Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution

Ioanna Lytra, Huy Tran, Uwe Zdun
Software Architecture Research Group
University of Vienna, Austria
Email: {*ioanna.lytra*|*huy.tran*|*uwe.zdun*}@*univie.ac.at*

***Abstract*—Software architecture evolution has become an integral part of the software lifecycle. Thus, the maintenance of a software system involves among others the maintenance of the software system architecture. Component models are widely used as an essential view to describe software architectures. In recent years, the software architecture community has proposed to additionally model the architectural design decisions for capturing the design rationale and recording the architectural knowledge. Unfortunately, there are no formal relations between design decisions and component models. This leads to potential inconsistencies between the two kinds of models as the software system evolves. In this paper, we propose to overcome this problem by introducing a constraint-based approach for checking the consistency between the decisions and the corresponding component models. Our approach enables explicit formalized mappings of architectural design decisions onto component models. Based on these mappings, component models along with the constraints used for consistency checking between the decisions and the component models can be automatically generated using model-driven techniques. Our approach can cope with changes in the decision model by re-generating the constraints for the component model. Thus, our component model gets updated and validated as the architectural decisions evolve. The evaluation of our approach shows that our prototypical implementation scales sufficiently for large component model sizes and large sets of decisions.**

***Keywords*-software architecture; software architecture evolution; architectural decisions; architectural knowledge; component models; constraint checking**

I. Introduction

Documenting the design decisions as well as the architecture of a software system does not belong only to the early software development phases but is realized repeatedly in the software lifecycle. Architectural Design Decisions (ADDs) capture knowledge that may concern a software system as a whole, or one or more components of a software architecture. In recent years, software architecture is seen more and more as a set of principal ADDs rather than the components and connectors constituting a system's design [1]. The idea behind this new perspective is to document not only components and connectors but also the design rationale of the architecture as well as to contribute to the gathering of Architectural Knowledge (AK). For this reason, many approaches have been proposed for the capturing of AK. Tyree and Akerman defined a rich decision-capturing template [2]. Kruchten et al. presented an ontology for architectural decisions, defining types of ADDs, dependencies between them and a decision lifecycle [3]. Zimmermann et al. suggested a meta-model for decision capturing and modeling [4]. A considerable amount of tools have been developed to ease capturing, managing and sharing of ADDs [5]. These approaches mainly target the reasoning on software architectures, capturing and reusing of AK as well as on the communication of the ADDs between the stakeholders. Furthermore, by documenting the architectural decisions software architects can reason about the evolution, maintenance and reengineering of a software system. Unfortunately, in practice, the ADDs frequently do not get maintained over time as the requirements and the design of the software system change and they often do not get synchronized with other architectural views that represent the system structures [6].

Many software systems today are based on reusable software components, and Component-based Software Engineering (CBSE) has gained a lot of prominence in industry [7–9]. Component-and-connector diagrams offer a natural presentation to the software architects and designers to describe the software architecture using reusable, customizable and replaceable entities. In many enterprises today software architecture is mainly documented using component-and-connector diagrams, which are in many cases used in an informal or semi-formal fashion (e.g., as box-and-line diagrams). However, architectural documentations that concentrate only on components and connectors have many disadvantages, such as limited reusability of and reasoning about AK, and lack of stakeholders communication [10]. That happens because they derive from requirements and ADDs in a rather informal way and the consistencies with the ADDs depend highly on the interpretation and understanding by the software designer. The lack of a formal mapping of the ADDs and architectural views leads to inconsistencies and low traceability. Consequently, the reuse of derived component diagrams along with the reuse of AK is not possible. Component and connector based methods of architecture documentation, however, manage to visualize the software architecture in terms of software components as well as the wiring between them, the data flow and the properties associated to them.

Our work presented in this paper aims at bridging the gap between requirements and design, between ADDs and architectural views, combining ADDs with component models in a formal way. For this purpose, we propose a mapping model from ADDs onto a component model. Without loss of generality, a generalized component model (that can easily be mapped to popular component-based modeling approaches such as the UML 2 Component Diagram[1]) is considered in our approach for describing software architecture. Based on the mapping model, we can generate the component model and constraints for consistency checking between ADDs and the component model by using model-driven techniques. During the evolution of the software systems, the ADDs may be altered. Accordingly, the component model and the constraints can be re-generated, and therefore, remain in sync with the ADDs. As the component model is changed, the constraints checking shall verify whether these changes invalidate the corresponding ADDs or not. In case inconsistencies between the ADDs and the component model occur, the relevant design elements that invalidate the ADDs shall be highlighted.

The remainder of the paper is structured as follows. Section II presents an industrial case study as a motivating example and provides the problem statement and Section III presents an overview of our approach. In the context of this case study, we describe in detail our solution in Section IV. The case study is then revisited and resolved using this approach in Section V. Next, we evaluate our approach and discuss our findings in Section VI. The related work shall be discussed in Section VII and we conclude our major contributions in Section VIII.

II. Case Study and Problem Statement

To motivate our work, let us first discuss the problem using an industrial case study that deals with platform integration. In Section V, we shall explain how our solution can be applied to address the case study. The scenario is centered on a warehouse management system that is used to control the movement and storage of goods and materials in a warehouse. A product is delivered to the warehouse after it has been produced and assembled. A material flow computer (MFC) contains a registry of all slots and racks in the warehouse and tracks the delivery and the positioning of a product into a certain rack using actuators and sensors. The product orders are managed by an ERP system that communicates with the MFC through an integration platform. We call this kind of platforms Virtual Service Platform (VSP). Here we target the adaptation of parts of the aforementioned platforms to the VSP. We exemplify an excerpt of the system so that our example is still simple, comprehensible, and illustrative enough for elaborating our approach in detail. For this reason, we shall concentrate on the ADDs and designs that are influenced by the following excerpt of the case study requirements:

1) The orders are placed at the ERP system and forwarded to the MFC through the VSP.
2) All the messages sent from the ERP system are encrypted and compressed.
3) The MFC must be able to handle 1000 orders per minute.

In this context, the architectural decisions corresponding to the aforementioned functional and non-functional requirements shall be documented. Besides, the architecture of the system shall be designed accordingly to fulfill these requirements. The issue occurring in this case is that the documented ADDs and the relevant design are not explicitly and formally connected. As the number of services integrated to the VSP increases, the number of ADDs to be documented also increases and the architectural design becomes more and more complex. As long as the ADDs and designs remain disconnected, the corresponding designs relating to the ADDs likely become outdated during the evolution of software system. Even worse, the ADDs may become inconsistent with the design. In our approach, we demonstrate how this gap between the ADDs and the design is reconciled, and therefore, the ADDs and design remain consistent.

Our approach starts at the development stage where the requirements have been resolved into the ADDs. The ADDs are usually captured in an informal way using document templates or meta-models [2, 4]. The key concepts that our approach focuses on are architectural decisions and their implications. Therefore, most of existing approaches for capturing and representing ADDs can be applied because most of them provide these essential concepts. In this case study, the architecture decision description template proposed in [2] is exemplified to capture ADDs. We show an excerpt of the documented ADDs including three architectural decisions **D01**, **D02**, and **D03** in Table I.

D01	Expose Place Order functionality as Apache CXF Web Services.
D02	Connect Place Order to VSP using encrypted HTTPS connection and compress the messages using standard HTTP/1.1.
D03	Implement Order Picking as a BPEL flow that is able to handle asynchronously 1000 orders per minute.

Table I
Exemplified architecture design decisions of the case study

At this stage, we have necessary information that shall be used as inputs for our approach. But before we go deeper into the details of our solution, let us briefly introduce the software architecting and design process. Software architects and designers use the requirements in order to make and document the design decisions and design the details of

[1]Section 8 in http://www.omg.org/spec/UML/2.2/Superstructure/PDF

the architecture. In reality, the two roles can either be played by the same person or distinguished stakeholders who involve in both the decision making and designing. Architects and designers use templates or meta-models to capture the ADDs and component models to document the software architecture – sometimes represented in terms of informal box-and-line diagrams or formulated using a formal or semi-formal modeling techniques such as UML2 Component Diagrams.

The consistency between the ADDs and the component models depends highly on the understanding and interpretation of the requirements and on how often they get updated as the software system evolves. In the enterprise, ADD capturing and architectural design are performed sometimes independently and from different stakeholders and have both the requirements as a starting point. Also, some ADDs get forgotten or not updated at all. Besides, the software systems are constantly subject to a substantial amount of evolutionary changes that reflect new or modified functional and non-functional requirements, new or modified design decisions. Consequently, the software system evolution potentially leads to inconsistent and outdated ADDs. Apart from that, as the complexity and coupling of software systems increase, the traceability as well as the rationale of the design may be lost. The process of maintaining the traceability and consistency between ADDs and software architecture is not trivial as a single ADD may address multiple architectural concerns and an architectural element may be connected to multiple ADDs. Furthermore, ADDs and software architecture often use different terms and concepts to describe model elements related to them although they may both contain identical or related information. Our observations show that the main reason for this issue is that ADDs and component models are formulated and manipulated independently because there are no explicit formal connections between them.

III. Approach Overview

Our contributions to the software architecture and design process are summarized in Figure 1. Our approach is useful not only at design time but also during software system evolution and maintenance. We aim at bridging ADDs and component models by introducing a mapping between them based on the model-driven development approach. The great advantage of the mapping is to enable traceability and consistency checking between ADDs and component models and automate the generation of an initial instance of a component model that reflects the design decisions. Apart from the generation of the component model, we introduce the generation of constraints that are used for consistency checking between the ADDs and the component model. As the human decision is important in the interpretation of the ADDs and their connection to software architectures, the mapping from ADDs to the component model shall be performed in a semi-automatic manner. After the mapping is established, the generation of component models and constraints is fully automated. As such, the software architects and designers can use our tool to analyze and estimate how the changes of certain ADDs shall affect the design and/or leverage the generation to come up with a recommendation design directly derived from the ADDs rather starting from scratch. Apart from that, we aim at reducing the cost and burden of the maintenance of both ADDs and component models. Information included in the ADDs shall be fully reflected in the component model. Changes at the component model or the ADDs that cause inconsistencies get highlighted. Hence, we aim at ensuring that ADDs and component models remain consistent with each other and component models are traceable from ADDs and vice versa.

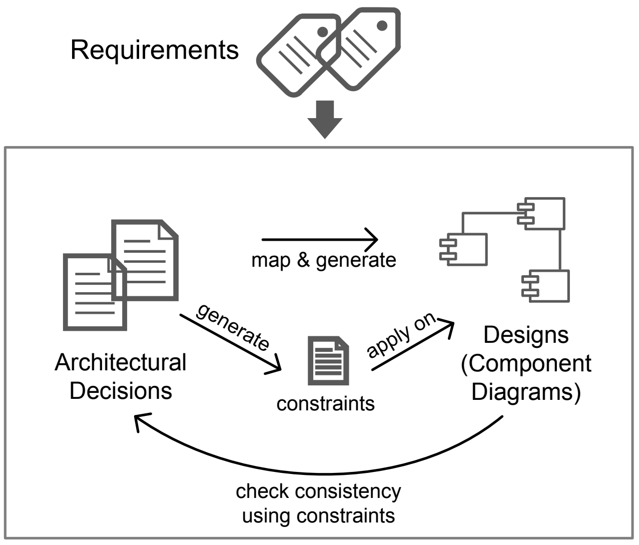

Figure 1. Mapping and consistency checking between ADDs and Designs

IV. Solution Details

In this section we introduce a generic component model used for describing system and software architectures. Next, we introduce a mapping model from ADDs to the component model. We elaborate afterwards this mapping model for the generation of the component model and of constraints for checking consistency using model-driven techniques. We revisit the case study and explain how our approach can be applied to bridge the gap between ADDs and the designs.

A. Component Model

Component models are often used as an essential view for describing software architectures [11]. Without loss of generality, we propose the use of a generic, abstract component model built upon the essential concepts such as components, ports, and connectors to specify the underlying system and software architectures. Other component-and-connector models such as the UML 2 Component Diagram are applicable in our approach as well.

For the sake of integrating different models, we define a Core model that provides essential concepts such as *NamedElement* and *AnnotatedElement* (see Figure 2). A *View* and its *Element* are abstractions of a design model and its model elements, respectively. A component model (see Figure 3) reflects a view of software architectures that comprises various *Component*s wired by several *Connector*s via the component's *Port*s. A component may have several subcomponents. We propose *Property* and *Stereotype* elements to enrich *Components*, *Connectors* and *Ports* with additional information.

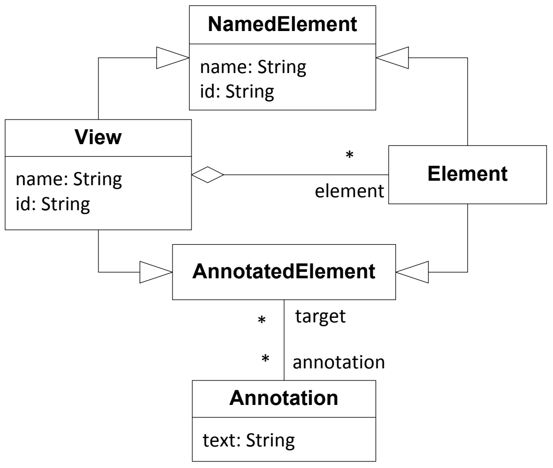

Figure 2. Core Model

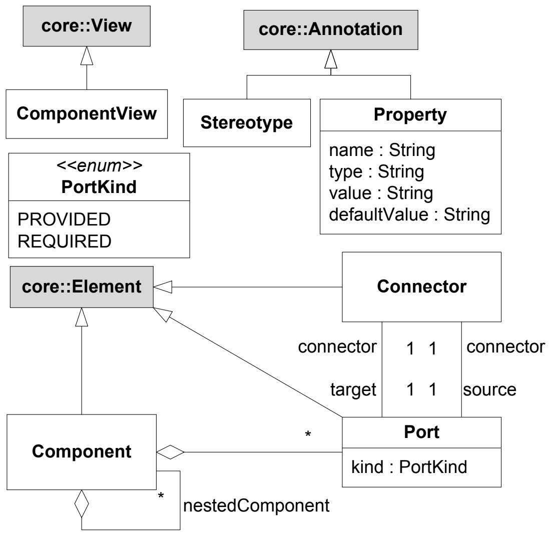

Figure 3. Component Model

B. *Mapping of ADDs to Component Model*

Capturing architectural design decisions is important for analyzing and understanding the rationale and implications of these decisions and reducing the problem of architectural knowledge vaporization [11, 12]. Several existing approaches have been proposed for addressing the aforementioned challenges [2, 4]. However, none of these approaches supports to explicitly capture the causal relationships between the decisions and relevant design artifacts. These relationships are crucial because they enable the traceability between ADDs and the designs for analyzing the design coverage (e.g., checking whether some ADDs have been realized or not), estimating change impacts (e.g., which design artifacts are affected by certain changes of ADDs), checking consistency between ADDs and the designs, and many other tasks. In this paper, formalizing these relationships in order to bridge the gap between ADDs and the designs as well as using them for generating component models and constraints for checking consistency are the key contributions.

We propose a generic concept, namely, *AD*, for representing ADDs (see Figure 4). Each AD has a number of *Outcome*s which are inputs for designing component models. An *Outcome* can be mapped to a certain element of the component model shown in Figure 3. The ADD's *Outcome*s are often expressed in natural language, and therefore, human interventions are necessary for instantiating the Mapping model. For instance, if an *Outcome* implies a new property of a component, the name of the component and the name, type, and value of the property should be defined manually.

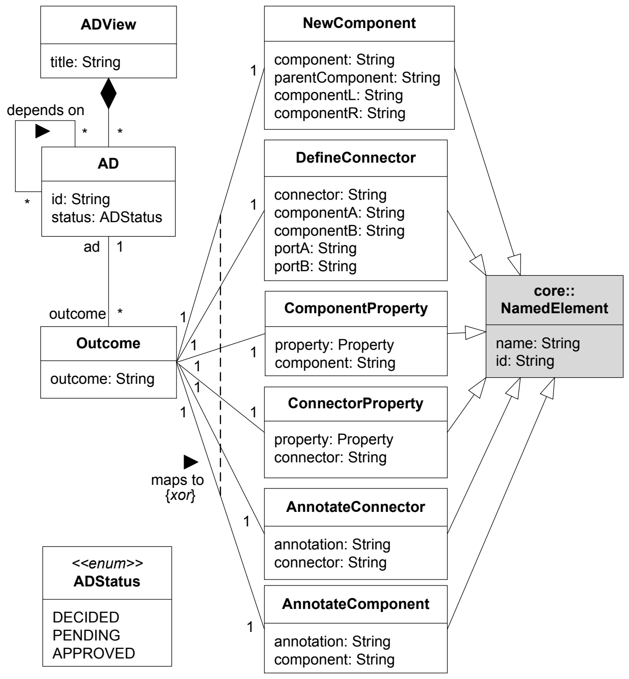

Figure 4. An excerpt of the Mapping model

C. *Generation of Component Model*

The Mapping model presented in the previous section shall connect the architectural decisions and designs at various levels of granularity. Each mapping represents a relationship between a certain *Outcome* of an ADD to an element of the design (in this case, the component model) such as a component, port, connector, an annotation, property, and so on. Among the benefits of the Mapping model mentioned before, we can also leverage these mappings to generate an initial component model that can serve as a starting point for designing the corresponding software architecture. In case of a green-field development scenario (i.e., there are no existing designs), this step can save tedious efforts that the software architects and designers have to spend to sketch the designs from scratch. Nevertheless, in case there are existing designs, the generated designs can be referenced for analyzing the deviation as well as estimating necessary changes in order to accomplish the architectural decisions. In addition, constraint checking can be performed beforehand to ensure that the component model can be updated without any errors.

The constraint-checking at this stage is necessary not only for generating the component model but also for finding out the issues due to that the component model cannot be generated, if any. For instance, assume that we want to assign a property to a connector linking the ports of two components. We illustrate the templates that will be used for the generation of the constraints that will be checked before the generation of the component model. The variables within the notion `$..$` (e.g. `$componentA$`, `$portA$`) shall be substituted with concrete values during the constraint instantiation.

$$\exists\, c \in Components \mid c.name = \$componentA\$$$

$$\exists\, c \in Components \mid c.name = \$componentB\$$$

$$\begin{aligned}&\exists\, c1, c2 \in Components \mid c1.name = \$componentA\$\\&\wedge c2.name = \$componentB\$\\&\wedge (\exists\, con \in Connectors)\\&\wedge (\exists\, p1 \in c1.ports,\ \exists\, p2 \in c2.ports \mid\\&(p1.name = \$portA\$ \ \wedge\ p2.name = \$portB\$)\\&\wedge ((p1 = con.source \ \wedge\ p2 = con.target)\\&\vee (p1 = con.target \ \wedge\ p2 = con.source)))\end{aligned}$$

Whenever a certain constraint is not satisfied, specific errors shall be reported. The stakeholders have to fix those errors before the component model can be generated properly. The component model gets successfully generated if and only if all constraints are satisfied.

Each kind of mapping might imply one or more consequent updates to the component model. For example, suppose that we want to assign the property to a connector, after the constraints are checked and satisfied, a new property should be created and annotated to the connector.

We implement the constraint checking using the the declarative constraint checking language *Check*. The component model is generated based on the mapping model by using the expression language *Xtend*. *Xtend* and *Check* are powerful OCL-like expression languages provided by the Eclipse Model-to-Text (M2T) project[2]. The process of constraint-based model validation and component model generation is integrated using the modeling workflow language provided in the Eclipse M2T project.

D. *Generation of Consistency Checking Constraints*

Each of the aforementioned types of mapping is related to a set of constraint templates from which concrete constraint "instances" are generated. The constraint templates have been already defined for each kind of mapping. The constraint "instances" are generated using the Velocity template engine[3] and the attributes to be replaced get values from the mapping of the ADDs to the component model. The generated constraints are also based on the *Check* language. We illustrate one of the constraint templates that is used for creating constraints on the mapping of an ADD to a new property of a certain component. As mentioned above, the variables within the notion `$..$` shall be substituted with concrete values during the instantiation of the constraints.

$$\begin{aligned}&\exists\, p \in Properties \mid\ p.name = \$name\$\\&\wedge p.value = \$value\$\\&\wedge\ (\exists\, c \in Components \mid c.name = \$component\$)\\&\wedge\ (\exists\, a \in c.annotations \mid a = p)\end{aligned}$$

These constraints are generated and validated as described in Section IV-C for the constraints that have to be checked before the component model generation. The consistency checking constraints shall check the consistency between the ADDs and the component model.

V. Case Study Resolved

In order to illustrate our approach let us revisit our case study – the Warehouse Case Study. A material flow computer in a warehouse receives orders from an ERP system and the communication between them is accomplished through a Virtual Service Platform (VSP). The proof of concept tooling of our approach based on EMF[4] and GMF[5] is shown in Figure 6. Our tool can support the development and generation of the constraints as well as the generation and the graphical representation of the component model.

A. *Mapping of ADDs to Component Model*

We extract the useful information of the ADDs that can be mapped to a component, a connector, an annotation, or a property. Figure 5 presents an excerpt of the Mapping model between the ADDs and the component model. For example,

[2]http://www.eclipse.org/modeling/m2t
[3]http://velocity.apache.org
[4]http://www.eclipse.org/emf
[5]http://www.eclipse.org/gmf

the first ADD refers to a new component (*Place Order*) connected to the *VSP* component which will be implemented as a *Web Service* (annotation of component *Place Order*) using *Apache CXF* (property of component *Place Order*).

B. Component Model Generation

Once the mapping of the ADDs to the component model is accomplished, we can generate an initial instance of the component model. In order to see how the generation is done, let us take the *AnnotateComponent* mapping from the decision **D01** (see Table I). Before creating a new annotation and attach it to a component, we must ensure that the same annotation has not been already assigned to the component (see Listing 1).

If the above function for our component graphical view and for $component = \text{“}PlaceOrder\text{”}$ and $annotation = \text{“}WebService\text{”}$ does not return any annotation we proceed with the creation of the new annotation (see Listing 2).

```
component::Stereotype componentHasAnnotation(
    component::ComponentView cv, component::
    Component component, String annotation):
    component.annotation.select(a|(a.text ==
        annotation) && (cv.annotation.typeSelect(
        component::Stereotype).exists(s|s == a)));
```

Listing 1. Constraints for verifying a component's annotation

If the generation of the component model completes without errors, the visualization of the component model that we get using our Eclipse Tooling is shown in Figure 6. A component is depicted in terms of a box associated with its ports. Two ports can be connected by a connector which is an arrow going from the required to the provided one. The stereotypes are shown inside the symbol “≪≫” (e.g., ≪HTTPS≫) and the properties are shown in form of “*name[:type]=value*” (e.g., “technology=Apache CXF”).

```
annotateComponent(component::ComponentView cv,
    component::Component component, String
    annotation):
  let stereotype = new component::Stereotype :
  stereotype.setText(annotation)
  -> cv.annotation.add(stereotype)
  -> component.annotation.add(stereotype);
```

Listing 2. Creating a new annotation for a component

C. Generation of Consistency Checking Constraints

Now we explain how the constraints that check the consistency of the component model get generated. Let us consider the *AnnotateComponent* mapping from the ADD **D01** (see Table I). The *AnnotateComponent* is mapped to the constraint template shown in Listing 3 that checks whether a component is associated with a specific stereotype or not (note that a variable is inside the notion `$..$`).

```
context component::ComponentView ERROR
        "(Architectural Decision --> $ad$)
             Component $component$ is not annotated
              as $annotation$":
        element.typeSelect(component::Component).
             exists(c|c.component == "$component$"
             && annotation.typeSelect(component::
             Stereotype).exists(s|c.annotation.
             exists(a|a == s && s.text == "
             $annotation$")));
```

Listing 3. The constraint template for checking a component's annotation

In our example, the component “Place Order” shall be annotated as “Web Service” (i.e.,g *annotation=Web Service*) according to the decision ADD **D01**. The resulting instantiated constraint is shown in Listing 4.

```
// Check whether component is annotated
context component::ComponentView ERROR
        "(Architectural Decision --> D01)
             Component Place Order is not annotated
              as Web Service":
        element.typeSelect(component::Component).
             exists(c|c.name == "Place Order" &&
             annotation.typeSelect(component::
             Stereotype).exists(s|c.annotation.
             exists(a|a == s && s.text == "Web
             Service")));
```

Listing 4. Constraint instances for checking a component's annotation

The above example illustrates a great advantage of our approach: a constraint template shall be defined once but can be efficiently reused and instantiated for several corresponding mappings from the ADDs to the component model.

VI. Evaluation

A. Generalisability

Our approach is to a large extent generic. We use an abstract component model to map ADDs to components, connectors, stereotypes, and properties, and we show how the constraints can check the consistency between ADDs and component models and allow updates of the component model. The resulting component diagram in Figure 6 contains all the information captured by the architectural decision template, the mapping however is conducted by the choice of the specific component model. Our approach can be applied for more complicated and domain-specific models which requires of course additional efforts for defining the mapping of the ADDs to the model and necessary constraint templates.

B. Scalability and Applicability

In order to show that our approach is scalable and applicable we conducted experiments for different sizes of component models. We measured the performance on a normal desktop machine, as our approach will usually need to run on the local machines of the software architects and designers. The machine has an Intel Core i5 2.53GHz

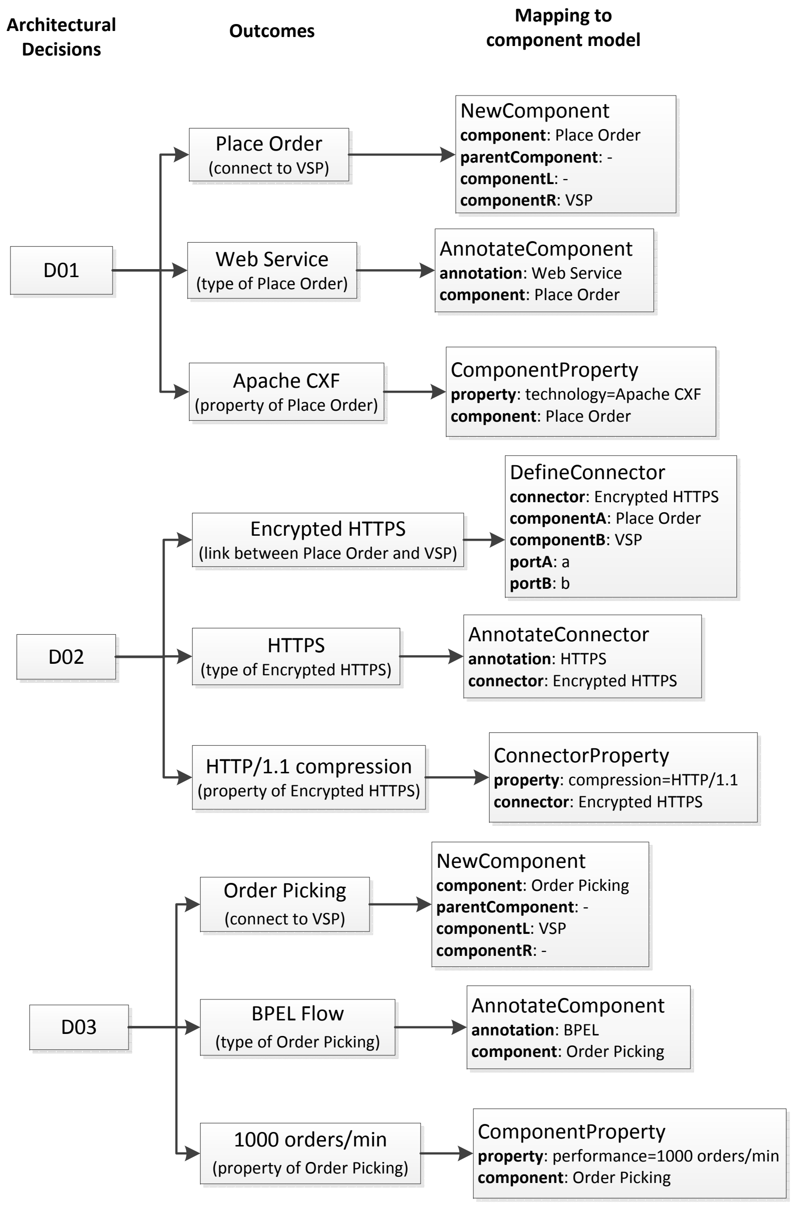
Architectural Decisions
Outcomes
Mapping to component model
D01
Place Order (connect to VSP)
NewComponent
component: Place Order
parentComponent: -
componentL: -
componentR: VSP
Web Service (type of Place Order)
AnnotateComponent
annotation: Web Service
component: Place Order
Apache CXF (property of Place Order)
ComponentProperty
property: technology=Apache CXF
component: Place Order
D02
Encrypted HTTPS (link between Place Order and VSP)
DefineConnector
connector: Encrypted HTTPS
componentA: Place Order
componentB: VSP
portA: a
portB: b
HTTPS (type of Encrypted HTTPS)
AnnotateConnector
annotation: HTTPS
connector: Encrypted HTTPS
HTTP/1.1 compression (property of Encrypted HTTPS)
ConnectorProperty
property: compression=HTTP/1.1
connector: Encrypted HTTPS
D03
Order Picking (connect to VSP)
NewComponent
component: Order Picking
parentComponent: -
componentL: VSP
componentR: -
BPEL Flow (type of Order Picking)
AnnotateComponent
annotation: BPEL
component: Order Picking
1000 orders/min (property of Order Picking)
ComponentProperty
property: performance=1000 orders/min
component: Order Picking

Figure 5. Mapping of architectural design decisions to the Component Model

processor with 4GB of RAM running Java JRE 1.6.0.24 and Eclipse Helios in Linux Ubuntu.

We first measure the number of constraints per number of components and the number of components per ADDs. After that, we measure the time needed for the generation of the component model, for the generation of the constraints and the constraints checking. Each measurement is performed 100 times and the resulting time, in milliseconds, is calculated on average. The deviations calculated were not high, thus only the average and not the minimum and maximum values are reported. Table II contains the number of constraints that were generated for a given number of ADDs and components. Figures 7–9 show the time needed for the generation of the component diagram, for the generation of the consistency checking constraints and for the validation of these constraints per number of components respectively.

The time for the component model generation increases exponentially with the number of components, however, it remains low for component models having about 100 components. The time for the generation of the constraints increases in linear to the number of components. For 30 components the generation of the component model and the constraints needs about 70 and 570 ms, respectively, for 60 components approximately 130 and 1060 ms and for 150

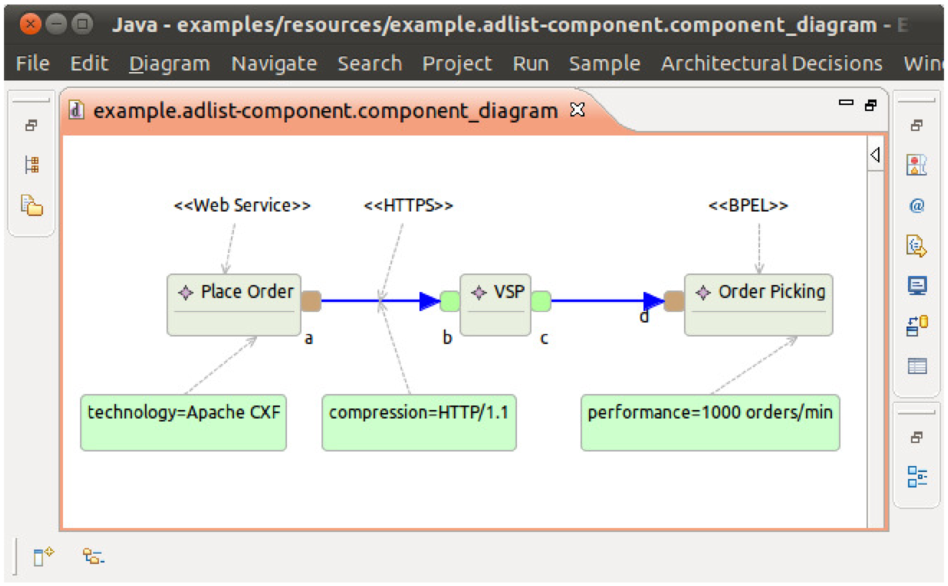
Java - examples/resources/example.adlist-component.component_diagram - E
File Edit Diagram Navigate Search Project Run Sample Architectural Decisions Win
example.adlist-component.component_diagram
<<Web Service>>
<<HTTPS>>
<<BPEL>>
Place Order
VSP
Order Picking
a
b
c
d
technology=Apache CXF
compression=HTTP/1.1
performance=1000 orders/min

Figure 6. Eclipse Tooling for ADD and Component Model Development

Table II
NUMBER OF COMPONENTS AND CONSTRAINTS PER NUMBER OF ADDS

ADDs	3	30	60	150	210	300	360	450
Components	3	50	100	250	350	500	600	750
Constraints	10	130	260	650	910	1300	1560	1950

components 400 and 2450 ms respectively. Regarding very big component models having around 500 components, the generation time of the component model and the constraints completes within 4–8 seconds. We are mainly interested in the range of 10–100 components, as even for big software projects, the component diagrams do not usually contain more than 100 components. Regarding this range of component model sizes our approach performs reasonably well.

The constraint checking time increases, as expected, exponentially with the number of components. For 30, 60 and 150 components the constraint checking takes about 200, 1200 and 18600 ms, respectively, to complete. A typical component diagram however, as mentioned before, does not consist of more than 100–150 components implying that the constraint checking time is acceptable. Apart from that, constraint checking does not need to run very often, and therefore, an execution time of few seconds is for this reason as well acceptable. For larger component models, a more powerful computer and/or a faster or incremental constraint checker would definitely improve the performance.

C. Limitations

First, bidirectional generation is not considered in our approach. That is, the designs (i.e., component models) can be derived and generated from ADDs but not the other way around. Such a reverse engineering solution, if even

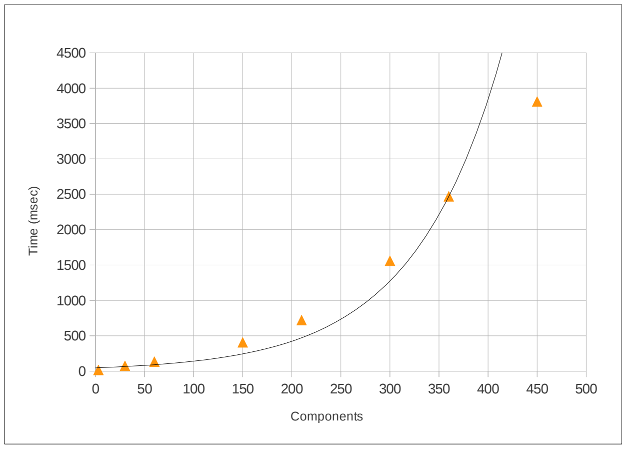

Figure 7. Time for generation of component diagram per number of components

exists, may be useful for abstracting and analyzing existing legacy designs. Unfortunately, it is a extremely challenging for harvesting rationale behind the ADDs in the reverse direction, i.e., extracting ADDs from the designs, because many necessary information on architectural decisions are not often embedded in the designs.

Second, the mapping between the ADDs and the component models is done in a semi-automatic manner and sometimes requires human interventions. That might be time-consuming for considerably large numbers of ADDs. This is one of our future endeavors to investigate techniques for enhancing the automation in deriving the aforementioned mapping, and therefore, reducing the involvement of the

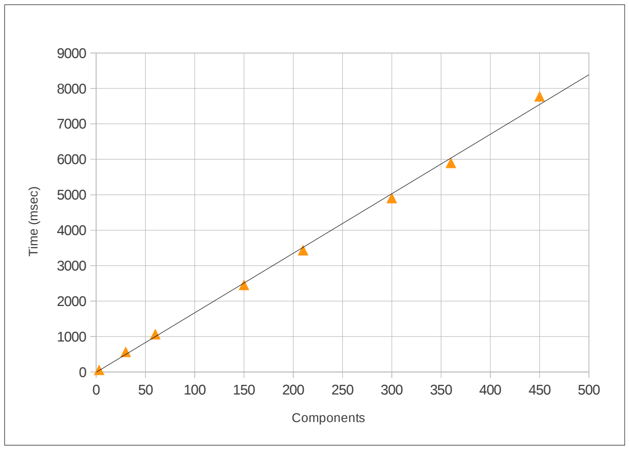

Figure 8. Time for generation of consistency checking constraints per number of components

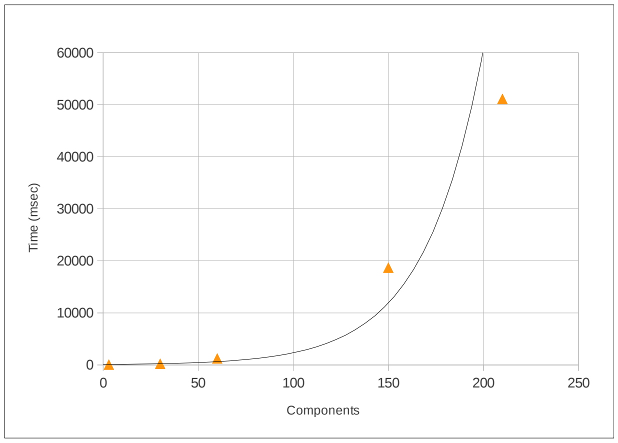

Figure 9. Time for checking of constraints per number of components

stakeholders.

Third, we consider component models as the central view on software architecture. Thus, our approach may be not applicable in the software projects where concepts of the models used for documenting the software architecture is significantly different from those of the component model shown in Figure 3. Nevertheless, the methodology presented in this paper can be adapted to other kinds of software architecture models with reasonable efforts. Our approach has not considered the representation of design patterns that are usually implied by ADDs. However, the extension of our approach to embrace design patterns is part of our future work.

VII. Related Work

A substantial amount of work has been done in the direction of documenting the AK using architectural decision modeling. Jansen and Bosch see software architecture as being composed of a set of design decisions [1] and propose a meta-model to capture decisions that consist of problems, solutions and attributes of the AK. Zimmermann et al. [4, 6] propose a more detailed meta-model for capturing ADDs, while Tyree and Akermann [2] propose a generic template. Some other ADD models and tools are summarized in [5]. Kruchten et al. [3] breaks down the decision capture in steps. Many of these approaches are focused on the visualization of the relationships between ADDs and their connection to alternatives, but not on the visualization of the software system, for which the ADDs are supposed to be the main input. Thus, a connection of the ADDs with the architecture design is absent. Our approach aims at bridging this gap using a (semi-)formal mapping between ADDs and component models.

Component models [13] are widely used to describe software systems. In most of component-based software systems, from COM/DCOM and CORBA to EJB, SOA and OSGi, reusable components and their relationships are essential architectural building blocks. Component-based Software Engineering (CBSE) has gained prominence in the industry because it brings architecture-centric thinking to the center of the software development process [7–9]. In our approach, we use an abstract component-and-connector model that is similar to the ones widely used in industry for describing and documenting software architecture.

Constraint languages are used in the literature to describe architecture constraints for component-based software [14] or constraints for primitives of architectural patterns [15]. Constraints for consistency checking are used extensively in many areas of software engineering [16–19]. In our case we use constraints to check the consistency between ADDs and component models.

Tang et al. [20] investigate four design decision models and four existing tools in order to analyze the level of support of architecture evolution. They conclude that these models and tools have limited support of architectural evolution and do not solve the problem of AK vaporization as a software system evolves. We claim that with our approach we support the architectural evolution and the maintenance of the design rationale.

A considerable amount of research has been conducted in relating requirements with software architecture. Liu et al. [21] propose a feature-oriented mapping and transformation from requirements to software architecture. We target mainly the ADDs that not only reflect the requirements but also include the solutions selected to resolve the requirements. Kaindl et al. [22] suggest that with the use of model-driven approaches we can map requirements to architectural design and Grunbacher et al. [23] introduce the mapping from requirements to intermediate models that are closer to software architecture. We assume that after the collection of requirements and before the design of the architecture the capturing of the ADDs takes place. The requirements belong to the problem space, while ADDs to the solution space. We claim that a mapping from ADDs to component models is more direct and natural because the ADDs contain the design details which can be reflected on a component-and-connector model. Apart from that, ADDs and software design change sometimes independently from the requirements and we want to focus on the software system evolution from the point of view of the ADDs.

VIII. Conclusion and Future Work

In this paper we presented an approach for bridging ADDs and designs and checking their consistencies based on constraints to support software architecture maintenance and evolution. In particular, we enable software architects and developers to explicitly define the mapping of the ADDs to elements, properties and annotations of a component model in a formal manner. These mappings are then leveraged using model-driven techniques for automatically creating initial component models and generating constraints used for checking the consistency between the ADDs and the component model. Our solution is feasible and sufficiently scalable and performs well enough for large sets of decisions and component model sizes.

One of our future endeavors is to extend the component model and the mapping of ADDs to the component model in order to enhance the automation of deriving the mapping as well as to support design patterns in our approach. We will also investigate the relation between reusable ADDs and component models. Furthermore, we plan to study how our approach can be implemented for different ways of documenting ADDs and software architectures by elaborating more industrial case studies and more complicated scenarios. Another concern is also to optimize the constraint-checking between ADDs and component models.

Acknowledgment

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

References

[1] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *The 5th Working IEEE/IFIP Conf. on Software Architecture*, pp. 109–120, IEEE Comp. Soc., 2005.

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

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

[4] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "Reusable architectural decision models for enterprise application development," in *Proc. of QoSA 2007*, pp. 15–32, Springer-Verlag, 2007.

[5] M. Shahin, P. Liang, and M. R. Khayyambashi, "Architectural design decision: Existing models and tools," in *IEEE/IFIP Conf. on Software Architecture/European Conf. on Software Architecture*, pp. 293–296, IEEE, 2009.

[6] 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 IEEE/IFIP Conf. on Software Architecture*, pp. 157–166, IEEE, 2008.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley, 2nd ed., 2002.

[8] W. Kozaczynski and G. Booch, "Guest Editors' Introduction: Component-Based Software Engineering," *IEEE Software*, vol. 15, no. 5, pp. 34–36, 1998.

[9] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

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

[11] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2nd ed., 2010.

[12] N. B. Harrison, P. Avgeriou, and U. Zdun, "Using Patterns to Capture Architectural Decisions," *IEEE Softw.*, vol. 24, pp. 38–45, July 2007.

[13] K.-K. Lau and Z. Wang, "Software Component Models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, 2007.

[14] C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse, "Component-based Specification of Software Architecture Constraints," in *14th Int'l ACM SIGSOFT Symposium on Component Based Software Engineering*, (Boulder, Colorado, USA), ACM Press, June 2011.

[15] U. Zdun and P. Avgeriou, "Modeling Architectural Patterns Using Architectural Primitives," in *20th ACM Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2005)*, (San Diego, CA, USA), pp. 133–146, ACM Press, October 2005.

[16] T. Sunetnanta and A. Finkelstein, "Automated Consistency Checking for Multiperspective Software Applications," in *Int'l. Conf. on Software Engineering Workshop on Advanced Separation of Concerns*, pp. 12–19, 2001.

[17] R. F. Paige, D. S. Kolovos, and F. A. Polack, "Refinement via Consistency Checking in MDA," *Electronic Notes in Theoretical Computer Science*, vol. 137, no. 2, pp. 151–161, 2005.

[18] B. Graaf and A. Van Deursen, "Model-Driven Consistency Checking of Behavioural Specifications," in *4th Int'l Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pp. 115–126, IEEE, 2007.

[19] A. Egyed, "Instant consistency checking for the UML," in *28th Int'l Conf. on Software Engineering*, pp. 381–390, ACM, 2006.

[20] A. Tang, J. Han, and R. Vasa, "Software Architecture Design Reasoning: A Case for Improved Methodology Support," *IEEE Software*, vol. 26, no. 2, pp. 43–49, 2009.

[21] D. Liu and H. Mei, "Mapping requirements to software architecture by feature-orientation," *Requirements Engineering*, vol. 25, no. 2, pp. 69–76, 2003.

[22] H. Kaindl and J. Falb, "Can We Transform Requirements into Architecture?," in *3rd Int'l Conf. on Software Engineering Advances*, pp. 91–96, IEEE Comp. Soc., 2008.

[23] P. Grunbacher, A. Egyed, and N. Medvidovic, "Reconciling software requirements and architectures with intermediate models," *Softw. Syst. Model.*, vol. 3, no. 3, pp. 235–253, 2003.