Modelling the Structure of Reusable Solutions for Architecture-based Quality Evaluation

Axel Busch, Yves Schneider, Anne Koziolek, Kiana Rostami
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: {busch, koziolek, rostami}@kit.edu,
yves.schneider@student.kit.edu

Jörg Kienzle
McGill University
Montreal, Canada
Email: joerg.kienzle@mcgill.ca

***Abstract*—When designing cloud applications many decisions must be made like the selection of the right set of software components. Often, there are several third-party implementations on the market from which software architects have the choice between several solutions that are functionally very similar. Even though they are comparable in functionality, the solutions differ in their quality attributes, and in their software architecture. This diversity hinders automated decision support in model-driven engineering approaches, since current state-of-the-art approaches for automated quality estimation often rely on similar architectures to compare several solutions. In this paper, we address this problem by contributing with a metamodel that unifies the architecture of several functional similar solutions, and describes the different solutions' architectural degrees of freedom. Such a model can be used later to extend the process of reuse from reusing libraries to reusing the corresponding models of these libraries with the lasting benefit of automated decision support at design-time that supports decisions when deploying applications into the cloud. Finally, we apply our approach on two intrusion detection systems.**

***Index Terms*—Software, Architecture, Structure, Model, Reuse, Solutions, Cloud, Concern, Decision support, Quality**

I. Introduction

Today, more and more business applications are deployed on cloud infrastructures. Such infrastructures come with many advantages such as reduced hardware costs, energy consumption, administration costs and of setting up a high quality software environment. However, deploying applications on cloud infrastructures does not solely solve the problem of developing high quality software as providing highly secure and highly performing systems. When a software architecture's (SA) design is vulnerable and does not scale well the security and performance quality, naturally, can not be of high quality.

One way to improve the quality of a software system is to reuse existing software solutions such as existing access control systems, intrusion detection systems (IDS), or payment systems as third-party subsystems. Such systems are widely used, well tested and offer functionality ready to be used. However, the different solutions differ in their quality attributes (QA), e.g., in their performance or in their security strength. Using SA models can help architects to understand the impact on the QAs of the architecture when making architectural design decisions upfront, i.e., at design time. To be aware of the impacts of design decisions on the SA's quality in turn helps to reduce the risk of failing to meet quality requirements.

9781-5090-1445-3/16$31.00 © 2016 IEEE (CloudSPD'16)

Several approaches exist that allow design time quality prediction using quality models, like the Palladio approach [1]. Such approaches rely on formal models used for predicting the SA quality at design time. However, even though such model-based approaches reduce the risk of having poor QAs after implementation, either constructing models for software architectures is very time-consuming and costly or if such models exist, the effort to reuse them is too high. Therefore, architects often shy the effort of applying such approaches and accept potential risks. Architects would benefit tremendously from having access to several models describing alternative solutions in decision making processes. This would make it possible to weed out solutions that would not achieve the desired QAs early on in the project.

In this paper, we present a metamodel for architectural design decisions with the focus on the reuse of the models of subsystems and on modelling their architectural degrees of freedom to be used in automated decision making processes. With the help of our metamodel, different subsystems realizing the same underlying concern can be modularized and packaged in a homogeneous structure. This structure can in turn be used to provide an architect tool-supported feedback in a decision making process when making trade-off decisions on several QAs. Due to the clear modelling structure, the SA models of such subsystems become better reusable in different contexts, and thus reduce the effort of model-driven software design. The result can later be used to make the right hardware and software component decisions when deploying the application on cloud infrastructures.

This paper is organized as follows: Section II introduces foundations about the Palladio and the PerOpteryx approach that we use as our basis and describes their limitations. Section III introduces terminology around our metamodel and the metamodel itself. Section IV demonstrates and discusses our methodology on two systems. Finally, Section V presents related work, while Section VI concludes the paper and discusses future work.

II. Foundations and their Limitations

Our approach is based on *Palladio* and *PerOpteryx*. In the following we describe both approaches and their limitations.

A. Architecture Model: Palladio Component Model

The Palladio Component Model (PCM, [1]) is a metamodel for component-based software architectures and also provides

a set of analysis tools for performance, reliability, and cost evaluation. Several parts from this introductory section are taken from [2].

The PCM is specifically designed for modelling component-based systems. Defining components in the PCM strictly follows the PCM component hierarchy. The component hierarchy is separated in the *provides*, *complete*, and *implementation* types. The *provides* type defines the provided interfaces comprised of the services a component offers to be used by other components or users. The *complete* type extends the *provides* type with required interfaces comprised of services a component requires to realise the provided services. The *implementation* type goes further and provides an abstract behavioral description on the implementation of the services of a component. Each type consequently focuses on the elements and the levels of detail needed for accomplishing a specific task. For instance when reusing a component the most interesting part of a component is the service provided by the component. Details on the implementation are hidden since they are not crucial for the initial task of reuse. We base our approach on this component hierarchy and adapt it to the idea of reusing subsystems.

A software architect composes component specifications from various component developers to form an application model. Components can either be modeled as basic components, i.e., a single component, or as composite structures. Composite structures can either be black box composite components or white box assemblies. The inner components of the black box composite components cannot be deployed on different hardware containers, while a separate deployment of white box assemblies is possible.

Palladio uses an abstract behavioral description, called service effect specification (SEFF), for the performance prediction of SAs. SEFFs model the abstract control flow through a component service in terms of internal actions (i.e., resource demands accessing the underlying hardware) and external calls (i.e., accessing connected components). Modelling each component behavior with separate SEFFs enables us to quickly exchange component specifications without the need to manually change system-wide behavior specifications (as required in e.g., UML sequence diagrams).

PCM strictly separates parametrized component performance models from the composition models and resource models. Thus, the PCM naturally supports many architectural degrees of freedom out-of-the-box. For example, it is easy to substitute a component with another component that provides the same interface. Similarly, it is easy to change the allocation of components to resources. This makes it possible to perform automated design space exploration. However, substituting a complex configuration of components is not supported yet.

Consider the Media Store ([1]) example in Fig. 1, which is realized using the Ecore-based PCM metamodel and visualized here in UML-like diagrams for quick comprehension. Media Store is a fully implemented file hosting application. Figure 1 shows an overview of the Media Store SA with an exemplary deployment configuration in a cloud environment. To use the Media Store's services, users must register and pay. After a successful payment the users can log in the system and list the existing files. They can then download the existing files or upload their own files. Before a user can download a media file, the raw audio file is tagged by a personal, individualized watermark, encoded by a selected codec and wrapped in a container format to be ready for download.

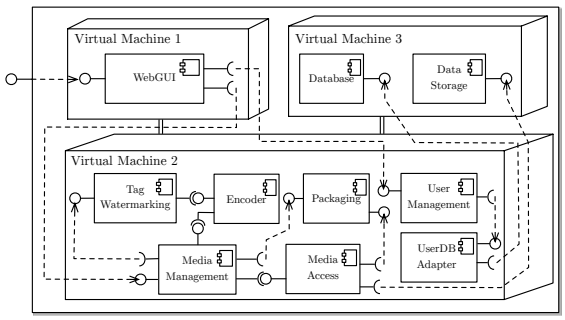

Fig. 1: Component and Deployment View of Media Store

The Media Store architecture model is comprised of several software components that run on three virtual machines. There are several components for user management, file handling and data storage. The software components are enriched by cost information, while the hardware nodes contain performance (processing rates) and cost (fixed and variable cost in an abstract cost unit) information. The particular function of each component is not important for the explanation. Such a model can then be transformed into analytical or simulation-based models for quality analyses.

B. Design Space Exploration: PerOpteryx

The PerOpteryx approach [2] explores the design space of a given SA and thus supports well-informed trade-off decisions for performance, reliability, and costs. Several parts of this introductory section are taken from [2].

For the exploration, PerOpteryx makes use of the so-called *degrees of freedom* of the SA that can either be predefined and derived automatically from the architecture model or be modeled manually by the architect. In our example (Figure 1), we have two types of predefined degrees of freedom: We can change the component allocation and the server configuration. For the allocation degree, we could have several virtual machines, each having different possible processing rates. As an example of a manually-modeled degree of freedom, let us consider that some of the architecture's components offer standard functionality for which other implementations, i.e., other components, are available. In this example, let us assume that there is another available component `FastEncoder` that can replace `Encoder`. Assuming that `FastEncoder` has lower resource demands, but is also more expensive than `Encoder`, the resulting architecture model would have a lower response time, but would result in higher costs.

The resulting degree of freedom instances (DoFI) span a design space which can be explored automatically. Together, they define a set of possible architecture models. The set of all possible architecture models corresponds to the set of all possible combinations of the design options.

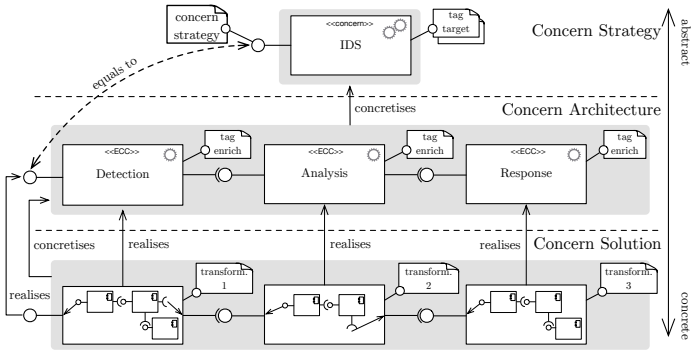

Fig. 2: Illustration of Model Elements

Using the quantitative quality evaluation provided by the PCM analysis tools, PerOpteryx can determine performance, reliability, and cost metrics for each possible architecture model.

Based on the DoFIs (as optimization variables) and the quality evaluation functions (as optimization objectives), PerOpteryx uses genetic algorithms and problem-specific heuristics to approximate the Pareto-front of optimal candidates. Details on the optimization are not required for the discussion in this paper, but can be found in [3], [2].

C. Limitations

Reusable subsystems such as the previously mentioned IDS are more complex in their architecture than a single component. They are comprised of many components with complex inner- and interrelations. Palladio and the metamodel of PerOpteryx currently do not support the exchanging of such complex subsystems during design exploration, due to a lack of a suitable metamodel to represent the architecture of such decisions and their architecture degrees of freedom. Thus, the impact of a decision such as the reuse of an IDS cannot be meaningfully studied with the current metamodel that PerOpteryx is based on. In this paper we address the limitations by a metamodel for modeling the architecture of complex subsystems such as the IDS.

III. Concern Structure Metamodel

A meta model illustration on a high level of abstraction is shown in Figure 2. The metamodel provides entities that allow a modeler to define architecture elements related to a specific *concern*, and reuse them in a context that is not fixed at design time. We have structured our metamodel into three parts, namely the concern definition, the solution definition, and the transformation description:

- The *concern definition* models the architecture of a concern, i.e., the components, their structure and relationships. In Figure 2 the concern definition is considered when defining the concern strategy and the concern architecture. Here, we define the features of a reusable subsystem (e.g. the IDS) and its meta architecture. The meta architecture represents the general components architecture a subsystem is comprised of (e.g., Analysis, Detection, Response functionality).
- The *solution definition* models the architecture of concrete solutions (e.g., from different vendors) that realize the concern architecture.
- The *transformation description* models the instructions on how to include the concern, and the concrete solutions in the target architecture model. Further, the transformation description contains a model that enriches the concern definition, and the solution definition with deployment constraints (e.g., Analysis must not be deployed together with Detection for security reasons).

The concern definition is a solution independent model and describes the meta structure of several independent solutions. The solution definition and the transformation description are specific for particular solutions.

A. Terminology

In [4], a concern is defined as "a domain with its own specialized knowledge space" and applied to low-level software design with class and sequence diagrams. We extend this definition and apply it to reusing concerns in component-based systems.

Definition III.1. Concern Strategy: The concern strategy defines one or several features in order to fulfill one or several functional or operationalized quality requirements (see [5]).

Illustrating Example: An IDS provides features to detect attacks, such as detecting denial of service attacks or bad user logins. Such features could be a subset of features of the strategy of such an IDS.

Definition III.2. (Architectural) Concern: An (architectural) concern is a matter of interest realizing a *concern strategy*. A concern's architecture is comprised of one or several concern elements that decompose the concern into elementary components. Further, the concern defines on architecture level how elementary components interact with each other to fulfill the concern strategy.

As defined previously, we decompose the concern into elementary components. We call such elementary components the *Elementary Concern Components* (ECC).

Definition III.3. Elementary Concern Component: An elementary concern component is a logical and structural self-contained entity. It fulfills a (sub-)feature of an architecture concern. Elementary concern components decompose an architecture concern into logical parts. It can require other elementary concern components (of the same concern).

Illustrating Example: The IDS is a matter of interest realizing the concern strategy, namely for instance to detect system attacks. Such a system can be decomposed to three elementary components (i.e., the ECCs), namely *Detection*, *Analysis*, and *Response*. Together, these three ECCs realize the functionality of an IDS.

The concern architecture ensures independence of the later usage context and an interchangeability of concern solution sets. We define such a concern solution as follows:

Definition III.4. Concern Solution: A concern solution is an instance of a concern realizing its concern strategy. The architecture of the concern solution follows the concern meta architecture, i.e., the architecture of the ECCs and their requirements to each other.

Illustrating Example: `OSSEC` (http://ossec.github.io/) is one solution for the IDS concern and is comprised of the entire ECCs of the IDS concern. Regarding the architecture of OSSEC, two components can be identified, namely `OSSECDecoder` and `OSSECAnalyzer`. Together, they build the ECC *Analyzer*. Components for the ECCs *Detection*, and *Response* can be identified in a similar manner. The Analyzer, though, cannot work for itself and requires raw data to detect attacks. This suggests that the *Analyzer* requires the ECC *Detection*.

A concern is designed by a concern expert who has deep knowledge of the domain of a concern.

Concern hierarchy

The general architectural structure of a concern follows the concern hierarchy. The architecture of a concern is in accordance with the previously mentioned definitions: the concern strategy, concern architecture and the concern solution (see Figure 2). On the concern strategy level, a concern expert would specify a concern and its related concern strategy. On this level, the concern remains the most abstract. The concern user can then concentrate entirely on the concern strategy, i.e., the services an architect may get when choosing to reuse the concern. On the concern strategy level the model does not contain any information about the underlying concern architecture. This is in accordance with the Information Hiding Principle [6], which dictates that all information that is not needed for the process of making architectural design decisions remains hidden. As a result, no realization dependencies can be created, and specific solutions can be interchanged transparently in the future. Reusing a concern and selecting a specific solution should be kept easy (for example, if the architect would have to choose from a large concern repository). The architect should concentrate on the benefits of selecting a certain concern, but should not come in touch with the details of the concern's architecture.

On the concern definition layer, the concern expert would define the architecture of the concern by making the abstract concern definition more concrete with ECCs and their interrelations to each other.

Finally, on the concern solution level, solution developers can then use the concern with its ECCs and relate them to their concrete components. Several solutions for the same concern are possible (solution instances). Such solution instances are then one degree of freedom for the concern and can be exchanged in decision making approaches (e.g., PerOpteryx).

In the following we describe and explain how the three levels are represented in our metamodel.

B. Concern Definition

Defining a new concern mainly takes place on the concern strategy and in the concern architecture layer. Each concern is associated with one particular set of ECCs. We provide six new elements for defining a concern.

The first element, the `ConcernRepository`, represents a container for all concerns and components. In a reuse process, a software architect looks for applicable concerns in the concern repository. We have chosen to extend the PCM repository to earn its advantages, such as quality prediction and (eclipse) tool support.

C is the set of all concerns, while c is one concern of the set of concerns in the repository. Each `Concern` contains `ElementaryConcernComponents` *ecc*. Each ECC belongs to one particular concern: Further, a concern has several `ConcernStrategy` entities cs, while each concern strategy is associates with one particular concern.

$$elemConcComp : ECC \rightarrow C, ecc \mapsto c \quad (1)$$

$$concStrat : CS \rightarrow C, cs \mapsto c, \quad (2)$$

while $ecc \in ECC$ and $cs \in CS$. In this paper, we simplify the concern strategy by using provided interfaces following the provided interfaces of components. Thus, the concern strategy inherits from `PCM::Operation Interface`. The operation interface uniquely determines features provided by the concern. Our model is comprised of two `Annotation` types (see Figure 2) that are used like tags in order to enrich the model with additional characteristics. The first type, namely $tag^{target} \in Tag^{target}$ belongs to concerns and declares where a concern applies in the target architecture. Such a tag is associated with one particular concern. This tag type advises where to assemble a certain ECC in the target architecture and uses the attribute `Annotations::maxAmount` indicating *how often* a particular ECC can be assembled. Where and how often an element is assembled impacts the quality attributes of the target SA.

$$ttarget : Tag^{target} \rightarrow C, tag^{target} \mapsto c \quad (3)$$

The second type of `Annotations` belongs to ECCs, namely the tag^{enrich}. Tag^{enrich} represents the set of this type of annotations. tag^{enrich} can be used to determine how a concern element has to be included in the target architecture. More concrete, it defines deployment constraints of ECCs.

$$tenrich : Tag^{enrich} \rightarrow EC, tag^{enrich} \mapsto ec \quad (4)$$

$$tagAmnt : Tag^{enrich} \rightarrow M, tag^{enrich} \mapsto maxAmount \quad (5)$$

while $M := \mathbb{N}^{+} \cup \{\infty\}$. Applying such additional components in the target architecture has influences on the SA quality: Each time the Detection ECC is applied the performance overhead increases. Thus, the number of assembled assemblies is another degree of freedom that is used by automated design space exploration approaches.

Finally, the ECC defines `DeploymentConstraints`. DeploymentConstraint specifies whether ECCs can be deployed with no constraint, together on the same hardware container, separated (i.e., ECCs must be deployed to different hardware

containers), or isolated (i.e., no other components on the same machine allowed).

$$deplConst_{pair} : Tags \rightarrow mode^{pair} \quad (6)$$

$$deplConst_{enrich} : Tags^{enrich} \rightarrow mode^{enrich} \quad (7)$$

while $Tags := (Tag^{enrich} \cup Tag^{target}) \times Tag^{target}$, $mode^{pair} := \{$*indifferent, together, separated*$\}$ and $mode^{enrich} := \{$*indifferent, isolated*$\}$. ECCs can require others:

$$requires : EC \rightarrow 2^{EC} \quad (8)$$

Illustrating Example: Let us consider the previously mentioned IDS concern. A security expert could define an IDS concern comprised of the previously mentioned ECCs, namely *Detection*, *Analysis*, and *Response*. Further, he defines a concern strategy "DetectIntrusion". The ECC *Detection* is annotated with the tag^{enrich} @*Observer* indicating that the ECC *Detection* will be assembled with all the components that are potentially at risk. An architect could annotate the components that need to be observed with the tag^{target} @*Observee*. Let us assume the WebGUI component of Media Store could be such a potential component fraught with risk. The software architect could then annotate the WebGUI component with @*Observee*. Since a software architect could porentially assemble such a *Detection* concern component with any number of components, we would set *maxAmount* of @Observee to ∞.

C. Solution Definition

Several solutions can be associated with a concern. The definition of a solution for a particular concern requires the definition of all the corresponding ECCs. The solution definition mainly takes place on the concern solution layer and is realized by the element `ConcernSolution`. Concern solution inherits from `PCM::RepositoryComponent` and contains the element `ConcernTransformation` that belongs to the transformation description.

The architecture model of a solution can be defined by using one basic component or several associated basic components. For the definition of several components a software architect can either use a black box *composite component* `PCM::CompositeComponent` or its white box correspondence, namely `PCM::Subsystem`. The white box flavor benefits in detached distribution of inner components, what in turn can be used by design space exploration approaches to find architecture candidates of higher quality.

D. Transformation Description

The transformation description mainly takes place on the concern solution layer. The `transformation` entity determines the strategy on how to include a particular ECC solution. Further, it defines where the strategy is applied in the target SA and if the strategy can be applied multiple times. We have modeled two types of strategies, namely an adapter strategy and the extension strategy. The adapter strategy adds a component between two others, while the extension strategy adds an interface to an existing component and modifies the caller sequence. The adapter strategy can be configured to adapt the actual call either *before*, *after*, or *around*.

$$tstrategy : Tag^{target} \times Tag^{enrich} \rightarrow strategy \times position \quad (9)$$

$$tmult : Tag^{target} \times Tag^{enrich} \rightarrow multiple \quad (10)$$

while $strategy := \{adapter, extension\}$, $position := \{before, after, around\}$ and $multiple := \{true, false\}$.
Illustrating Example: We define the transformation for the ECC *Detection* with the adapter strategy *before* to ensure the monitoring right before the actual execution of the command. Further, we set multiple to *true*, since a sensor can be included multiple times.

IV. Evaluation

In this section, we include the IDS concern in the Media Store as the target system that was previously mentioned in our running example to demonstrate the applicability and benefits our approach. We first model the IDS concern (as described in the previous Section), define two solutions to the IDS concern architecture, and finally demonstrate how the concern can be included with the target SA model.

A. Concern Definition: IDS

Our IDS concern is comprised of three ECCs, namely `Analyzer`, `Detection`, and `Response`. Table I shows the architecture of the IDS concern with its ECCs.

ECC	Analysis	Detection	Response
tag^{enrich}	@Analyzer	@Observer	@Reaction
$required(ECC)$	{Detection}	{}	{Analysis}
tag^{target}	@Observee, ∞		
Deployment	@Observee, @Observer, TOGETHER		
Constraint	@Observee, @Analyzer, SEPARATED		

TABLE I: IDS concern with elementary concern components

The *Analysis* ECC requires *Detection*, while *Response* requires *Analysis*. We define the tag^{target} annotation @Observee that is later annotated to critical components to mark them worth to be observed. This annotation can potentially be annotated to any number of components in the target architecture.

B. Concern solution

As examples for the concern solution, we modeled two host-based IDS, namely *OSSEC* and *Samhain* (http://www.la-samhna.de/samhain/). Both implement a client-server based architecture. Table II shows the application of our two example systems. These two concern solutions can then be included in the target SA by annotating the target components with the tag^{target} annotation @Observee. Besides, a software architect needs to regard the transformation instruction and deployment constraints.

C. Concern Application

The previous defined models define the architectural degrees of freedom for including the IDS concern in the Media Store architecture. Our model includes two deployment constraints: First, the @Observer and @Observee annotated components have to be deployed together, while @Observee and @Analyzer must not allowed to be deployed together. Further, according to our model, we can potentially observe all components with

	OSSEC	Samhain
Analyzer Components	OSSECDecoder, OSSECAnalyzer	YuleLogCollector, Database
Detection Components	OSSECCollect	SamhainIntegrityCheck, LogChecker
Response Components	OSSECAlert	BeltainConsole, ActiveResponse
Transform.	(@Observee, @Observer) → (Adapter, before) (@Observee, @Observer) → true	(@Observee, @Observer) → (Adapter, before) (@Observee, @Observer) → true

TABLE II: IDS concern solution using OSSEC and Samhain

the ECC *Detection* by annotating them with the tag^{target} annotation @Observee. This results in a security / performance trade-off, though. The more often *Detection* is assembled, the more often the SEFFs of its inner components are called, data is intercepted and sent to the *Analyzer* to be processed.

Let us assume, we annotate the components $\{WebGUI, Encoder, Packaging, UserDBAdapter\}$ with @Observee. According to the transformation instruction, adapters are used to include @Observer annotated ECCs to the @Observee annotated components of the target SA. With the combination of components, ECCs, the solutions of elementary candidates, namely OSSEC and Samhain, and virtual machines, the degree of freedom space spans many architecture candidates that can quickly become millions of candidates. Such a huge number of architecture candidates can neither be modeled, nor evaluated in quality prediction approaches by hand. To enable an automated evaluation of the architecture candidates we plan to combine our metamodel that describes the architecture degrees of freedom of complex design decisions such as IDS with PCM and PerOpteryx. This extension then allows software architects to make trade-off decisions between performance and other quality attributes for complex design decisions.

V. Related Work

In concern-oriented reuse (CORE) [4], software development is structured around modules called *concerns* that provide a variety of reusable solutions for recurring software development issues. CORE concerns form modular units of reuse that encapsulate a set of software development artifacts, i.e., models and code, describing all properties of a domain of interest during software development in a versatile, generic way behind 3 interfaces: variation, customization and usage interface [7]. While the ideas of CORE are generic, i.e., they can be applied to any modelling language, CORE has so far only be demonstrated on class and sequence diagrams. The approach for modularizing architectural concerns presented in this paper is compatible with CORE. The tags are equivalent to the CORE customization interface, and the interface defined in the concern strategy corresponds to the usage interface of CORE concerns. Our proposed approach has no explicit variation interface that regroups the different existing architectural solutions for a given strategy, but it would make sense to use the CORE variation interface for that purpose. On the other hand, the goal models used in CORE to express the quality impacts of different solutions lack precision when applied to quality goals.

[8] define AO-ADL, an aspect-oriented architectural description language, that can be used to capture architectural patterns using parameterizable component diagrams similarly to what the component models of our concerns. They show how to specify reusable architectural patterns that can improve non-functional properties of the system, in particular usability, and integrate them into applications using aspect-oriented modelling techniques. However, AO-ADL addresses only on structural properties of software architectures, and cannot be used to specify behavioural approximations to optimize performance.

VI. Conclusions

In this paper, we presented an approach for modeling the architecture of third-party subsystems (e.g., intrusion detection systems or access control systems) with the purpose of improving software quality at design time. Our metamodel focuses on the reuse of such solutions in existing software architectures (SA). Architects who design cloud systems can use our approach to assess the effect of different security solutions on other quality attributes, such as performance and reliability. We have introduced the terminology around our approach and presented a detailed description of the metamodel elements, how they can be used to model the architecture degrees of freedom of such architectures. We have shown the applicability of the approach on one concern, the IDS, with two existing concern solutions, OSSEC and Samhain. The IDS concern was then reused in an existing SA model, namely the Media Store system, and we explained how such a model could help to make trade-off decisions with respect to performance.

We plan to include our approach into PerOpteryx to allow automated decision support for complex design decisions, such as the IDS. To generate SA candidates automatically, we are currently developing an approach that uses our structure model as input and results in a set of architectural degrees of freedom. They can then be used to automatically optimize SAs in PerOpteryx to support the architect in decision making processes. The resulting candidates can then be used by software architects to decide which hardware configuration of a cloud service provider should be configured and which particular software components should be selected to achieve the expected software quality.

Acknowledgments. We thank Dominik Werle for his productive input and discussions.

References

[1] R. Reussner *et al.*, *Modeling and Simulating Software Architectures - The Palladio Approach*. MIT Press, 2016, to appear.

[2] A. Koziolek, *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. KIT SP, Karlsruhe, 2013.

[3] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: autom. application of tactics in mult.-obj. SA optimization." QoSA-ISARCS, 2011.

[4] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *MODELS 2013*. Springer, 2013, pp. 604–621.

[5] M. Glinz, "On non-functional requirements," in *15th IEEE International Requirements Engineering Conference (RE 2007)*, 2007.

[6] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, 1972.

[7] J. Kienzle, G. Mussbacher, O. Alam *et al.*, "VCU: The Three Dimensions of Reuse," in *ICSR 2016*, ser. LNCS. Springer, 2016.

[8] M. Pinto and L. Fuentes, "Modeling quality attributes with aspect-oriented architectural templates," *J. UCS*, vol. 17, no. 5, pp. 639–669, 2011. [Online]. Available: http://dx.doi.org/10.3217/jucs-017-05-0639