the software architects. The first kind of fixes – model fixes for the C&C diagrams – can be executed automatically, while the second kind of fixes – recommendations for revising ADDs – have to be performed by the architects.

Reusability is achieved by formally mapping *Constraint Templates* to *Reusable ADDs*. The template variables are replaced by actual values as soon as actual ADDs are made. The templates have to be created manually by the software architects, but only once for each reusable ADD model. This way, for *Actual ADDs* we can instantiate the corresponding *Constraints* many times whenever new ADDs are made or existing ADDs are modified.

B. Case Study

In this subsection, we discuss a case study from the area of service-based platform integration which will be used to demonstrate our approach for inconsistency management between ADDs and C&C views. In this case study, three heterogeneous platforms from the industry automation area, a Warehouse Management System (WMS, storage of goods or storage bins into racks via conveyor systems), a Yard Management System (YMS, scheduling, coordination, loading and unloading of trucks), and an Enterprise Resource Planning System (ERP, overall commissioning and handling of goods on an abstract level beyond real storage places) need to provide domain-specific services in an integrated manner. For this, an intermediate integration layer will provide services to operator applications developed on top of it[3]. The design of the integration layer requires many recurring decisions that cover 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, and so on. To address this, we have defined in our previous work a reusable pattern-based ADD model, covering architectural design issues related to integration and adaptation, interface design, communication style, and communication flow [20].

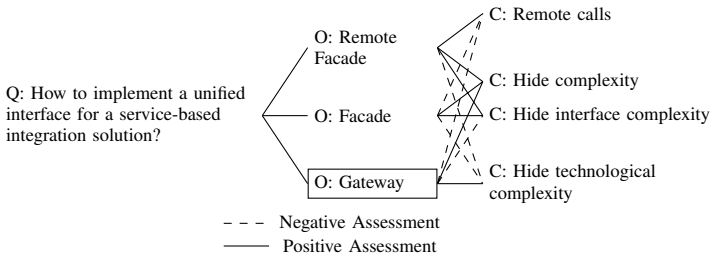

Figure 2. Reusable ADD based on QOC

C. Example of a Reusable ADD

One of the various ADDs that have to be made concerns the design of unified interfaces that will provide integrated services for the three aforementioned platforms. We illustrate this ADD in Figure 2 using the Question, Options and Criteria approach [18]. We provide three alternative solutions for this design issue, namely a "Gateway", a "Facade" and a "Remote Facade" related to four criteria ("Remote calls", "Hide complexity", "Hide interface complexity" and "Hide technological complexity") which are either positively (solid lines) or negatively (dotted lines) assessed. Such a reusable ADD is modeled with the ADvISE tooling and can be reused many times by answering the derived questionnaires.

D. Transformation of ADDs into C&C Views

For transforming reusable ADDs into C&C diagrams a transformation language (AK Transformation Language) [17] is used to express actions that create or update the corresponding C&C models. These transformation actions reflect the ADDs on the C&C views and will be generated from a reusable transformation action template related to the design options of the reusable ADDs. Once an actual ADD is made by the software architect the template parameters will be bound to the concrete decision information. Assume that in our running example we decide to implement a Gateway

[3]Please refer to project INDENICA ((http://www.indenica.eu) for the complete case study.

for subscribing to the Yard Management System notification service from an application build on top of the integration layer. The gateway component ("YMSNotificationGateway") will invoke services from the "Orchestration" component in the integration layer. The transformation actions of Listing 1 will transform the aforementioned ADD in elements of the C&C view. In particular, the execution of the transformation actions will result in the creation of one component, three ports, one connector and one stereotype.

```
add component "YMSNotificationGateway"
add stereotype <<"Gateway">> to Indenica.YMSNotificationGateway
add port "P" kind=PROVIDED to Indenica.Orchestration
add port "R" kind=REQUIRED to Indenica.YMSNotificationGateway
add connector "YMSNotificationGatewayOrchestration" from Indenica
      .YMSNotificationGateway.R to Indenica.Orchestration.P
add port "P" kind=PROVIDED to Indenica.YMSNotificationGateway
```

Listing 1. Transformation actions for implementing the Gateway "YMSNotificationGateway"

The automated transformation of ADDs into C&C views will provide an initial architectural design. However, this unidirectional transformation can not support the evolution of decisions and designs. The reason is that this approach assumes that the design is created from the same initial stage and that existing ADDs and C&C views are not modified. Therefore, it can not cope with the synchronization of ADDs and C&C views when modifications on one or both sides occur, especially when these modifications cause inconsistencies. In this case, software architect's intention to change the ADDs and the corresponding C&C views has to be taken into consideration in order to synchronize ADDs and C&C views.

In the following subsections, we discuss how the inconsistency management between decisions and designs can be addressed by introducing constraints with predefined fixes between the ADDs and C&C views.

E. Detection and Fixing of Inconsistencies

We propose using consistency checking to ensure the integrity of C&C views and their corresponding ADDs. For instance, if implementing an ADD implies (among other things) the creation of a component in the C&C view the consistency is preserved as long as the underlying component exists in the view. In this context, two main sources of inconsistencies exist: 1) the C&C views are modified, so that existing ADDs are not valid anymore, 2) the ADDs are modified, so that the existing C&C view becomes outdated with regard to the ADDs. In our approach, we implement the management of this kind of inconsistencies (i.e., detection and handling) by introducing OCL-based constraints between reusable ADDs and C&C view elements. We implement these constraints using Epsilon Validation Language (EVL) [21], as EVL provides complementary support to OCL for providing user feedback, repairing inconsistencies, and introducing dependent constraints.

As presented in Figure 3 a number of constraints (similar to invariants in OCL) can be defined in a specific context. A context specifies the kind of C&C view elements on which the constraints will be evaluated. Each constraint has a name, an error message and a body (expression), and can additionally define two kind of fixes: either model actions which are

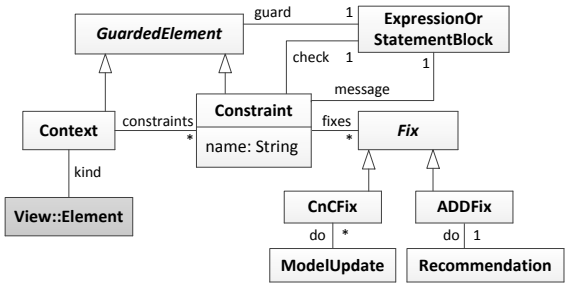

Figure 3. Abstract Syntax of Constraints

automatically applied on the C&C view or recommendations which have to be executed by the software architect.

These constraints do not have to be edited for each ADD separately but are generated from predefined templates mapped to reusable ADDs. These mappings have to be defined once for each reusable ADD and can be "instantiated" many times afterwards, whenever actual ADDs are made or modified. Also, the constraints in template form can be reused among different reusable ADDs. In Table I we list exemplary reusable constraint templates along with error messages and suggested fixes for potential invalidations. All variables to be replaced when the templates are instantiated are indicated with $. Listing 2 is an example of one constraint derived from the ADD discussed before as a running example: the use of a Gateway for the Yard Management System notification service.

```
context ComponentView {
  constraint ComponentYMSNotificationGatewayExists {
    check: self.componentExists("YMSNotificationGateway")
    message: 'Component YMSNotificationGateway does not exist'
    fix {
      title: 'Change existing component to YMSNotificationGateway'
      do {
        var cName: String;
        cName = UserInput.prompt("Select component of "+self.name);
        if (cName.isDefined()) {
          self.getComponent(cName).name = "YMSNotificationGateway";
        }
      }
    }
    fix {
      title: 'Create component YMSNotificationGateway'
      do {
        var newC: new Component;
        newC.name = "YMSNotificationGateway";
        self.element.add(newC);
      }
    }
    fix {
      title: 'Reconsider ADD OperatorInterface.AD1.
           InterfaceComplexity'
      do {
        UserInput.info("Please reconsider the decision and re-
             generate the constraints.");
      }
    }
  }
}
operation ComponentView componentExists(name: String): Boolean {
  return Component.allInstances.exists(c:Component|c.name.equals(
       name));
}
operation ComponentView getComponent(name: String): Component {
  return Component.allInstances.selectOne(c:Component|c.name.equals
       (name));
}
```

Listing 2. Example of a constraint with fixes

Table I
EXEMPLARY REUSABLE CONSTRAINTS WITH FIXES IN TEMPLATE FORM

Constraint	Error Message	Fixes
`Component.allInstances.exists(c:Component|c.name=$comp)`	`Component $comp does not exist`	1) Change existing component to `$comp` 2) Create component `$comp` 3) Reconsider related ADD `$add`
`Connector.allInstances.exists(c:Connector|c.name=$conn and (c.source.name=$a and c.target.name=$b))`	`Connector $conn between $a and $b does not exist`	1) Change connector between `$a` and `$b` to `$conn` 2) Create connector `$conn` between `$a` and `$b` 3) Reconsider related ADD `$add`
`Component.allInstances.exists(c:Component|c.name=$comp and c.port.exists(p:Port\|p.name=$port and p.kind=PortKind#$kind)))`	`Port $port of kind $kind for component $comp does not exist`	1) Change existing port to `$port` of kind `$kind` 2) Change kind of existing port `$port` to `$kind` 3) Create port `$port` of kind `$kind` for component `$comp` 4) Reconsider related ADD `$add`
`Element.allInstances.exists(e:Element|e.name=$elem and e.annotation.exists(p:Property\|p.name=$prop and p.type=$type and p.value=$value))`	`Element $elem does not have property $prop $type=$value`	1) Change existing property `$prop` of `$elem` to `$type=$value` 2) Change existing property of `$elem` to `$prop $type=$value` 3) Create property `$prop $type=$value` for element `$elem` 4) Reconsider related ADD `$add`
`Element.allInstances.exists(e:Element|e.name=$elem and e.annotation.exists(s:Stereotype\|s.name=$stereo))`	`Element $elem is not annotated as $stereo`	1) Change existing annotation of `$elem` to `$stereo` 2) Create annotation `$stereo` for element `$elem` 3) Reconsider related ADD `$add`
`Component.allInstances.exists(c:Component|c.name=$cont and c.nestedComponent.exists(c:Component\|c.name=$comp))`	`Component $comp is not contained in $cont`	1) Move component `$comp` into `$cont` 2) Reconsider related ADD `$add`

The constraint `ComponentYMSNotificationGatewayExists` is used to check if the component `YMSNotificationGateway` still exists in the C&C view. If this evaluates to false it returns the error message `Component YMSNotificationGateway does not exist`. For fixing this inconsistency three alternatives exist: a) an existing component will be renamed to `YMSNotificationGateway` (lines 6–12), b) the component `YMSNotificationGateway` will be added to the C&C view (lines 15–20), or c) the related ADD will be revised (lines 23–26). The first two fixes can be applied automatically on the C&C view while the last fix – recommendation has to be performed by the user. The selection of the fix to be executed is left to the software architect. Every time an ADD changes, the newly generated constraints will check the conformance of the C&C view with respect to the modified ADDs.

F. Dependencies between Fixes

A single modification in the C&C view (e.g., the deletion of a connector) or the revision of an existing ADD may cause many constraints to be invalidated. In the case that many changes have been performed in both sides the software architect will be overwhelmed by a big number of error messages and possible fixes. Apart from that, some fixes may not be executable, as they depend on some preconditions. For instance, checking the existence of a port of a component requires that the component still exists in the C&C view. To reduce evaluated constraints we use guards which limit the applicability of invariants according to the result of an expression. Thus, the evaluation of some constraints is delayed until the preconditions are satisfied after the execution of other fixes.

In our running example, the constraint that checks if the port `YMSNotificationGateway.P` exists (the one created by line 6 of Listing 1) will be validated only if the component `YMSNotificationGateway` already exists in the C&C view or as soon as it is restored by executing another fix, as shown in Listing 3.

By introducing dependencies between the various constraints we reduce on the one hand the number of currently invalidated constraints, and on the other hand we ensure that all proposed fixes are executable.

```
1 context ComponentView {
2   constraint PortYMSNotificationGatewayPExists {
3     guard : ComponentYMSNotificationGatewayExists
4     check : self.portExists("YMSNotificationGateway.P")
5     message : 'Port YMSNotificationGateway.P does not exist'
6     ...
7   }
8 }
```

Listing 3. Example of constraint dependencies

IV. EVALUATION

A. Prototype Implementation

We applied our approach in the context of the case study for service-based platform integration in a warehouse described in detail in Section III. For this, we used the following tools: a) ADvISE for modeling reusable ADDs and making concrete decisions, b) VbMF for editing the corresponding C&C views, c) a prototype implementation for mapping ADDs to constraints in template form and generating concrete constraints, and d) we finally reused EVL Eclipse plugin[4] for editing and

[4]http://www.eclipse.org/epsilon/doc/evl/

validating constraints with fixes applying on the underlying C&C views.

In order to support ADD making the ADvISE tooling provides interactive questionnaires, such as the one shown in Figure 4. From the predefined mappings between the ADDs and constraint templates actual constraints are generated which can be afterwards validated using the EVL tool. The screenshot in Figure 5 displays the result of this validation. In particular, an excerpt of the C&C view from the design of the integration layer in the warehouse, as well as invalidated constraints and proposed fixes for one of the detected inconsistencies are displayed.

Table II summarizes the number of ADDs that were reused in the case study for an excerpt of the ADD model consisting of 6 reusable ADDs (Remote Proxy, Remote Adapter, Result Callback, Request-Ack, Gateway and Facade) classified in 3 categories (Interface Integration, Communication Style and Interface Design), along with the generated constraints, total number of provided fixes, and C&C view elements. Except for the actual ADDs that are made by software architects using tool support based on the ADvISE Questionnaires, all other artifacts are automatically generated from the corresponding templates using model-driven techniques and a template engine.

Table II
REUSABLE ADDS AND GENERATED ARTIFACTS

ADD Category	Reusable ADD	ADDs	C&C Elements	Constraints	Fixes
Interface Integration	Remote Proxy	3	21	24	84
	Remote Adapter	1	7	8	28
Communication Style	Result Callback	7	42	84	280
	Request-Ack	2	12	24	80
Interface Design	Gateway	2	8	12	42
	Facade	1	10	12	43
Total		16	100	164	557

In the following subsections, we evaluate the efficiency and scalability for handling the inconsistencies between ADDs and C&C diagrams for different sizes of ADD models and C&C view models, and also different numbers of detected inconsistencies.

B. Efficiency and Scalability

The results documented in this section are based on the ADDs and C&C views that were captured in the context of our case study. We conducted all our measurements on a normal desktop machine, as our approach will usually run on the local machines of the software architects. The machine for testing had an Intel Quad Core i5 2.53GHz with 8GB of memory running Java VM 1.6 on Debian Linux. All measurements were performed 100 times and the resulting time was reported on average, as the deviations calculated were small.

First, we measured the time needed for validating the constraints checking the consistency of the corresponding C&C views with respect to the actual ADDs. The measurements were conducted with different versions of C&C models in different iterations of the development of the case study. In particular, C&C diagrams or parts of them were used to create the different C&C diagrams of Table III. Afterwards, we validated the generated constraints with fixes for these models. Figure 6 shows that the validation time increases linearly with the number of C&C view elements – at least for the C&C view sizes that we tested – and it remains very low (in a fragment of one second) even for large C&C diagrams, with more than 250 elements (models of such size are rarely used in practice).

Table III
CONSTRAINT VALIDATION TIME

ADDs	6	12	18	24	30	36	42	48	54	60
C&C Elements	24	48	72	96	120	144	168	192	216	240
Constraints	30	56	80	110	137	161	188	218	246	276
Validation time (ms)	54	65	73	88	102	114	141	161	185	214

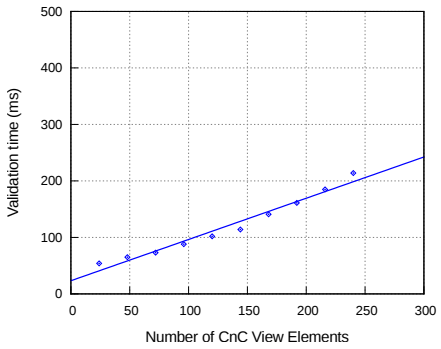

Figure 6. Time for Constraint Validation for Different C&C Model Sizes

In addition, we measured the detected inconsistencies and the number of model updates that would be required for fixing these inconsistencies manually in various situations. Our aim was to estimate the manual effort that is saved by performing the fixes using our approach and accompanying tooling. To produce these inconsistencies we randomly modified (deleted, renamed, etc.) elements of the C&C view from our case study. Afterwards, we modified the related ADDs, and repeated our measurements. The results are presented in Table IV and Table V respectively. In the first case, changing only 15% of the C&C view elements requires that 96 model updates have to be performed manually on the C&C view, while in the second case, changing half of the ADDs will require more than 100 model updates. Please note that these numbers are calculated as mean values based on the setting we created for the case study (see also Table II) and can not be generalizable. They are documented, here, in order to give some indication about the number of inconsistencies that can be caused by changing the ADDs or the C&C views for designs of the size of our case study, as well as the effort for resolving all inconsistencies manually. From both tables we can conclude that even small or medium size changes, either on the ADDs or on the C&C views, lead to a significant amount of required model updates. These model updates can be performed in our approach automatically.

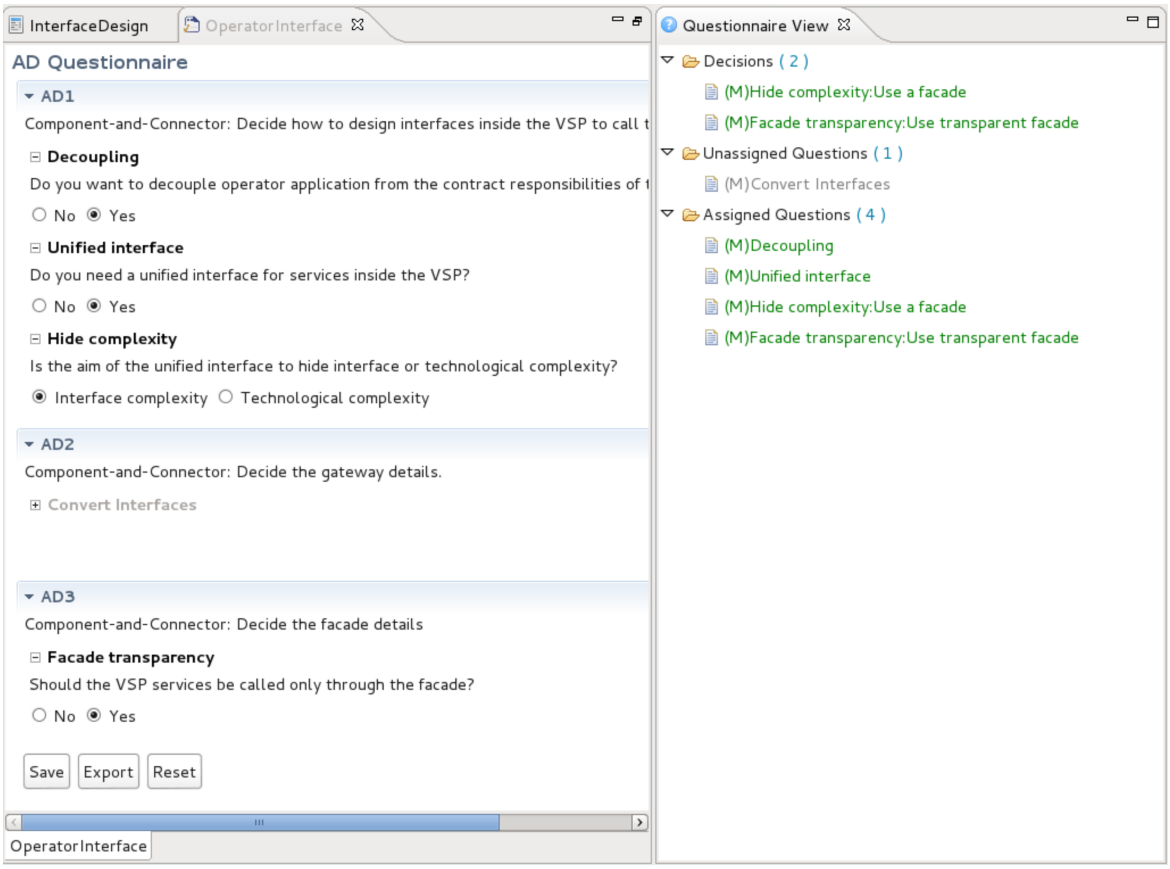

Figure 4. ADvISE Questionnaire

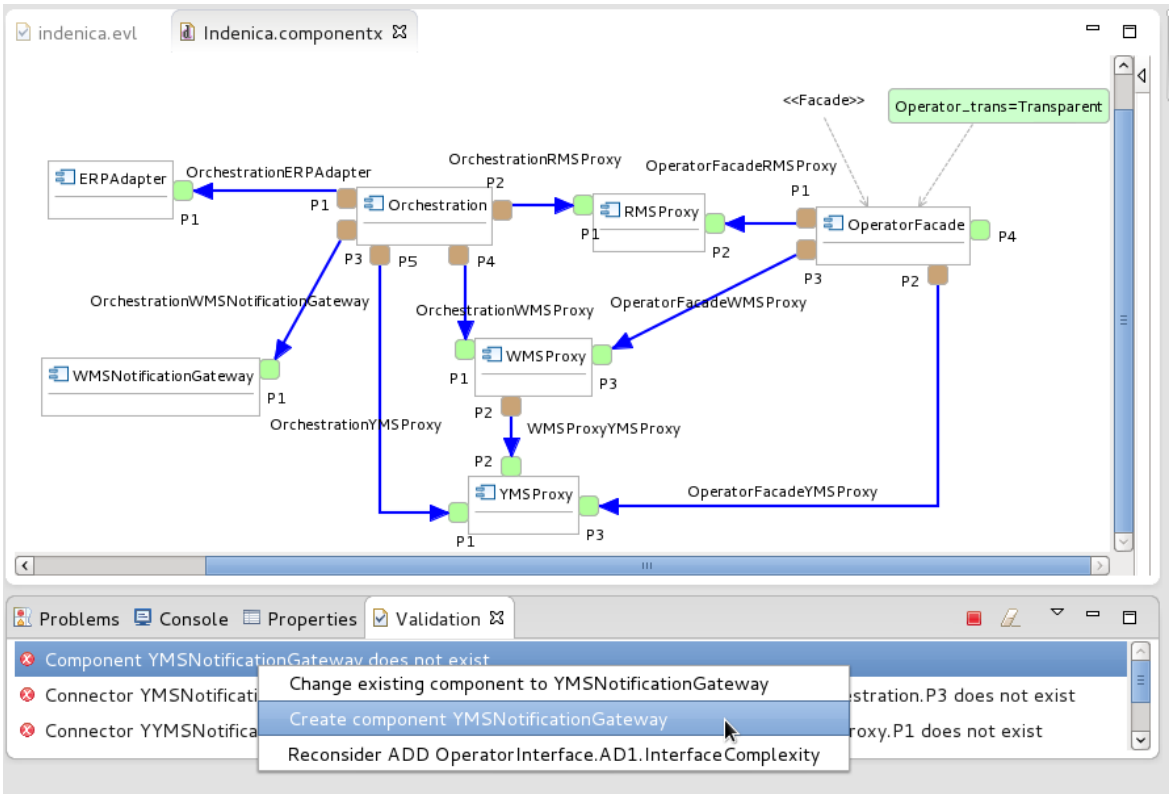

Figure 5. Validation of EVL Constraints

Table IV
C&C VIEW MODIFICATIONS

C&C View Changes	% of all Elements	Inconsistencies	Model Updates Required
2	2%	6	13
5	5%	14	32
10	10%	28	64
15	15%	42	96
20	20%	56	128
25	25%	70	160
30	30%	84	192
50	50%	140	320

Table V
ADD MODIFICATIONS

ADD Changes	% of all ADDs	Inconsistencies	Model Updates Required
1	6%	7	14
2	13%	13	27
3	19%	20	41
4	25%	27	55
5	31%	34	69
6	38%	40	82
7	44%	47	96
8	50%	54	110

C. Discussion and Limitations

Our key contributions are providing semi-automated support for inconsistency detection and handling between reusable ADDs and C&C views, and the high reusability of the assets used for inconsistency management. Our approach does not target the automatic resolution of inconsistencies but rather the suggestion of available fixes (executable fixes or recommendations) to the software architect. Thus, the decision about inconsistency handling is left to the architect and depends on the architect's intention. Due to the big manual effort for keeping ADDs and C&C views synchronized the handling of inconsistencies without automation is very tedious for small numbers of ADDs, C&C view elements and inconsistencies and becomes unfeasible as the number of inconsistencies increases. 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/recurring ADDs, however, with small efforts some non-reusable ADDs can also be integrated to the C&C views.

Our approach is generalizable to other ADD models and architectural views. The constraint 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. Although our inconsistency management approach is based on ADD models based on Questions-Options-Criteria, most of the existing reusable ADD models in the literature [6, 22] contain the notion of trade-offs for alternative options. Also, the constraints can be easily modified to cover similar C&C models or architectural views on different scenarios as well. As the VbMF C&C view contains very similar elements as other typical C&C views our approach is applicable for most of existing component models such as UML component diagram with marginal effort for adapting the actions to accommodate new elements.

In our approach, we use existing general concepts of inconsistency detection and handling [10, 21] and set the focus on the inconsistencies between ADDs and corresponding C&C views. To the best of our knowledge, this is the first approach that addresses the resolution of such inconsistencies between decisions and designs systematically and proposes tool support. Of course, not all aspects of inconsistency management are discussed in this paper, for instance, how to deal with contradicting constraints, how to predict side effects of fixing inconsistencies, and so on. However, many solutions to these issues can be reused from the existing work [10, 12] and get adapted to the context of our work.

We did not perform any usability studies within human designers, thus we were not able to evaluate the usefulness of our approach in the design process and in the long term. Lacking such an empirical evaluation, we are not able to predict the time needed for fixing the inconsistencies by software architects, the advantage of introducing dependencies between the constraints, or the maximum number of ADDs, C&C view elements and detected inconsistencies architects can deal with. An empirical evaluation of our proposal for inconsistency management is, however, part of our future work.

V. RELATED WORK

The documentation of the design rationale as well as the gathering of Architectural Knowledge (AK) have promoted ADDs to first class citizens in software architecture. Most of the approaches based on decision-capturing templates [3], ontologies for architectural decisions [23] and decision metamodels [6], as well as related tools mainly target reasoning on software architectures and reusing of AK and do not tackle the maintenance and synchronization of ADDs with architectural views.

The generation of architectural design views from specifications has been studied extensively in the literature. Kaindl et al. [24] suggest to map requirements to architectural design using model-driven approaches and Grunbacher et al. [25] introduce a mapping from requirements to intermediate models that are closer to software architecture. A different approach by van Lamsweerde et al. [26] derives software architectures from the formal specifications of a system goal model (KAOS) using transformation rules and refines the architectures incrementally using patterns that satisfy quality of service goals like availability and fault tolerance. The aforementioned approaches do not focus on the consistency checking between requirements and architectural views. Another disadvantage compared to our approach is that the rationale that led from the requirements to the architectural views is not documented. That is, because the requirements belong to the problem space, while ADDs belong to the solution space. In all cases, the mapping is unidirectional

and does not allow synchronization between requirements and architectural models.

Our approach is not the first one to relate ADDs to software architectures. The problem of inconsistencies between ADDs and software architectures that cause design knowledge vaporization has been discussed before by Choi et al. [27]. For this, they propose to make ADDs more explicit by introducing a meta-model for relating decisions with architectural elements and a decision constraint graph for representing decision relationships and studying decision change impact analysis. Compared to our approach, this approach demands that most of the work is done manually: decision making, architectural design and change propagation during software evolution. STREAM-ADD [28] 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 AK than on the handling of inconsistencies between decisions and designs.

To support maintenance of architectural decisions as other software artifacts evolve, traceability links between decision models and architecture models have been used. Capilla et al. [29] introduce fine-grained traceability links between design decisions and other software artifacts. The LISA approach [30] captures traceability relations from an architectural component model to design decisions, the code base, and models of architecture-significant requirements in a semi-automatic way. Mirakhorli and Cleland-Huang [31] also 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. The establishment of such traceability links, however, has not been combined with any kind of inconsistency detection and handling between the connected software artifacts.

The consistency checking of software designs and the resolution of inconsistencies have been considered as central activities in software evolution and maintenance, and many approaches have been proposed for inconsistency management. Almeida da Silva et al. [8] use Prolog-based inconsistency rules to detect inconsistencies in UML models and specify their causes from the last changes. A search-based algorithm finds the best plan from a subset of predefined plans expressed as a set of actions applied on the model elements. Egyed [10] proposes to fix inconsistencies in UML design models by executing inconsistency rules, whose execution returns references to the affected model elements. The main idea of this approach is to identify all possible changes and provide to the user the ones which resolve the inconsistency. Nentwich et al. [12] introduce a framework for detecting and fixing inconsistencies in distributed XML documents. The constraints are described as first order logic formulae and the repairs are generated automatically from predefined mappings. In a different approach, Dam and Winikoff [32] describe software designs as agent systems and use Belief-Desire-Intention (BDI) plans derived from OCL rules and from a library of repair plan types to express alternative repair scenarios when the constraints are violated. Other approaches use graph transformation rules to detect and propose resolutions for various types of inconsistencies in UML models [13] or formalize classified inconsistencies, resolution rules conditions and conclusions for UML models using equational logics [33] and description logics [9]. In all cases, the consistency checking rules and the repair plans for resolving inconsistencies are predefined. The aforementioned approaches concentrate mainly on the well-formedness of UML models which is not the case for the inconsistency management between decisions and designs. In our approach, unique constraints are generated for each ADD that is reflected in the C&C diagram, thus, detected inconsistencies have to be handled in each case separately, dependent on the reusable ADD model and its mapping to the corresponding C&C view.

Puissant et al. [14] use an artificial intelligence technique based on automated planning which does not require any manually written resolution rules. In this approach, all repair actions are calculated automatically given the initial state, a set of possible actions and the desired goal. In our case, the handling of inconsistencies is more complex, as potential inconsistencies have to be resolved in two directions: by changing the C&C models or by reconsidering the ADDs and are highly dependent on the architect's intention.

The inconsistency problem can be solved also by propagating the changes from one model to the other and by synchronizing the models using bidirectional model transformations. For example, Chechik et al. [34] introduce an algorithm that locates changes for activity and sequence diagrams and use relationships between their elements to propagate changes. For model synchronization using bidirectional transformations the technique of triple graph grammars is widely used (see, for example, [15]). Such predefined mappings at meta-model level are not possible in our case, as ADDs are reflected in different ways in the architectural design.

VI. Conclusions

Our approach presented in this paper is one of the first attempts for inconsistency management between reusable ADDs and architectural design views, namely C&C views. While we focused in our work on the C&C view, our approach can easily be applied for other design views, like logical views (modeled for instance in class diagrams) or deployment views. In contrast to the related work on inconsistency management of software designs, inconsistency management for ADDs requires decision-specific treatment of inconsistencies, as the concrete architectural knowledge in the decision must be considered for providing the appropriate fixes for inconsistency resolution.

Our approach provides means for automatically detecting and proposing the resolution of inconsistencies to the software architect based on automatically generated constraints with fixes. These fixes can either be applied directly on the C&C view elements or provide feedback to the user for reconsidering an existing ADD. We have applied our approach in an industrial case study on service-based platform integration in

a warehouse, using Eclipse-based tools that implement our proposal. In the context of this case study we discussed the efficiency and reusability of our approach and we showed that the validation of constraints is scalable for large numbers of C&C view elements and constraints. In our future work, we will extend our design space to include more reusable ADD models. In addition, we will study more complex relationships between ADDs and design views in other contexts and case studies. We also plan to evaluate the usability and effectiveness of our tools in empirical studies with software architects from the industry.

ACKNOWLEDGMENT

This work was partially supported by the EU FP7 project INDENICA (http://www.indenica.eu), grant no. 257483.

REFERENCES

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

[2] *ISO/IEC CD1 42010, Systems and software engineering — Architecture description*, ISO, 2010.

[3] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software*, vol. 22, no. 2, pp. 19–27, 2005.

[4] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA), Pittsburgh, PA, USA*. IEEE, 2005, pp. 109–120.

[5] 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 *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, ser. WICSA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 157–166.

[6] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "Reusable architectural decision models for enterprise application development," in *3rd International Conference on Quality of Software Architectures (QoSA), Medford, MA, USA*. Springer, 2007, pp. 15–32.

[7] G. Spanoudakis and A. Zisman, "Inconsistency management in software engineering: Survey and open research issues," in *in Handbook of Software Engineering and Knowledge Engineering*. World Scientific, 2001, pp. 329–380.

[8] M. A. A. da Silva, A. Mougenot, X. Blanc, and R. Bendraou, "Towards Automated Inconsistency Handling in Design Models," in *CAiSE*, 2010, pp. 348–362.

[9] R. V. D. Straeten and M. D'Hondt, "Model refactorings through rule-based inconsistency resolution," in *SAC*, 2006, pp. 1210–1217.

[10] A. Egyed, "Fixing Inconsistencies in UML Design Models," in *29th International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 292–301.

[11] H. K. Dam, L.-S. Le, and A. Ghose, "Supporting Change Propagation in the Evolution of Enterprise Architectures," in *14th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, ser. EDOC '10. IEEE, 2010, pp. 24–33.

[12] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," in *25th International Conference on Software Engineering (ICSE)*, 2003, pp. 455–464.

[13] T. Mens, R. V. D. Straeten, and M. D'Hondt, "Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis," in *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, 2006, pp. 200–214.

[14] J. P. Puissant, R. V. D. Straeten, and T. Mens, "Badger: A Regression Planner to Resolve Design Model Inconsistencies," in *8th European Conference on Modelling Foundations and Applications (ECMFA)*. Springer, 2012.

[15] A. Königs and A. Schürr, "Tool Integration with Triple Graph Grammars - A Survey," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 113–150, 2006.

[16] 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 Conference on Software Maintenance and Reengineering (CSMR), Szeged, Hungary*. Springer, 2012, pp. 287–296.

[17] ——, "Supporting Consistency between Architectural Design Decisions and Component Models through Reusable Architectural Knowledge Transformations," in *European Conference on Software Architecture (ECSA)*. Springer, July 2013, pp. 224–239.

[18] A. MacLean, R. Young, V. Bellotti, and T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," *Human-Computer Interaction*, vol. 6, pp. 201–2502, 1991.

[19] H. Tran, U. Zdun, and S. Dustdar, "View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA," in *International Conference Business Process and Services Computing (BPSC)*. Leipzig, Germany: Lecture Notes in Informatics (LNI), 2007, pp. 105–124.

[20] 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 Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA), Helsinki, Finland*. IEEE, 2012.

[21] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "Rigorous methods for software construction and analysis," J.-R. Abrial and U. Glässer, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. On the evolution of OCL for capturing structural constraints in modelling languages, pp. 204–218.

[22] M. Shahin, P. Liang, and M. R. Khayyambashi, "Architectural design decision: Existing models and tools," in *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), Cambridge, UK*. IEEE, 2009, pp. 293–296.

[23] L. Lee and P. Kruchten, "Capturing Software Architectural Design Decisions," in *2007 Canadian Conference on Electrical and Computer Engineering*. IEEE, 2007, pp. 686–689.

[24] H. Kaindl and J. Falb, "Can We Transform Requirements into Architecture?" in *3rd International Conference on Software Engineering Advances (ICSEA), Sliema, Malta*. IEEE, 2008, pp. 91–96.

[25] P. Grunbacher, A. Egyed, and N. Medvidovic, "Reconciling Software Requirements and Architectures with Intermediate Models," *Software and Systems Modeling*, vol. 3, no. 3, pp. 235–253, 2003.

[26] A. van Lamsweerde, "From System Goals to Software Architecture," in *School on Formal Methods*, M. Bernardo and P. Inverardi, Eds., vol. LNCS 2804. Springer, 2003, pp. 25–43.

[27] Y. Choi, H. Choi, and M. Oh, "An architectural design decision-centric approach to architectural evolution," in *11th International Conference on Advanced Communication Technology (ICACT), Gangwon-Do, South Korea*. IEEE Press, 2009, pp. 417–422.

[28] 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 Conference (COMPSAC), Izmir, Turkey*. IEEE, 2012, pp. 602–611.

[29] 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 Conference in Software Architecture (ECSA), Essen, Germany*. Springer, 2011, pp. 303–318.

[30] G. Buchgeher and R. Weinreich, "Automatic Tracing of Decisions to Architecture and Implementation," in *9th Working IEEE/IFIP Conference on Software Architecture (WICSA), Colorado, USA*. IEEE, 2011, pp. 46–55.

[31] M. Mirakhorli and J. Cleland-Huang, "Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance," in *27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA*. IEEE, 2011, pp. 123–132.

[32] H. K. Dam and M. Winikoff, "An agent-oriented approach to change propagation in software maintenance," *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 3, pp. 384–452, Nov. 2011.

[33] A. Boronat and J. Meseguer, "Automated Model Synchronization: A Case Study on UML with Maude," *Electronic Communications of the EASST*, vol. 41, 2011.

[34] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh, and R. Salay, "Relationship-based change propagation: A case study," in *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, ser. MISE '09. IEEE, 2009, pp. 7–12.