Automatic Evaluation of Complex Design Decisions in Component-based Software Architectures

Max Scheerer
FZI Research Center for Information Technology
Germany
scheerer@fzi.de

Axel Busch
Karlsruhe Institute of Technology
Germany
busch@kit.edu

Anne Koziolek
Karlsruhe Institute of Technology
Germany
koziolek@kit.edu

ABSTRACT

The quality of modern industrial plants depends on the quality of the hardware used, as well as software. While the impact on quality is comparably well understood by making decisions about the choice of hardware components, this is less true for the decisions on software components. The quality of the resulting software system is strongly influenced by its software architecture. Especially in early project phases a software architect has to make many design decisions. Each design decision highly influences the software architecture and thus, the resulting software quality. However, the impact on the resulting quality of architecture design decisions is hard to estimate in advance. For instance, a software architect could decide to deploy software components on a dedicated server in order to improve the system performance. However, such a decision may increase the network overhead as side-effect. Model-driven approaches have been shown as promising techniques enabling design-time quality prediction for different quality attributes such as performance or reliability. However, such approaches are limited in their automated decision support to simple design decisions like the exchange of one single component. In this paper, we present an approach that automatically evaluates complex design decisions in software architecture models. Such design decisions require the reuse of subsystems with many involved components coming with inhomogeneous architectures. We evaluate our approach using a real-world example system demonstrating the benefits of our approach.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; *Model-driven software engineering*; *Extra-functional properties*; Reusability; • **Applied computing** → Multi-criterion optimization and decision-making;

KEYWORDS

Reuse, model-driven engineering, component-based, software architecture, decision support, software quality

MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria

ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127059

ACM Reference Format:
Max Scheerer, Axel Busch, and Anne Koziolek. 2017. Automatic Evaluation of Complex Design Decisions in Component-based Software Architectures. In *Proceedings of MEMOCODE '17, Vienna, Austria, September 29-October 2, 2017,* 11 pages.
https://doi.org/10.1145/3127041.3127059

1 INTRODUCTION

In recent years, the share of software in modern industrial plants has continued to grow. One reason for this is that modern cyber-physical-systems are becoming increasingly complex, have more responsibilities, and at the same time have to be more flexibly configurable. Especially software has to provide many features and, at the same time, is subject to high quality and cost constraints. Thus, designing software is challenging since these aspects tend to be antithetic to each other. To cope with such high demands on functionality and quality, in modern software development processes the reuse of software components has been established as common practice. Most functionality and features are not developed from scratch, but are served by reusing libraries or COTS[1]-components.

The reuse of components and libraries goes far beyond the reuse of the pure functionality. Besides the functionality a software architect reuses knowledge about the architecture and the internal- and external interaction of libraries. Reusing software artifacts helps to decrease the development time and prevents recurring errors. Developing software more cost efficient and with higher quality in turn increases the overall quality of systems with a crucial interplay between hardware and software such as a cyber-physical-system.

However, when reusing common solutions, often, there are several different alternatives regarding a similar set of functional requirements. For instance, let us consider the category of Intrusion Detection Systems (IDS) like AppSensor[2] and OSSEC[3] which are able to observe system attacks. The decision to choose AppSensor instead of OSSEC depends on both the functional and the quality attributes of the solutions. While the functional attributes might be evident from the documentation, quality attributes usually are not. Since each feature typically influences several quality attributes of a system, a software architect would be interested in the implications of decisions upfront in order to make better design decisions. However, without any upfront design evaluating several decisions by hand might be time-consuming and costly. Using software models

[1] Commercial Off The Shelf: A system or component which has been bought from an external supplier.
[2] https://www.owasp.org/index.php/OWASP_AppSensor_Project
[3] https://ossec.github.io/

can ease such a decision approach, since solutions are not to be purchased, installed and finally evaluated. By the help of model-based quality assessment approaches, such as the Palladio approach [1], a software architect can use the Palladio Component Model (PCM) to predict quality attributes, such as the response time, of software architectures (SA) at design time.

Nevertheless, evaluating several architecture decisions is not trivial even with such model-driven techniques. Each design decision comes with several degrees of freedom, such as the selection of a concrete solution (e.g., AppSensor vs. OSSEC), the deployment configuration on hardware (e.g., 2-tier architecture vs. 3-tier architecture), or the hardware configuration itself (e.g., CPU clock speed, disk transfer rate, ...). Such architecture degrees of freedom shift the number of architecture candidates to be evaluated to many thousand candidates – each coming with different quality attributes. Because of this high number of candidates, a manual evaluation is often infeasible.

Several approaches support an automatic evaluation of such a huge design space, such as PerOpteryx [2], ArcheE [3], and ArcheOpteryx [4]. PerOpteryx results in Pareto-optimal solutions regarding the quality attributes of a SA.

The contribution of this paper is to extend the PerOpteryx approach to include the analysis of several different solutions of the same decision category such as the category of Intrusion Detection Systems. Our extension gives a software architect tool-supported feedback about the quality effects of such complex design decisions. Such complex design decisions are typically comprised of many components and do not have coherent interfaces to interact with the main system. Therefore, we developed a model integration mechanism which can be described with triple graph grammars to integrate automatically a subsystem into an existing PCM SA model. In order to compare how different subsystems affect the overall software quality, we integrated our approach into PerOpteryx. As a result, we provide a fully-automated approach enabling a software architect to evaluate different design decisions more efficiently.

The paper is organized as follows: Section 2 motivates the problem domain this paper is concerned with. Section 3 introduces foundations about Palladio and PerOpteryx. Moreover, a meta model for architectural design decisions with the focus on the reuse of subsystems is introduced as well as fundamental concepts of model synchronization using triple graph grammars (TGG). Section 4 formalizes the general problem based on TGGs. Section 5 describes our approach in more detail. Section 6 validates our approach and demonstrates the benefits in an industry-related example. Finally, Section 7 discusses related work before Section 8 concludes the paper.

2 MOTIVATING SCENARIO

In order to motivate our problem domain, let us consider the example SA depicted in Fig. 1 which is a remote diagnostic solutions (RDS) system of industrial devices. Industrial devices periodically contact the system in order to upload diagnostic information about the internal status. A user of the system can track the device on a website. Moreover, different reports or analysis results can be requested. The illustrated SA originated from an industrial case study [5]. For the sake of simplification, we reduced the size and complexity of the SA to four software components namely `RDSConnectionPoint`, `DeviceDataProcessing`, `DataAccess` and `Database`. They are distributed and deployed on three hardware nodes. The software components can contain cost annotations (not depicted here), while the hardware nodes contain annotations for performance (processing rates) and cost (fixed and variable cost in an abstract cost unit).

Let us consider that a new requirement has to be implemented. The requirement necessitates an IDS in order to prevent potential attacker scenarios. Let us consider AppSensor and OSSEC as two concrete solutions of an IDS. Both can be integrated in a component-based SA in order to improve the overall security. We assume the IDS to be inserted between the two components `DataAccess` and `Database` to observe the data-flow and to detect suspicious user behavior. In such a scenario the software architect might have to make a decision between AppSensor and OSSEC. Most solutions differ in their implementation, their architecture and therefore in their impact on the software quality. The typical main objective of a decision making process is to find the solution that satisfies the non-functional (quality) requirements best. In our case, each concrete concern solution (e.g., OSSEC and AppSensor) that realizes a given concern (IDS) might improve several quality attributes but might also deteriorate others. Using an IDS may improve the overall security while at the same time the performance is decreased because of the overhead to analyze the data flow. For this reason, the software architect may be interested upfront in the effects this decision would have on performance and costs to achieve a gain in security.

In order to compare both concern solutions by the Palladio prediction techniques, the software architect has to integrate each model by hand. Depending on the size and complexity of the meta model such an integration process is very time consuming and potentially error prone. In addition, a software architect has to integrate each concern solution in the PCM model manually in order to apply PerOpteryx on the extended PCM model. In terms of AppSensor and OSSEC, two different PCM models need to be created in which one of them is extended by AppSensor and the other by OSSEC. Finally, PerOpteryx can be applied on each model in order to evaluate the optimal candidates. As a consequence, the result of the PCM model extended by AppSensor needs to be compared to the result of the same PCM model extended by OSSEC in order to identify which concern solution has the best impact on the software quality. The results have to be merged and visualized in order to pick the best candidate. However, this is an inconvenient task. The merged results contain candidates which are actually dominated by other candidates. This results from the two separated runs with PerOpteryx which considers only one concern solution. Therefore, the process of picking the best candidate is more difficult.

A possible solution to these problems is to realize a model integration approach which automates the emerging modelling effort. An automated approach for integrating models would input a concern solution and an existing PCM model. The output is again the PCM model extended by the concern solution. Such a model integration approach can then again be reused by PerOpteryx. More precisely, PerOpteryx could be extended by considering concerns and their corresponding concern solutions as degree of freedom. As a result, each produced candidate as well considers AppSensor or

OSSEC as potential concern solution. Thus, no separated analysis have to be done with PerOpteryx.

Extending PerOpteryx by such a model integration approach provides a software architect a powerful tool in order to select the concern solution which fulfills quality requirements best and reduces the manual modelling effort.

3 BACKGROUND

3.1 Architecture Model: Palladio Component Model

The Palladio Component Model ([1]) is a meta model for component-based SAs and also provides a set of analysis tools for performance, reliability, and cost evaluation.

The PCM is specifically designed for component-based systems and strictly separates parametrized component performance models from the composition models and resource models. Thus, the PCM naturally supports many architectural degrees of freedom (e.g., substituting components, changing component allocation, etc.) for enabling automated design space exploration.

A PCM model can be decomposed into the view-types repository, assembly, allocation and usage model view-type. Each view-type is concerned with a specific view on the modelled SA.

The repository view-type gathers all components represented by the meta class `PCM:RepositoryComponent` and interfaces represented by the meta class `PCM:Interface`. A component can require or provide a given interface if there is a role `PCM:RequiredRole` or `PCM:ProvidedRole` which references the interface. If a component provides a specific interface it has to describe the services contained in the interface. Therefore, the component developers provide an abstract behavioral description called service effect specification (SEFF). 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). Modeling 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). For performance annotations, component developers can use the extended resource-demanding service effect specifications (RDSEFFs). Using RDSEFFs, developers specify resource demands for their components (e.g., in terms of CPU instructions to be executed).

Further, the assembly view-type exposes which components are actually instantiated and connected to other components. The meta class `PCM:AssemblyContext` instantiates a given component. Moreover, there are several connectors represented by `PCM:Connector`. The `PCM:AssemblyConnector` connects two assembly contexts which instantiated components contain a required and provided role that reference the same interface. There are several types of connectors. For this paper it is sufficient to know the assembly connector.

The allocation view type describes on which resource container (e.g. server) a specific instantiated component is allocated. Therefore, it contains a set of `PCM:AllocationContexts`.

Finally, the usage model view-type specifies the interaction of one or more users with the overall system (e.g. input data, amount of interacting users).

Consider again the minimal PCM model example in Fig. 1, which is realized using the Ecore-based PCM meta model and visualized here in UML-like diagrams for quick comprehension. Fig. 1 shows

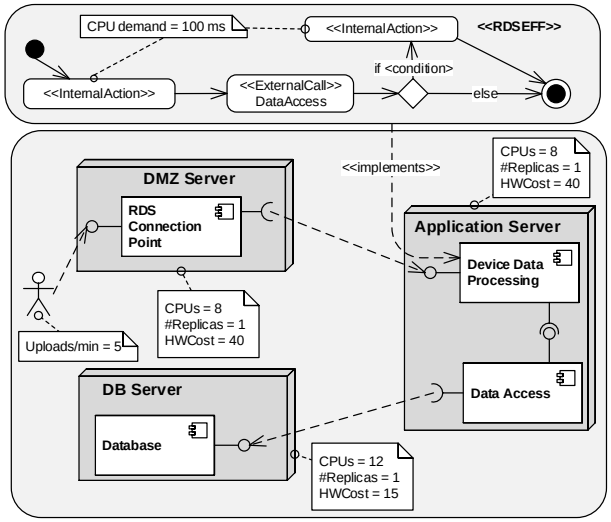


Figure 1: Example PCM Model: Remote Diagnostic System

the instantiated and connected components `RDSConnectionPoint`, `DeviceDataProcessing`, `DataAccess` and `Database`. Each assembly context is assigned to a specific server which indicates that the assembly context is allocated to. An example RDSEFF is shown in Fig. 1. At the top, a RDSEFF for one of the services `DeviceDataProcessing` provide is presented.

3.2 Design Space Exploration: PerOpteryx

We apply our methodology based on PerOpteryx [2], but the concepts are not limited to this approach. The PerOpteryx approach explores the huge set of SA configurations in which each configuration is a specific combination of all possible design decisions. Thus, PerOpteryx supports making well-informed trade-off decisions for performance, reliability, and costs.

For the exploration, PerOpteryx makes use of 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), there are two types of predefined degrees of freedom: The component allocation and the server configuration which can be changed. For the allocation degree, there are several servers, 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 there is a fifth available component `QuickDeviceDataProcessing` that can replace `DeviceDataProcessing`. Assuming that `QuickDeviceDataProcessing` has less resource demand but is also more expensive than `DeviceDataProcessing`, the resulting architecture model has a lower response time but higher costs.

The degrees of freedom span a design space and can be explored automatically. Together, they define a set of possible architecture models. Each of these possible architecture models is defined by choosing one design option for each DoFI. We call such a possible architecture model a *candidate model*. The set of all possible candidate models corresponds to the set of all possible combinations of the design options. We call this set of possible architecture models the *design space*.

Using the quantitative quality evaluation provided by the PCM analysis tools, PerOpteryx can determine performance, reliability, and cost metrics for each candidate model. In general, to quantify a quality attribute q, we choose a quality metric $m(q)$. The quality evaluation Φ_q for a quality attribute q can be expressed as a *quality evaluation function* from the set of valid PCM instances M to the set of possible values of the quality metric $m(q)$, denoted $\mathcal{V}_{m(q)}$: $\Phi_q : M \rightarrow \mathcal{V}_{m(q)}$. In addition to the evaluation functions, PerOpteryx requires a specification whether a quality is to be maximized or minimized.

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 [6, 7].

In its previous version described in this section, PerOpteryx does not support the analysis of the effect of decisions that require more complex modifications on the architecture model. Thus, the effects on performance and costs due to the decision of including a IDS system cannot be meaningfully studied with the previous version of PerOpteryx.

3.3 Concern Structure Meta Model

For the formal description of our entities of reuse (i.e., concerns), we use the meta model from one of our previous works (see [8]) namely the *Concern Structure Meta Model*. The concern structure meta model is comprised of three parts, the concern definition, solution definition and transformation description.

The concern definition part consist of a `ConcernRepository` that stores all (pre-)defined `Concerns`. Such a concern is an abstract entity that can be decomposed into its basic elements namely the *Elementary Concern Components* (ECC). ECCs can require other ECCs. Consider an IDS as concern. An IDS concern consists of the ECCs *Detection*, *Analysis* and *Response*. The *Detection* ECC inputs the data to observe. The actual analysis task is performed by the *Analysis* ECC. Therefore, *Detection* requires *Analysis*. Finally, *Analysis* requires *Response* in order to respond adequately when an attack is detected. ECCs represent the fundamental architecture of a concern.

The solution definition part structures the concrete architecture of the realization of a concern solution (a concern solution may be a reference implementation of a concept, such as AppSensor or OSSEC of the concept IDS). Each concern solution defines a repository model which enriches the fundamental ECC structure by concrete components.

Finally, the transformation description defines how a given concern or concern solution can be integrated into the target SA. The `TransformationRepository` stores all defined `Transformations`. A transformation has two subclasses, namely the `AdapterTransformation` and `DecoratorTransformation`. With the `AdapterTransformation` a concern solution is integrated into the target SA by using an adapter component as connection point. The `DecoratorTransformation` integrates a concern solution directly into the target SA.

Components annotated with tags, identify the integration points in the target SA where a concern solution shall be integrated. Technically, we used EMF profiles ([9]) in order to annotate repository components with tags. Given the annotated components and transformation descriptions, the concern integration engine can determine how a concern solution has to be integrated in the target SA.

3.4 Model Synchronization Based on Triple Graph Grammars

In this Section, we introduce the concepts of TGGs [10] in the notation of [11]. After that, we describe the model synchronization approach of [11] based on TGGs. Model elements of two models, that are semantically related, can be described with TGGs. The model synchronization approach considers integrated models consisting of a source and target model which share a correspondence structure. For instance, consider a repository and assembly model. For each component in the repository which is supposed to be instantiated in the system, there is an assembly context in the assembly model. An assembly context indicates that a given component is instantiated. Therefore, components and assembly contexts have a semantic relation. For instance, when a new component is added to the repository, there need to exist a corresponding assembly context in order to make use of the component in the system.

We describe the source and target models as well as the correspondence structure by triple graph grammars. A triple graph grammar comprises a source graph G^S, a target graph G^T, and a correspondence graph G^C. Recall that a graph $G := (V, E, s, t)$ consists of a set of vertices V and a set of edges E. The function $s, t : E \rightarrow V$ inputs an edge and returns the corresponding source or target vertex. Further, there are two graph morphisms $s_G : G^C \rightarrow G^S$ and $t_G : G^C \rightarrow G^T$. According to [10] a graph morphism of two graphs $G := (V, E, s, t)$ and $G' := (V', E', s', t')$ is defined as a pair of functions $h := (h_V, h_E)$ with $h_V : V \rightarrow V'$ and $h_E : E \rightarrow E'$ from G to G' so that $h : G \rightarrow G'$ if:

$$\forall e \in E : h_V(s(e)) = s'(h_E(e)) \wedge h_V(t(e)) = t'(h_E(e))$$

Based on the graph morphisms s_G and t_G the correspondence $r : G^S \leftrightarrow G^T$ relates elements in both directions $G^S \rightarrow G^T$ and $G^T \rightarrow G^S$.

Further, a triple graph G is typed. As a result, for each triple graph there is a type triple graph TG. Moreover, there is a graph morphism $type_G : G \rightarrow TG$ which means that each element of G can be typed by TG. Given a type triple graph $TG := (TG^S \leftarrow TG^C \rightarrow TG^T)$, $VL(TG)$ as well as $VL(TG^S)$ and $VL(TG^T)$ denote the set of graphs which can be derived from TG, TG^S and TG^T. In other words, in the context of models and meta models, a type triple graph TG represents a meta model while $VL(TG)$ represents the set of all model instances (or graphs) derived from TG, TG^S or TG^T.

In order to synchronize models Hermann et al. [11] defined forward and backward operations based on graph modifications. For our approach, we consider operations which forward propagate a given model change. Therefore, we concentrate on the forward operation which we define in the following as f_{fw}. A forward propagation operation can be applied based on a model change or a graph modification $\delta : G \to G'$. More precisely, $\delta : G \overset{i_1}{\leftarrow} I \overset{i_2}{\rightarrow} G'$ is given by two graph morphisms i_1 and i_2, while I contains the elements which are preserved during the model change. With the graph morphism $i_1 : I \to G$ the elements in G that are supposed to be deleted can be derived. Equally, the elements that are supposed to be added can be derived with $i_2 : I \to G'$. Based on a given model change or modification δ, the forward propagation operation can be described as follows:

$$
\begin{aligned}
f_{fw} &: (R \otimes \Delta_S) \to (R \times \Delta_T) \\
\Delta_S &:= \{\delta_S : G^S \to G'^S | G^S, G'^S \in VL(TG^S)\} \\
\Delta_T &:= \{\delta_T : G^T \to G'^T | G^T, G'^T \in VL(TG^T)\} \\
(R \otimes \Delta_S) &:= \{(r \times \delta_S) \in (R \times \Delta_S) | r : G^S \longleftrightarrow G^T, \\
&\qquad \delta_S : G^S \to G'^S\}, \delta_S \text{ and } r \text{ coincide on } G^S
\end{aligned}
$$

while R is the set of correspondence relations, Δ_S the set of graph modifications of the source graph G^S and Δ_T the set of graph modifications of the target graph G^T. More precisely, a forward propagation operation inputs a pair consisting of a specific correspondence relation r_1 and graph modification δ_S and outputs a correspondence relation r_2 and graph modification δ_T which are needed in order to obtain and synchronize the target model G'^T.

$$
\begin{array}{ccc}
G^S & \overset{r_1}{\longleftrightarrow} & G^T \\
\delta_S \downarrow & \searrow f_{fw} & \delta_T \downarrow \\
G'^S & \overset{r_2}{\longleftrightarrow} & G'^T
\end{array}
$$

4 FORMALIZATION THE MODEL CHANGE PROPAGATION

In this Section, we describe the general model change propagation problem to integrate a concern solution in a PCM model. More precisely, Palladio consists of several models describing the SA. In order to integrate a concern solution, we define a model that serves as entry model for the integration process. There are two reasons why a model M_2 is affected by a change of model M_1. A model change of M_1 could affect model M_2 because M_2 is inconsistent by now. For instance, each assembly context needs a corresponding allocation context. When an arbitrary assembly context is removed, the corresponding allocation context needs to be removed equally. Moreover, a model change of M_1 could affect model M_2 in order to make use of a concern solution at all. For instance, when the components of a given concern solution are added to the repository no other models are affected. The repository serves as central location which only stores different components. The components which are actually used in the system have to specified in another model. Therefore, to make use of the concern solution other models need to be modified.

The model integration's main objective is to automatically extend a given PCM model by a second (partial) model (i.e., a concern solution) according to a given rule set. In terms of Palladio, extending an existing PCM model by a concrete concern solution is not trivial because a change of a certain view-type or model which represents the view-type might cause a chain of model modifications. First, the software architect defines the entry point of the integration process. Changes of view-type VT_i could cause modifications on other view-types $VT_{j \neq i}$. More formally, let us define M_{VT} as the set of all models that represents a given view-type. Further, let us consider the graph $G_{mod} := (V, E, s, t)$ A sequence $(v_0, ..., v_i, v_{i+1}, ..., v_n)$ represents a path of a graph. The graph G_{mod} is an out-tree which means that it contains a vertex v_0 which can be considered as the root of the tree. For each other vertex v_i there is exactly one path from v_0 to v_i. In the context of our problem the vertices $V \subseteq M_{VT}$ represent the view-type models which are affected by a specific model change of the primary view-type model v_0. Finally, an edge $e \in E$ indicates that a change of the model $s(e) \in V$ affects another model $t(e) \in V$ which needs to be modified accordingly. In the following, we denote G_{mod} as modification tree which indicates the change propagation according to a given primary view-type model v_0. In terms of Palladio the repository model is the primary view-type model. As soon as components are created, other actions can be performed (e.g. instantiating components). Including a concrete concern solution into the PCM model requires new components and dependencies between components. Accordingly, the assembly model representing the assembly view-type needs to be modified. The newly created components need to be (i) instantiated and (ii) the new connections between the components have to be created or updated. Given the model changes of the assembly view-type the allocation view-type needs to be synchronized in order to allocate the newly instantiated components. Finally, changes in the assembly model might also cause changes in the model representing the usage model view-type. Here, services of the overall system of the PCM model are exposed to the user. Internally, the services are delegated to the component which provides the service. Therefore, the usage model needs to be synchronized if a concern solution is inserted between the system boundary and the delegated component. Only when the components of the concern solution are instantiated, the usage model needs to be adapted.

Based on the definitions and concepts given by [11], we define the model change propagation problem in the context of the PCM as follows:
Given an out-tree G_{mod} with $V \subseteq M_{VT}$ and $v_0 := m_0 \in M_{VT}$ in which m_0 corresponds to the repository model of the PCM. Let us assume the set $S_{G_{mod}}$ includes all paths from G_{mod} starting at v_0 and ending at v_n in which v_n is denoting a vertex which is subject to $\nexists e \in E : s(e) = v_n$ (leaf node). For all paths $p \in S_{G_{mod}}$, the model change that starts at v_0 and propagates through v_n needs to be handled by applying the corresponding forward propagation operation f_{fw_j} to each pair (v_i, v_{i+1}) in the path. The corresponding forward propagation operation f_{fw_j} has to be applied in order to modify the model $v_{i+1} = m_1 \in M_{VT}$ which is affected by a model change of $v_i = m_2 \in M_{VT}$. If (v_i, v_{i+1}) has already been modified in a different path, they do not have to be considered anymore. In order to integrate a given concern solution in a PCM model,

these model modifications have to be performed. Let us denote G^S_R, G^S_{Ass}, G^S_{All}, G^S_U as the source graphs of the repository, assembly, allocation, and usage model. Further, we define G^T_R, G^T_{Ass}, G^T_{All}, G^T_U as the corresponding target graphs. The model modifications can be performed by applying the synchronization operations defined by [11].

$$\begin{array}{ccc} G^S_R & \overset{r_{R\to Ass}}{\longleftrightarrow} & G^T_{Ass} \\ {\scriptstyle \delta_R}\downarrow & \searrow f_{R\to Ass} & {\scriptstyle \delta_{Ass}}\downarrow \\ G'^S_R & \overset{r'_{R\to Ass}}{\longleftrightarrow} & G'^T_{Ass} \end{array}$$

$$\begin{array}{ccc} (G^S_{Ass} = G^T_{Ass}) & \overset{r_{Ass\to All}}{\longleftrightarrow} & G^T_{All} \\ {\scriptstyle \delta_{Ass}}\downarrow & \searrow f_{Ass\to All} & {\scriptstyle \delta_{All}}\downarrow \\ (G'^S_{Ass} = G'^T_{Ass}) & \overset{r'_{Ass\to All}}{\longleftrightarrow} & G'^T_{All} \end{array}$$

$$\begin{array}{ccc} (G^S_{Ass} = G^T_{Ass}) & \overset{r_{Ass\to U}}{\longleftrightarrow} & G^T_U \\ {\scriptstyle \delta_{Ass}}\downarrow & \searrow f_{Ass\to U} & {\scriptstyle \delta_U}\downarrow \\ (G'^S_{Ass} = G'^T_{Ass}) & \overset{r'_{Ass\to U}}{\longleftrightarrow} & G'^T_U \end{array}$$

The model change propagation problem can not be equated with the synchronization problem. This results from the fact, that we consider model modifications which result not only from model inconsistencies. Nevertheless, the forward propagation operations can be applied.

In the following we describe our concern integration approach which realizes the primary repository model change δ_R and the resulting forward propagation operations $f_{R\to Ass}$, $f_{Ass\to All}$ and $f_{Ass\to U}$ to adapt all affected (secondary) models.

5 CONCERN INTEGRATION

In this Section we present our concern integration approach. For illustration, we consider our example system depicted in Fig. 1. We assume an IDS as the concern to be considered as new requirement. To achieve the data for threat analysis, a software architect has to insert the IDS concern between two components, namely the `DataAccess` and `Database` by using the adapter transformation strategy. All relevant IDS components of the concern solution are already contained in the component repository.

For our approach, we introduce several definitions to describe the formal integration process:

- In general, we consider a model X as a set of model elements $x \in X$.
- The function $f_p : Req \to \mathcal{P}(P)$ inputs a required role $r \in Req$ and outputs a set of provided roles $p \in \mathcal{P}(P)$ in which each provided role provides the interface r requires and $\mathcal{P}(P)$ denotes the power set of P.
- The concern integration can be considered as a model transformation which inputs a model conforming to a given meta model and outputs another model which conforms to the same meta model. Therefore, let us consider the function $\mathcal{T} : M \to M$ in which M is the set of all possible instances of given meta model.
- The function $f_{con} : (AC \times AC) \to \mathcal{P}(Con)$ requires a pair of assembly contexts and results in a set of connectors which connects both assembly contexts. *Con* is the set of all connectors. Note that f_{con} can output more than one connectors. For instance, let us consider the assembly contexts a_1, a_2 which instantiates the components c_1, c_2. Further, c_1 contains the required roles $r_1, r_2 \in c_1$ and c_2 contains the complementary provided roles $p_1, p_2 \in c_2$ with $p_1 \in f_p(r_1), p_2 \in f_p(r_2)$. Given a_1 and a_2, $|f_{con}(a_1, a_2)| = 2$.

5.1 Repository Integration

In order to start the integration process, the initial model change δ_R on the repository model $R \in M_{VT}$ needs to be performed. Based on δ_R the remaining view-type models, which are affected by the change, need to be modified. With respect to the adapter transformation strategy the integration process creates and adds a new adapter component to the repository model so that the following statement hold true:

$$|R| + 1 = |\mathcal{T}(R)|$$

In the SA of the RDS, the adapter is inserted between `DataAccess` and `Database`. More precisely, `DataAccess` contains a required role which references an interface. `Database` contains a provided role which references the same interface. In order to insert the adapter between both components, the adapter must contain the complementary roles so that `DataAccess` can be reconnected to the adapter and the adapter to `Database`. More formally, let's assume r_{DA} represents the required role of `DataAccess` and p_{DB} is the complementary provided role of `Database`. In order to insert the adapter component $c_{adapter}$ between both components, $c_{adapter}$ needs to contain the complementary roles $p_{DA}, r_{DB} \in c_{adapter}$ in which $p_{DA} \in f_p(r_{DA})$ and $p_{DB} \in f_p(r_{DB})$. In addition, $c_{adapter}$ needs the corresponding required roles which requires the services provided by a concern. In terms of the IDS as our integration objective, we assume exactly one provided service. More formally, let p_{IDS} be the provided role which references the interface a concern solution provides. In order to integrate the concern solution, the adapter component needs to contain the complementary required role $r_{IDS} \in c_{adapter}$ with $p_{IDS} \in f_p(r_{IDS})$.

5.1.1 Service Effect Specification Integration. To finish the repository integration, the SEFF for the adapter component needs to be created. The SEFF of the adapter considers the order of the service calls to the connected concern. Further, it can be defined when to invoke the services of the concern. There are three different options namely *BEFORE*, *AFTER* and *AROUND*. Selecting *BEFORE* means to invoke the provided services of the concern before calling the services of the extended component. Accordingly, the setting *AFTER* invokes the concern services after calling the services of the extended component. Finally, *AROUND* means that the concern services are invoked before and after calling the services of the extended component. Let us assume, we use the label *BEFORE*. In order to create the SEFF of the adapter the control flow needs to consider the provided services of the concern and the provided

services of Database. To be more accurate, DataAccess requires the services provided by Database which means that DataAccess invokes this services internally. In order to insert the adapter between DataAccess and Database the corresponding roles have been added to the adapter which enables DataAccess to use the provided services of the adapter and not Database any more. However, the adapter does not actually provide all services that DataAccess requires. The actual implementation of the services are contained in Database. Therefore, each time DataAccess invokes a service of the adapter, the adapter internally delegates this call to Database. Depending on the chosen option, the adapter invokes the services of a concern before, after or around the delegated service call to Database. More formally, $EC_{DA \to DB}$ is the set of all external calls from DataAccess to Database which are abstract actions. Recall that a SEFF primarily consists of a chain of abstract actions (e.g. internal action, external call action). Given the abstract actions a_1 and a_2 then $\mapsto$ indicates an ordered sequence of abstract actions. For instance, $a_1 \mapsto a_2$ means that after a_1, a_2 follows. After the application to the external calls from DataAccess to Database, the corresponding SEFF in the adapter which consists of a sequence of external service calls is added. Here, the service call to the concern has been invoked before calling one of the services of Database. Thus, consider the following formal description:

$$\forall ec \in EC_{BT \to PS}, \exists_{=1} seff \in c_{adapter} : \\ start \mapsto ec_{IDS} \mapsto ec \mapsto stop$$

$c_{adapter}$ denotes the adapter component, ec_{IDS} the external call to the IDS and *start*, *stop* denote the abstract actions defining the start and end of the abstract action sequence, respectively.

5.2 Assembly Integration

Considering the modification tree G_{mod}, after the initial model change δ_R has been performed the assembly model needs to be modified in order to instantiate all newly created components and to establish the corresponding connections. In the following we describe the forward propagation operation $f_{R \to Ass}$.

In general, the assembly model consists of a set of assembly contexts and a set of connectors. Given the updated repository model $\mathcal{T}(R)$ the sets of assembly contexts *AC* and connectors *Con* contained in the assembly model *Ass* need to be modified in order to instantiate the concern solution. An assembly context for the adapter component has to be created as well as for the components contained in *R* which represent the structure of the IDS. Recall that the components of the IDS concern solution are already contained in *R*. In order to adapt *AC* the assembly contexts of the adapter component and the concern solution components need to be created and added so that after applying the concern integration engine the following statement holds true:

$$|AC| + N_C + 1 = |AC'|$$

N_C is the number of components representing the concern solution of the IDS and $AC' \subset \mathcal{T}(Ass)$. Finally, receiving $Con' \subset \mathcal{T}(Ass)$ is more complex. First, the previous connector of DataAccess and Database needs to be removed and replaced by two new connectors which connects DataAccess with the adapter component and the adapter component with Database. More formally, let us denote $a_{DA} \in AC$ the assembly context of DataAccess and $a_{DB} \in AC$ the assembly context of Database. After the integration process has been applied the following statement holds true:

$$\exists_{=1} a_0 \in AC' : |f_{con}(a_{DA}, a_0)| = 1 \wedge |f_{con}(a_0, a_{DB})| = 1$$

a_0 represents the assembly context of the adapter component. Additionally, connectors from the adapter to the consumed services of the concern have to be created. Recall that a concern solution is represented by a repository model. For that reason, after the assembly contexts have been created, the corresponding connectors have to be created. In order to resolve the connectors automatically we made an assumption which the components of a given concern solution must satisfy. We assume that for each required role contained in one of the components of the concern solution there is exactly one complementary provided role. Due to this consideration, a bijective function can be created. More formally, let us denote *RR* the set of all required roles and *PR* the set of all provided roles contained in an arbitrary concern solution. Consider the bijective function $g_p : RR \to PP$ with $r = g_p^{-1} \circ g_p(r), r \in RR$. Based on the previously made assumptions, the following holds true:

$$\forall r \in RR, \exists_{=1} p \in PR : p = g_p(r)$$

Based on the previous considerations, all connected role pairs can be resolved and the according connectors can be created and added to *Con*. Finally, after the integration process has been applied the following statement holds true:

$$|Con'| = |Con| - 1 + 2 + N_P + |RR|$$

N_P is the number of the provided services of the concern solution. Recall that in terms of the IDS example we assumed that the IDS provides only a single service, so that $N_P = 1$. $|RR|$ is the number of connectors that have been added in order to connect the assembly contexts of the concern solution. This follows from the bijective property of g_p.

5.3 Allocation Integration

As a consequence of the forward propagation operation $f_{R \to Ass}$ the allocation model is inconsistent and need to be synchronized. Therefore, in the following we describe the forward propagation operation $f_{Ass \to All}$.

In order to synchronize the allocation model *All* for each newly created assembly context there needs to exist an allocation context. More formally:

$$|All'| = |All| + |(AC' \setminus AC)|, All' = \mathcal{T}(All)$$

must hold true.

5.4 Usage Model Integration

Finally, in this Section we describe the forward propagation operation $f_{Ass \to U}$. Each PCM model is represented to the user as a system which provides several services. Internally, the system consists of different components representing the software architecture. The services provided by the system are delegated to the components which actually provide the services. However, these components or more precisely the corresponding provided roles are referenced by so called entry level system calls in the usage model. When an adapter is inserted between the system and a delegated component, the references of the entry level system calls have to be

updated only, if the adapter is instantiated in the assembly model. Therefore, let us denote $E \subset U$ as the set of entry level system calls and $PR \subset R$ the set of all provided roles contained in the repository model. Further, consider the relation $U_{E \to PR} \subseteq E \times PR$ which relates an entry level system call to a corresponding provided role. Given the delegated component c_{del} and the adapter $c_{adapter}$ which is inserted between the system and c_{del}. When the concern integration engine has been applied, the sets $E' \subset \mathcal{T}(U)$ and $PR' \subset \mathcal{T}(R)$ results so that the following holds true:

$$\forall (e, pr) \in (U'_{E' \to PR'} \setminus U_{E \to PR}) : pr \in c_{adapter}$$

In other words, all provided roles contained in c_{del} which are referenced by the system entry must be replaced with the corresponding provided roles of the adapter. Considering the IDS example, if an IDS is not inserted between the system and one of the delegated components or not instantiated, then $U_{E \to PR} = U'_{E' \to PR'}$.

5.5 Integration in PerOpteryx

The design space of PerOpteryx results by all identified degree of freedom instances. In order to combine our concern integration approach with PerOpteryx we extended the degree of freedoms by a so called *Concern Degree*. A concern degree specifies a concern as primary changed value. In terms of our running example the IDS represents such a concern. Moreover, a degree of freedom specifies different design options. AppSensor and OSSEC represent two different concern solutions of an IDS concern and can be considered as design options.

Finally, we integrated the concern integration engine into PerOpteryx.

6 EVALUATION

We applied our approach on an industrial example. We have already introduced a simplified version as running example (see Fig. 1).

In this evaluation we show the benefits of our approach by applying our approach on three different application scenarios. In each scenario we annotate a different component which is supposed to be extended by a concern. More precisely, we consider an IDS as a concern and AppSensor as well as OSSEC as two possible concern solutions. We consider the quality attributes performance, costs and security level. The performance attribute is quantified by considering the response times of the system. The cost attribute will be derived by considering the cost annotations of each component as well as the variable costs (e.g., the higher the CPU utilization, the higher the costs). We describe the security quality as a not-quantified quality attribute [12] in order to represent the knowledge of the software architect. We annotate the access restriction level of AppSensor with *low*-quality and OSSEC with *high* quality [4]. The annotations are considered by PerOpteryx and affect the resulting Pareto-optimal candidates. Due to a higher security level, our OSSEC model consumes more resources, while AppSensor has less resource demand and a lower access restriction level. As a result, OSSEC is more secure but requires higher resources while AppSensor is less secure but more efficient. By this setting, a software architect can weight up the differences in resource demand, response times, cost, and security level to find the best concern solution for a given scenario.

[4]Please note, that we have chosen both levels for access restriction levels for illustration.

6.1 Scenarios

In the first scenario, we annotated the `DataAccess` component in order to insert an IDS between `DataAccess` and the `Database` component. By this, the IDS analyses the data flow in order to prevent attacks to the database (e.g. *SQL Injection*). In the second scenario, the component `ServiceEngineerWebsite` is annotated. This component is not depicted in Fig. 1. For the sake of simplification, we reduced the complexity of the RDS to the set of components which describe the main functionality of the system. The `ServiceEngineerWebsite` represents a system entry for software engineers. In order to detect suspicious behaviour at this entry point, a software architect may insert an IDS between `ServiceEngineerWebsite` and `DataAccess`. In the last scenario, we annotate the components `ServiceEngineerWebsite` and `ServiceParser` in order to demonstrate the scenario of annotating several components. The `ServiceParser` component is called by `RDSConnectionPoint`. The primary task of `ServiceParser` is to compute the incoming data from `RDSConnectionPoint` before the data is processed. Incoming user data could include malicious character sequences. Thus, it may be another interesting position to include such an IDS solution.

6.2 Result

For each run with PerOpteryx we evaluated more than 900 architecture candidates. The result of each run is depicted in Fig. 2. The first scenario results in 20 Pareto-optimal candidates. The results are illustrated in Figure 2a. Figure 2a shows the higher resource demand by the large gap between response times and costs of both solutions. RDS invokes many service calls to the database. Each time, the intrusion detection engines are invoked as well. Therefore, the response times for the candidates which use OSSEC are much higher than candidates which uses AppSensor due to the higher resource demand of OSSEC.

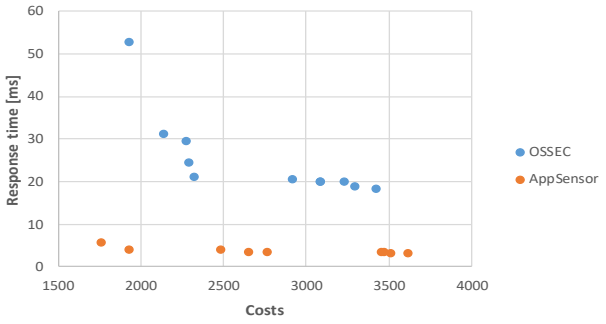
The second run results in a set of 13 Pareto-optimal candidates. The results are illustrated in Figure 2b. Compared to the first scenario, the intrusion detection engine is called less frequently in Scenario 2. This makes the difference in the response time in Scenario 2 less pronounced.

The third scenario results in 16 Pareto-optimal candidates. The results are illustrated in Figure 2c. The scenario contains additional processing steps (analysing the data by an IDS) that are introduced at two different locations. The additional overhead of another component to be monitored again increases the response time.

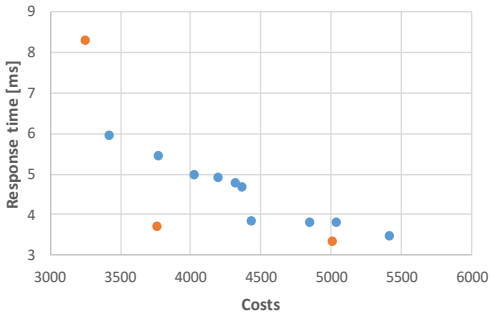
A software architect could choose OSSEC if security is more important than performance or AppSensor in the other case with respect to the costs of each candidate. In addition, the software architect receives information about the cost of response time of a more secure solution.

6.3 Discussion

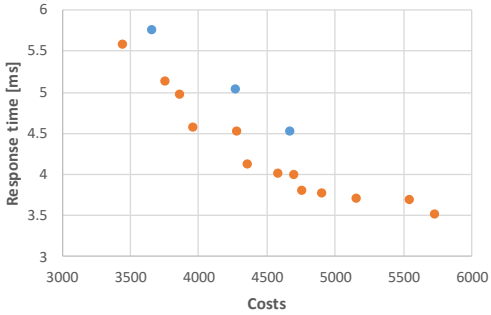
The different application scenarios demonstrated the benefits of our approach. Extending PerOpteryx by our concern integration approach makes it comfortable to evaluate different concern solutions. Our scenarios show the effects on performance and costs



(a) Pareto-optimal candidates of the first scenario



(b) Pareto-optimal candidates of the second scenario



(c) Pareto-optimal candidates of the third scenario

Figure 2: Results of the three scenarios

when integrating different solutions in different locations of the SA. A software architect can then decide which solution and which locations to choose according to individually defined quality requirements.

Without our automated integration approach a software architect would have to evaluate each solution separately. Further, he would have to combine the results by regarding possible dominated architecture candidates. In addition, the modelling effort, that emerges without our approach, is also time-consuming and potentially error prone. Moreover, we demonstrated with the industrial example system that not only the choice on a particular solution can influence the quality of the system, but also the location (i.e. target in the system) where the solution is integrated. When reusing subsystems the software architect always has the choice between several solutions and several possibilities to integrate a solution in the target SA. Using our approach supports such a decision making process by a light-weight upfront analysis on the effects of SA decisions on the resulting quality. Using these results, the software architect can decide which design decision best meets the requirements based on known metrics, such as response time and costs.

7 RELATED WORK

GeKo ([13]) is a meta model independent aspect-oriented model weaver which integrates several models. For the model integration process, GeKo considers three different models, namely *base-*, *pointcut-* and *advice*-model. GeKo uses the pointcut rule set as an instruction set to integrate the advice model into the base model. With other words, the base-model is suspect to extension, while the pointcut-model defines join points used to select elements of the base-model to be modified, and the pointcut-model represents a correspondence to elements of the base-model to be modified. The advice-model contains instructions how to modify a join point. To apply the proper changes for a given join point there is a mapping between the pointcut- and the advice-model. However, a software architect would have to define all pointcut and corresponding advice model elements individually for each integration location. Further, due to GeKo's genericity both models would have to be designed meta model specific. For another PCM instance another individual pointcut and advice models are necessary to be created. As a result, the reusability of models is severely restricted.

The *Concern-Oriented Reuse* (CORE) approach [14] defines several concepts our model integration approach is based on. CORE defines concerns as main units of reuse and consider them as design decisions. Different features of a given concern are described by a feature model. Additionally, with a goal model CORE assesses how a particularly selected feature configuration of the concern impacts high-level stakeholder goals or non-functional requirements. However, in the context of our approach we are mainly interested in software quality prediction at design time. In addition, with CORE a software architect selects a feature configuration which satisfies the defined goals (defined in the goal model) best. Nevertheless, the selection can conflict with other quality attributes. Consequently, to determine the best configuration is hard. In contrast to our approach, we target on analysing which concern solution satisfies the software quality best with respect to other model changes which again affect the software quality. Further, the concern reuse process defined by CORE is mainly designed result in working software and less for design time prediction of the resulting software quality.

Another category of related work is the field of *Software Product Lines* (SPL). There are several works in the context of SPL such as [15] or [16]. In software companies it is a common practice to make copies of existing software products which only differs in custom specific features. Many copies are hard to maintain. Therefore, the concepts of SPL have been introduced which extracts a common core functionality for all products. The differences between the copies are treated as features which extends the core functionality. In contrast to our approach, SPLs are applied at code level and are less focused on improving design decisions at design time.

8 CONCLUSIONS

In this paper, we presented a design-time approach for rapid evaluation of complex design decisions considering quality attributes by using software architecture models. Our approach focuses on time-efficient design decision evaluation with the focus on quality attributes such as performance, reliability, and costs. System developers or software architects can benefit from our approach by design time estimations of design decisions such as integrating complex subsystems. With our approach there is no need to acquire or develop expensive subsystems for the upfront prediction about the expected quality attributes. By our approach, existing models can be better reused in different contexts. Making models better reusable reduces the modeling effort of such model-based approaches and makes them more time and cost-efficient. As a result, quality risks on the side of the software in the complex interplay between hardware and software, as is often the case in cyber-physical-systems, can thus be evaluated upfront.

In the context of our approach, we formalized the problem domain as a model synchronization problem and described a formal model integration process. We presented an in-depth description of our integration operations and their implementation. Further, we described the integration of our approach into a design-time decision making process, namely PerOpteryx in order to consider complex design decisions and different solutions for such design decisions. To do so, we extended PerOpteryx' design space by a degree of freedom for considering different solutions. We showed four different application scenarios showing the benefits of our approach using a real-word cyber-physical-system. In the course of the scenarios, we have shown how our approach can be used to select the best design decision whenever its influence on the quality attributes of the system is unclear.

REFERENCES

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

[2] A. Martens, H. Koziolek *et al.*, "Automatically improve software models for performance, reliability and cost using genetic algorithms," ser. WOSP/SIPEW ICPE'10, 2010.

[3] F. Bachmann, L. Bass, M. Klein, and C. Shelton, "Designing software architectures to achieve quality attribute req." *SW Proceedings*, 2005.

[4] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya, "Archeopterix: An extendable tool for arch. optim. of aadl models," in *MOMPES '09*, 2009.

[5] T. De Gooijer, A. Jansen, H. Koziolek, and A. Koziolek, "An industr. case study of performance and cost design space explor." in *ICPE*. ACM, 2012.

[6] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: automated application of tactics in multi-objective software architecture optimization." QoSA-ISARCS 2011, 2011.

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

[8] A. Busch, Y. Schneider, A. Koziolek, K. Rostami, and J. Kienzle, "Modelling the Structure of Reusable Solutions for Architecture-based Quality Evaluation," ser. CloudSPD'16. IEEE, 2016.

[9] P. Langer, K. Wieland, M. Wimmer, and J. Cabot, "From uml profiles to emf profiles and beyond," in *TOOLS*. Springer, 2011.

[10] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in CS*. Springer, 1995.

[11] F. Hermann *et al.*, "Correctness of model synchronization based on triple graph grammars," in *MoDELS*. Springer, 2011.

[12] A. Busch and A. Koziolek, "Considering not-quantified quality attributes in an automated design space exploration," in *QoSA*. IEEE, 2016.

[13] M. Kramer, "Generic and extensible model weaving and its application to building models," Master's thesis, KIT, 2012.

[14] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *MoDELS*. Springer, 2013.

[15] B. Klatt, K. Krogmann, and C. Wende, "Consolidating Customized Product Copies to Software Product Lines," in *WSRE'14*, 2014.

[16] H. Eichelberger, C. Qin, R. Sizonenko, and K. Schmid, "Using ivml to model the topology of big data proc. pipelines," in *SPLC*. ACM, 2016.