Challenges in the Evolution of Palladio—Refactoring Design Smells in a Historically-Grown Approach to Software Architecture Analysis

Robert Heinrich, Jörg Henss, Sandro Koch, and Ralf Reussner

Abstract In this chapter, we provide insights into Palladio—a tool-supported approach to modelling and analysing software architectures. Palladio serves as a case study for the evolution of historically-grown approaches to model-based analysis. We report about design smells in Palladio's metamodel and simulators caused by evolution and growth over several years. Design smells are structures that require refactoring. Decomposition is key for refactoring these design smells. We discuss how techniques for decomposition and purpose-oriented composition can help refactoring design smells in Palladio's metamodel and simulators.

11.1 Introduction and Problem Statement

Palladio is a tool-supported approach to modelling and analysing software architectures for various quality properties [Reu+16]. It is named after the Italian Renaissance architect Andrea Palladio. Initially, Palladio was focused on performance and then has been extended for several quality properties, such as reliability [Bro+12], scalability and elasticity [Leh14], energy consumption [Sti18], security [TH16], confidentiality [SHR19], and maintainability [Ros+15]. With Palladio, costly changes to software after it has been implemented can be avoided by analysing the quality of a software system of a given architecture early in development. Decisions in software design are typically made on the basis of experience

R. Heinrich (✉) · S. Koch · R. Reussner
Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: robert.heinrich@kit.edu; sandro.koch@kit.edu; ralf reussner@kit.edu

J. Henß
FZI Research Center for Information Technology, Karlsruhe, Germany
e-mail: henss@fzi.de

or, when lacking those, by making an educated guess. The information provided by the Palladio approach enables to choose the best-suited design alternative and to make trade-off decisions [Hei+18].

The Palladio approach consists of three essential parts that are designed to work hand in hand [Reu+16]. First, the *Palladio Component Model* (PCM) as a domain-specific modelling language defined in the form of a metamodel is targeted at specifying and documenting software architectural knowledge. Second, various analysis techniques ranging from queuing network analysis to discrete-event simulation can be applied to predict the quality of a system modelled based on the PCM. Third, the Palladio approach is aligned with a development process that comprises several developer roles and activities tailored to component-based software design.

In this chapter, we focus on the evolution of the PCM and the associated simulators. We understand the term simulator to be a software tool that implements one or more techniques of simulative analysis for approximating the quality properties of a system under study. We understand the term simulation to be the execution of a stimulative technique using a simulator. Simulation is therefore an example of automated analysis (cf. Chap. 2 of this book [Hei+21]). The PCM is an established and widely used metamodel. The PCM and the associated simulators provide various useful features for quality modelling and analysis of component-based software architectures.

We use the term feature to specify what a modelling language should express and what a simulator should analyse on a conceptual level. A feature of a metamodel (or a modelling language in general) is an abstraction of a thing to be modelled [HSR19]. Examples of language features in Palladio are amongst others those for modelling the component structure, component-internal behaviour, system usage, and performance-related annotation [SHR18]. A feature of a simulator is an abstraction of a property to be analysed by simulation. Examples of simulator features in Palladio are amongst others those for analysing user behaviour, system behaviour, resource usage, and for eliciting performance-related measurements.

The PCM consists of 203 classes dispersed amongst 24 packages [HSR19]. It is organised into five partial metamodels. Since its inception in August 2006, the PCM has a long history of evolution. There are at least 12 documented extensions to the PCM publicly available. However, many more extensions exist that are not publicly available (e.g., student theses, experimental, incubation). Owing to its historically-grown structure, the PCM exhibits some shortcomings such as package structure erosion, uncontrolled growth of dependencies, instance incompatibility, and incompatible extensions. The simulators for reasoning about model instances of the PCM show similar size and complexity. For example, the original simulator SimuCom [Bec08] consists of 231 classes in 50 packages. Due to historical growth, also the simulators show shortcomings such as package structure erosion, uncontrolled growth of dependencies, underdefined semantics, and incompatible extensions.

This chapter provides insights into the evolution of Palladio to serve as a case study for decomposition and composition of model-based analysis. We report about design smells in the metamodel and simulators caused by evolution and growth over several years. Design smells are structures that indicate the violation of fundamental design principles and therefore negatively affect the quality of the metamodel and simulators. Thus, design smells require refactoring. Decomposition is key for refactoring design smells in Palladio's metamodel and simulators. Due to the rigorous quality assurance process of Palladio, most of the design smells have already been addressed. Nevertheless, the design smells reported may provide food for thought for others evolving historically-grown metamodels and simulators and motivate the usage of techniques for decomposition and composition. We discuss how techniques for decomposition and purpose-oriented composition can help refactor design smells in the metamodel and simulators. This chapter, therefore, illustrates concepts discussed in Chaps. 4 and 5 of this book [Hei+21].

The remainder of this chapter is structured as follows. Section 11.2 gives an overview of Palladio's modelling environment—the Palladio-Bench. We report about design smells in the PCM in Sect. 11.3 and in the simulators in Sect. 11.4. The application of techniques for decomposition and composition to resolve design smells is described in Sect. 11.5. This chapter concludes in Sect. 11.6.

11.2 Overview of the Palladio-Bench

Before discussing design smells in the evolution of Palladio, this section gives a detailed overview of the three essential parts of the Palladio approach [Reu+16]—the domain-specific modelling language PCM, the various analysis techniques, and the development process comprising several developer roles. These three parts of the Palladio approach are implemented in the Palladio-Bench that is based on the Eclipse *integrated development environment* (IDE) [Hei+18].

The PCM consists of the partial metamodels shown on the left-hand side in Fig. 11.1 to reflect different architectural views on a software system. The several developer roles use graphical editors provided by the Palladio-Bench [Hei+18] to specify the partial models of the Palladio approach. The component developer designs the software component specifications. The component repository model is created by the component developer to design the software components and their required and provided interfaces stored in a repository. Moreover, the component developer specifies the components' inner behaviour in the form of the so-called *Service Effect Specification* (SEFF). A SEFF expresses internal actions of a component's services typically annotated with quality-specific information depending on its context and external service calls. The software architect designs the software architecture in the system model by assembling components from the repository. Thus, the quality of a system can be estimated with respect to the component assembly described in the system model. The system deployer specifies the execution containers (i.e., servers) including their processing resources

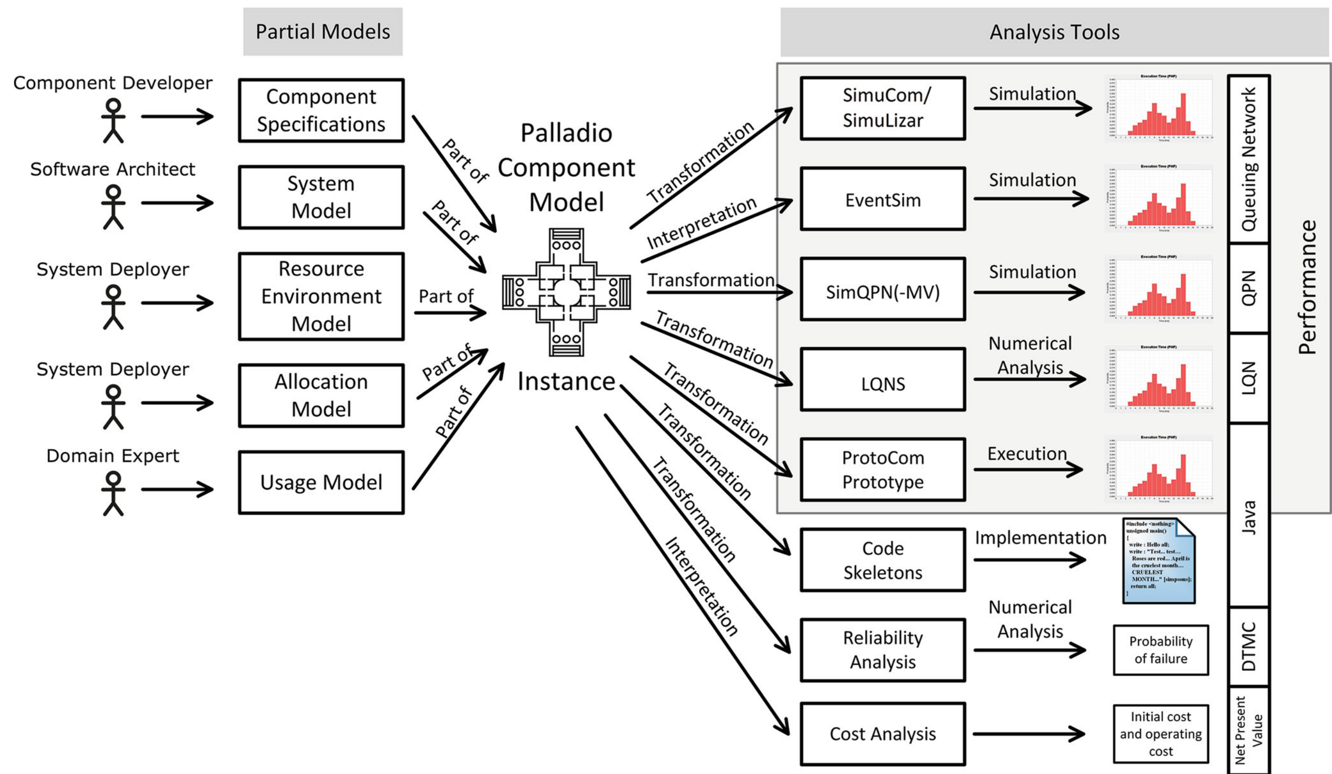

Fig. 11.1 Partial models of the PCM and transformation to analysis tools, extension of [Reu+16]

(i.e., CPU, hard disk, and network) in the resource environment model. For each execution container, quality-relevant properties like processing rate of the CPU are part of the resource environment model. Moreover, the system deployer describes the deployment of the components to the execution containers in the allocation model. The domain expert specifies the workload of the system in terms of user behaviour and usage intensity in the usage model.

The Palladio-Bench offers several analysis tools for reasoning about quality depicted on the right-hand side in Fig. 11.1. Note, although the focus of this chapter is on simulators, we deliberately depict other tools of the Palladio-Bench in Fig. 11.1 to give a comprehensive overview. We therefore introduce the broader term analysis tool here (cf. Chap. 2 of this book [Hei+21]). An analysis tool in the context of the Palladio-Bench is a software tool that implements one or more analysis techniques for approximating the quality properties of a system under study. Analysis tools for estimating the performance of a software system are central to the Palladio-Bench, and a wide range of tools are available. These performance tools are highlighted in the grey box in Fig. 11.1 and differ mainly in their range of functions, result accuracy, and analysis speed. The Palladio-Bench also offers tools for the analysis of reliability [Bro+12] and prediction of costs [Mar+10] as well as various extensions, e.g., for the analysis of energy consumption [Sti18], security [TH16], confidentiality [SHR19], and maintainability [Ros+15], not depicted in the figure.

Palladio's original simulator SimuCom [Bec08] is a discrete-event performance simulator that estimates response times of both, system-level and component-level services, as well as utilisation of processing resources specified in the resource environment. The performance simulator SimuLizar [BLB13] is focused on analysing self-adaptations in cloud computing environments, e.g., when scaling out components by replication. EventSim [MH11] is a discrete-event performance simulator that complements SimuCom in that it primarily addresses highly complex models in simulation by applying event-scheduling simulation techniques. Besides the simulators, the Palladio-Bench offers tools for transforming model instances of the PCM to the formalisms *queuing Petri net* (QPN) and *layered queuing network* (LQN). These are established formalisms and commonly used for software performance prediction independent of the Palladio approach. ProtoCom [Bec08] is a tool provided by the Palladio-Bench to create performance prototypes in the form of Java code that mimic demands to different types of processing resources to evaluate the system performance in a realistic environment.

The reliability analysis tool of the Palladio-Bench estimates software and hardware failure potentials using *discrete-time Markov chain* (DTMC) [Bro+12]. The simple cost analysis provided by the Palladio-Bench allows to assign costs to software components and hardware that is then used to estimate the initial and operating costs of the system [Mar+10].

Referring to the analysis orchestration strategies introduced in Chap. 5 of this book [Hei+21], the Palladio-Bench applies the single analysis orchestration strategy. The aforementioned developer roles use the Palladio-Bench to create a domain-specific model of the system to conduct quality analyses based on one of the aforementioned analysis tools. The Palladio-Bench transforms the domain-specific

model into an analysis model specific to the given analysis tool for quality analysis. After the analysis has been finished, the results are lifted back to the Palladio-Bench. The Palladio-Bench in turn displays the results to the developers. There is no interaction between the individual monolithic analysis tools.

11.3 Design Smells in the Palladio Component Model

In this section, we give examples of design smells that occurred in the PCM while it evolved over the course of several years. These design smells serve as motivation for the decomposition and purpose-oriented composition of the PCM to refactor the design smells as described in the following sections.

In object-oriented design, the term design smell is commonly understood as a structure that indicates the violation of fundamental design principles and therefore negatively affects quality properties of the system like maintainability and evolvability. Design smells in object-oriented design are classified as creational, structural, and behavioural smells [GS13].

Design smells not only occur in the object-oriented design of software systems but also in the design of metamodels. Strittmatter [Str19] investigated design smells in metamodels and identified that many structural design smells known in object-oriented design can also be found in metamodel design. This is reasonable as there are many commonalities in object-oriented design and metamodel design from a structural point of view. Both, object-oriented design and metamodel design, specify classes and their attributes, package structures, as well as dependencies between classes [Str19]. Creational and behavioural smells from object orientation cannot be found in metamodels as with respect to these categories object-oriented design and metamodel design differ [Str19].

In the following, we discuss some examples of design smells that refer to the modularity of metamodels and explain their occurrence in the PCM to demonstrate the need for refactoring by decomposition and purpose-oriented composition of the PCM. We thereby focus on design smells on the level of the package structure of the metamodel or on the level of metamodel files. A complete overview of metamodel design smells is given in [Str19].

Language Feature Scattering The content of a metamodel is logically partitioned by its package structure. A language feature is implemented by one or several classes in the metamodel. Language features are hard to grasp, if they are not adequately reflected in the package structure. If classes that constitute a language feature are spread over multiple packages that do not share a meaningful parent, it is defined as *Language Feature Scattering* [Str+16]. When a language feature is scattered over multiple packages, it is hard to understand the purpose of such a package without considering all other dependent packages. Consequently, this smell hampers the comprehensibility of the metamodel. Also the maintainability of the metamodel may be negatively affected. Language Feature Scattering occurs in the

PCM. For example, the language features for modelling the software repository, resource interfaces, middleware infrastructure, events, performance, and reliability (cf. [SHR18]) are all scattered over multiple packages.

Package Blob A package that contains classes of multiple language features is defined as *Package Blob* design smell [Str+16]. The Package Blob smell reduces understandability of the package as one needs to identify and understand all the contained language features and their respective classes in order to understand the package. Furthermore, it unnecessarily increases complexity and negatively affects reusability of the package as it is not possible to selectively depend only on the necessary language features. Examples for the Package Blob smell in the PCM are data types and the abstract component-type hierarchy, which both are located in a single package, namely the repository package.

Metamodel Monolith The *Metamodel Monolith* design smell is defined as a metamodel file that implements multiple language features. This is the analogy of the Package Blob on the level of metamodel files. The Metamodel Monolith smell negatively affects the reusability of the metamodel file as it is not possible to selectively depend only on the necessary language features. The complexity of the metamodel files is unnecessarily increased, and the understandability is reduced due to lack of modularity [Str19]. The Metamodel Monolith smell occurs in the PCM as the entire PCM with all its packages is contained in a single metamodel file.

11.4 Design Smells in the Simulators

In this section, design smells in the Palladio simulators are discussed. These design smells serve as motivation for the decomposition and purpose-oriented composition of the simulators to refactor the design smells as described in the following sections.

Stepney [Ste12] collected smells in scientific simulation. Some of these smells refer to simulator design and can also be found in similar form in the simulators of Palladio. Moreover, we identified additional smells in the Palladio simulators that we could not yet find in the literature. These additional smells result from our professional experience in using the simulators of Palladio both, in academic and industrial projects. In the following discussion, design smells inspired by Stepney are marked by the reference [Ste12].

Amateur Science [Ste12] The *Amateur Science* smell denotes simulator development without the involvement of domain experts, e.g., because the simulator developers assume to be familiar with a given domain, and thus making simplifying assumptions. This smell is represented by modelling languages and simulators that are oversimplified for the given analysis task. This may result in neglected domain knowledge and thus negatively affect the accuracy of the simulation results. In the simulators of Palladio, the simulation of the network resources is implemented in

a very simplistic way. The assumption was made that the impact of the network resources on the accuracy of the simulation results would not be of significance. However, with this assumption, we underestimated the impact of network resources on the distortion of service response times [KBH07, Ver+07]. Especially for modern distributed systems, network latency and throughput may have significant impact on the overall system performance. Therefore, network resources need to be adequately considered in simulation to achieve accurate results.

Analysis Paralysis [Ste12] Simulator developers may spend too much time analysing and modelling the domain, trying to get everything perfect, and not getting to the simulation. This is defined as the *Analysis Paralysis* smell. This smell is represented by modelling languages and simulators that are unnecessarily complex or detailed for the given analysis task. As a consequence, developing and maintaining the modelling languages and simulators is more time-consuming and error-prone than actually necessary. The PCM allows the modelling of a component-type hierarchy to provide support for an iterative specification of components. Components can be specified at different levels of abstraction based on the amount of knowledge currently available for these components [Reu+16]. However, for the goal of performance analysis, the structure of the component-type hierarchy has no effects on the simulation. Thus, the PCM is unnecessarily detailed for the task of performance analysis with respect to the component-type hierarchy as it is not used for analysing the performance of the software system in the simulators of Palladio.

Everything but the Kitchen Sink [Ste12] Simulator developers may add irrelevant features not related to the actual analysis task to a modelling language and simulator. This is denoted as the *Everything but the Kitchen Sink* smell. In contrast to the Analysis Paralysis smell, the modelling language and simulator do not show unnecessarily complex or detailed features but features that are not relevant to the analysis task at all, e.g., adding a reliability-related feature to a pure performance simulation. The Everything but the Kitchen Sink smell is represented by convoluted and monolithic modelling languages and simulators with unclear focus and purpose and seldom used or even unused features. As a consequence, developing and maintaining the modelling languages and simulators is more time-consuming and error-prone than actually necessary. The main purpose of Palladio's simulators is software architecture-based performance analysis. However, features like the *Accuracy Influence Analysis* [Gro13] and the *Sensitivity Analysis* [Bro+12] are part of SimuCom. Although these features are seldom used, each change in Palladio (e.g., updating the Java version or changes to the PCM) potentially requires effort to keep them functional. Moreover, the strong interconnection of these features to other features of the simulator may result in negative side effects.

Living Flatland [Ste12] When simulator developers use a wrong level of abstraction like simulating a 2D space and then naively translating the results in a 3D space, it is defined as the *Living Flatland* smell. This smell may negatively affect the accuracy of simulation results. In Palladio, for example, a simple processor-sharing scheduler was implemented in the simulator SimuCom, with the assumption made,

that this kind of scheduler is sufficient to approximate all kinds of CPU-scheduling policies. This resulted in inaccurate simulation results and, as a consequence, development overhead, because the *Linux Exact Scheduler* [Hap08] had to be implemented in order to fix shortcomings caused by the initial assumption.

Underdefined Semantics The semantics of the input model of a simulator may not exactly correspond to the semantics actually implemented in the simulator as the simulator's semantics is underspecified. This is denoted as the *Underdefined Semantics* smell. This smell results in gaps in semantics definition of model and simulator, and thus ad hoc definition of semantics during simulator development. Moreover, there is a high risk that the simulation will provide faulty results due to underdefined semantics. Furthermore, underdefined semantics can lead to semantic shifts, rendering older models invalid as they were created with a different understanding of model elements in mind. This can also interfere with the reproducibility of simulation experiments. In the early years of Palladio development, several extensions were made to the PCM without defining a clear semantic mapping to the simulator SimuCom. Examples are the output parameters and the fork join actions. This led to the problem that semantics were defined in an ad hoc way during simulator development and had to be adjusted in several iterations or are still not well defined up to now. The interested reader is referred to Chap. 9 of this book [Hei+21] where further discussion on the topic is given.

Excessive Events/Event Flooding A simulator utilising an unnecessarily large number of events is defined as the *Excessive Events* or *Event Flooding* smell. The massive creation of unnecessary events in simulation largely impacts the execution efficiency. Therefore, simulator developers should try to minimise the number of events to be managed in simulation. In Palladio, we discovered several shortcomings in the realisation of the resource schedulers in SimuCom. Requests created excessive numbers of events when running in fair-share mode in overload scenarios leading to starvation and crashes in simulation.

Simulator Feature Scattering A simulator is logically partitioned by its component structure. A simulator feature is implemented by one or several classes of the simulator. Simulator features are hard to grasp, if they are not adequately reflected in the component structure. Classes that implement a feature of a simulator may be spread over multiple components of the simulator that do not share a meaningful parent. This is defined as the *Simulator Feature Scattering* smell. When a feature of a simulator is scattered over multiple components, it is hard to understand the purpose of such a component without considering all other dependent components. Consequently, this smell hampers the comprehensibility of the simulator. Also, the maintainability of the simulator may be negatively affected. In Palladio's simulator SimuLizar, for example, the simulator feature to handle the language feature *Usage* [SHR18] that contains amongst others the usage model is implemented in 18 classes scattered over three components.

Simulator Component Blob A simulator component that contains classes of multiple simulator features is defined as *Simulator Component Blob* smell. The

Simulator Component Blob smell reduces understandability of the simulator component as one needs to identify and understand all the contained simulator features and their respective classes in order to understand the component. Furthermore, it unnecessarily increases complexity and negatively affects reusability of the simulator component as it is not possible to selectively depend only on the necessary language features. In EventSim, for example, the different features of the simulator like simulation of users, resources, and network are heavily interwoven in the core simulator component [MH11].

Simulator Monolith If there is no decomposition of the simulator at all, we denote this design smell as *Simulator Monolith*. The Simulator Monolith smell negatively affects the reusability of the simulator as it is not possible to selectively depend only on the necessary simulator features. The complexity of the simulator is unnecessarily increased, and the understandability is reduced due to lack of modularity. There is no example of a Simulator Monolith in the Palladio context as all simulators are at least partially decomposed.

Global State Object and God Parameter The state of an entire simulation may be stored in a single object. This is defined as the *Global State Object* smell. A global state object is an object that encapsulates large parts of the world model of a simulator. Thus, it is an instance of a god class [LM06]. Every entity in the simulation has access to this object. Entities in the simulation usually access the global state object directly to query and manipulate the global state of the simulation similar to the blackboard pattern [Bus+96]. When extending the simulator, developers usually add more and more fields to the global state object. This introduces large maintainability problems when changing fields as no clear interfaces and access restrictions exist. This design smell can be accompanied by the *God Parameter* smell, a field in the global state object that can be used to manipulate the behaviour of entities ignoring any existing encapsulation. In SimuLizar, for example, the simulation control information is passed through the whole simulation, if it is required or not.

Intrusive Extension A simulator may have extensions that are tightly coupled into the code base. This is defined as the *Intrusive Extensions* smell. The intrusive extensions can clutter the codebase and introduce technical debts. Furthermore, they may cause dead code in the long term if not used anymore. In SimuCom, the reliability extension [Bro+12] is an example of this design smell. Though being used rarely, it could not be disabled in the generation of SimuCom code and led to several problems and bugs.

11.5 Application of Decomposition and Composition Techniques to Palladio

This section provides insights into the application of decomposition and composition techniques to Palladio. Applying these techniques enables to fix many of the aforementioned design smells in the metamodel and simulators. First, we discuss techniques for the decomposition and composition of the PCM. This serves as a preparatory step for the decomposition and composition of the associated simulators. Then, we discuss the decomposition and composition of the simulators.

11.5.1 Decomposition and Composition of the Palladio Component Model

One way to address the aforementioned design smells in the PCM is the application of techniques for decomposition and composition as known from object-oriented design to metamodels in combination with a reference architecture to structure metamodels and support the decomposition and purpose-oriented composition of metamodels for quality modelling and analysis [HSR19].

Many commonalities in object-oriented design and metamodel design exist from a structural point of view. Both, object-oriented design and metamodel design, specify classes and their attributes, package structures as well as dependencies between classes, may it be for example association or inheritance [Str19]. Encouraged by these commonalities, we transferred established concepts from object-oriented design, such as decomposition and composition, acyclic dependencies, dependency inversion, extension, and layering to metamodels [HSR19].

Also transferring the idea of a reference architecture to metamodels seems reasonable. In our work, we focus on metamodels for quality modelling and analysis of software-intensive systems in different domains like information systems, production automation, and automotive. When comparing metamodels for modelling and analysing different quality properties in these domains, substantial parts of the metamodels exhibit quite similar language features [HSR19].

A Layered Reference Architecture for Metamodels

In [HSR19], we proposed a layered reference architecture for metamodels for quality modelling and analysis of software-intensive systems to address shortcomings in the evolution of metamodels. The reference architecture leverages reoccurring patterns in various domains. We studied different metamodels used for quality modelling and analysis in various domains as well as their extensions and identified that these metamodels reflect in most cases language features from distinct categories—structure, behaviour, and quality. This observation led to the separation

of parts of a metamodel into different layers in the reference architecture. A layer is a set of metamodel components. A metamodel component is defined as a container of packages and classifiers that has explicit dependencies. Metamodel components can be extended by lower-level layers and reused in different metamodels [HSR19]. The layers of the reference architecture are dedicated to structure/behaviour, quality, and the corresponding analysis. We further separated the structure/behaviour layer into paradigm and domain to distinguish domain-spanning fundamental concepts from domain-specific concepts. Metamodel components are assigned to one specific layer depending on the features they offer to the language. Based on concepts taken from object-oriented design and detailed application guidelines of these concepts described in [HSR19], the reference architecture supports (a) the top-level decomposition of metamodels for quality modelling and analysis into the four layers—paradigm, domain, quality, and analysis, (b) the decomposition of partial metamodels assigned to one of the layers into reusable metamodel components, and (c) the reuse of metamodel components in different contexts and thus the purpose-oriented composition of metamodels.

In the following, we give more detailed definitions of the single layers of the reference architecture taken from [HSR19] before we describe the application of the reference architecture to the PCM. The *paradigm* (π) layer is the most basic and most abstract layer. The foundation of the language is defined on the π layer by specifying language features for reoccurring patterns of structure and behaviour but without dynamic semantics. Furthermore, π does not carry any domain-specific semantics as this layer is not intended to be used without any additional layer. The *domain* (Δ) layer builds upon the π layer and assigns domain-specific semantics to the abstract first-class language features of π. Therefore, the Δ layer builds upon structural as well as on behavioural language features of π. The *quality* (Ω) layer defines quality-related properties of language features located on previous layers. The analysis layer (Σ) builds upon the previous layers and specifies language features used by analyses. Σ comprises language features to define configuration data, runtime state, output data, and input data that do not belong to Δ language features.

The result of the application of the reference architecture to the PCM is depicted as an excerpt in Fig. 11.2. The figure and the explaining text come from [HSR19]. We split the largest metamodel component of the original PCM into 23 smaller components to separate features properly. The other four metamodel components of the original PCM were already sufficiently modular. The number of classes in the decomposed PCM grew from 203 to 229. This is because during refactoring we split classes and created new containers for extensions. The number of references in the decomposed PCM reduced from 198 to 174. This is because we removed or remodelled redundant dependencies that violated the reference architecture. The decomposed PCM populates the layers π, Δ, and Ω. The Σ layer is populated by analysis-specific extensions of the PCM.

The most important metamodel components of the decomposed PCM are depicted in Fig. 11.2. On the π layer, these are *repository*, *composition*, *control flow*, and *annotations*. *Repository* specifies abstract components, interfaces, and

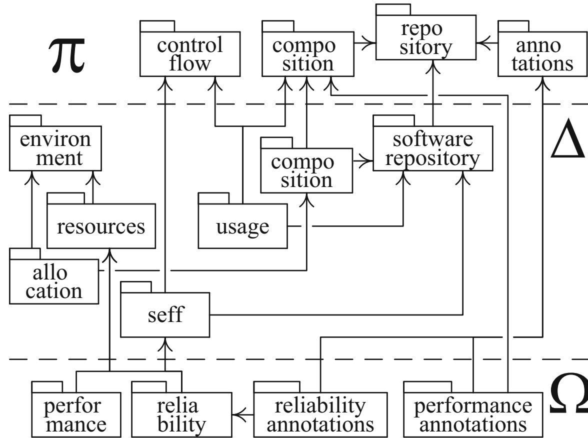

Fig. 11.2 Excerpt of the decomposed PCM. ©2021 IEEE. Reprinted, with permission, from [HSR19]

roles. *Composition* introduces component composition and therefore extends the *repository* metamodel component. *Control flow* defines a structure similar to activity diagrams. *Annotations* contains quality-independent annotations as an extension of the *repository* metamodel component. The domain (Δ) layer comprises the *composition* and *software repository* metamodel components, which extend their counterpart from the π layer and carry additional domain-specific content. This means the specialisation of abstract components to software components is happening in these two metamodel components. The *environment* metamodel component specifies execution containers and network links between the execution containers. The *resources* metamodel component extends the *environment* metamodel component to add hardware resource specifications to the execution containers and the network links. The *allocation* metamodel component enables software component instances (from the *composition* metamodel component) to be deployed on the execution containers of the *environment* metamodel component. The *usage* metamodel component specifies system usage profiles, which can be applied to interfaces from the *software repository* metamodel component. It therefore reuses the *control flow* metamodel component of π, which is also reused by the *seff* metamodel component to define the control flow between component-internal actions and component-external services. The quality (Ω) layer comprises the *performance* metamodel component, which extends the *resources* extension of the *environment* metamodel component by performance-relevant properties. It also extends the *seff* metamodel component by resource demand specifications. The dependencies of the *reliability* metamodel component are analogous. Finally, there are two metamodel components that enable the annotation of both quality properties in a component-based architecture by reusing the abstract definition of *annotations* on the π layer.

For the purpose-oriented composition of metamodel components, different metamodel extension mechanisms are proposed in [HSR19], which serve as composition operators. The concept of extension is well known and established in object-oriented design, for example, by means of stereotyping. However, EMOF on which the PCM is based does not support an extend relation. For this reason, several ways of how to enable the creation of extensions with EMF's Ecore are identified and discussed in [HSR19]. These include EMF Profiles [Lan+11] that enable the support for stereotypes, different kinds of plain referencing in combination with the introduction of new containers or inheritance relations, and cross-module inheritance.

The interested reader may refer to Chap. 2 for foundations of model and analysis composition, to Chap. 4 for general discussion on compositional semantics and to Chap. 9 of this book [Hei+21] for its application in the context of GTSMorpher.

Refactoring Metamodel Design Smells

Based on metamodel decomposition techniques and the reference architecture, detailed guidelines for metamodel refactoring have been proposed in [HSR19]. These guidelines comprise refactorings on metaclass level as well as on metamodel component level. In the following, we describe how the design smells in the PCM discussed in Sect. 11.3 can be refactored.

The Language Feature Scattering smell can be refactored by decomposing metamodel packages and locating all classifiers that implement a specific language feature into a single metamodel package [HSR19]. Classifiers within a metamodel package that are more closely related should be placed into their own subpackage. Details on refactoring metaclass and packages are described in [Str19]. The reference architecture proposed in this chapter helps to distinguish classifiers of fundamental (abstract) language features (π), domain-specific features (Δ), quality-specific features (Ω), and features specific to analyses (Σ). In the decomposed PCM (see Fig. 11.2), all classes for representing the resources feature, for example, have been located in a metamodel package on the Δ layer called resources. All classes for implementing the performance feature and reliability feature, respectively, have been placed into the metamodel packages performance and reliability on the Ω layer and extend the resource-specific classes on the Δ layer.

For resolving the Package Blob smell, the metamodel package must be split so that each package only contains classifiers of a single language feature [HSR19]. Subpackaging may be applied to further decompose metamodel packages. Details on refactoring metaclass and packages are described in [Str19]. The refactored metamodel packages may be located on different layers of the reference architecture depending on the purpose they satisfy. In the decomposed PCM (see Fig. 11.2), the repository package is split to distinguish the various features implemented in this package. A package repository is located on the π layer to implement a domain-independent repository feature that is further subdivided into packages to implement features for component composition and annotation. On the Δ layer, the repository package is extended by domain-specific classes to represent software components.

Representing components of other domains, like electrics/electronics or mechanics, as extension of the domain-independent repository feature is possible on the Δ layer but out of the scope of the original PCM. Extensions to represent performance and reliability are located on the Ω layer.

The Metamodel Monolith smell can be refactored by splitting the metamodel files according to their language features following the metamodel decomposition techniques proposed in this chapter. Each metamodel file then contains a single metamodel component. Based on the language features they provide, the metamodel components can be composed to form a language specific to a given purpose.

11.5.2 Decomposition and Composition of the Simulators

The layered reference architecture for quality modelling and analysis introduced in the previous section cannot only be applied to metamodels but also to simulators working on instances of the metamodels. Simulators may be decomposed into simulator components along the features they provide. We define a simulator component as a container of packages and classes that has explicit interfaces to other simulator components. The individual simulator components may be composed to satisfy a specific purpose for which a system is to be analysed. This requires composition operators for simulators.

Three forms of composition of analyses in general—model composition (white-box composition), result composition (black-box composition), and analysis composition (grey-box composition)—have been introduced in Chap. 4 of this book [Hei+21]. In this chapter, we give concrete examples of how to implement these forms of composition by discussing specific composition operators for simulators in the context of Palladio.

First attempts at composition operators for simulators in the context of Palladio have been described in [Hei+17]. These composition operators are:

- Composition by result exchange between isolated simulators
- Composition by co-simulation
- Composition by transformation into a joint formalism
- Composition by extension of one simulator by another

Composition by result exchange between isolated simulators conforms to the form result composition (black-box composition) in Chap. 4 of this book [Hei+21]. It is the most simple way of simulator composition. This way of composition can only be applied if one simulator requires the results of another simulator, but there is no interaction between the simulators required during simulation. Both simulators are executed in isolation, and information is exchanged ex-post by inserting the results of one simulator as input into another simulator.

Composition by co-simulation conforms to the form analysis composition (grey-box composition) in Chap. 4 of this book [Hei+21]. It enables information exchange during simulation. Simulators are interlinked in order to exchange information dur-

ing simulation. Co-simulation commonly requires additional efforts, for example, a coordinator for time management, model synchronisation, and connectivity in order to enable coherent simulation.

Composition by transformation into a joint formalism conforms to the form model composition (white-box composition) in Chap. 4 of this book [Hei+21]. It uses model transformations for creating a homogeneous simulation model. A characteristic of this approach is that a single formalism model is used as input to the simulation. Commonly, general-purpose simulation formalisms like Petri nets or queuing networks are used as the target formalism. This way of simulator composition can only be applied if there is a joint formalism to integrate the models of all the simulators (or if such an integrated formalism can be constructed). For Palladio, for instance, transformations to layered queuing networks [KR08] and queuing coloured Petri nets [MKK11] have been developed so far.

Composition by extension is another way to implement the form model composition (white-box composition) in Chap. 4 of this book [Hei+21]. This way of simulator composition is about extending the metamodel and simulation routines of one simulator by the metamodel and simulation routines of another simulator to form an integrated and unified simulator. Composition by extension is applicable if all the simulators build upon the same (or compatible) modelling paradigm and simulation formalism.

In the following, we give examples of the application of the composition operators in the context of Palladio.

IntBIIS

The approach *Integrated Business IT Impact Simulation* (IntBIIS) [Hei+17] is an example of composition by extension. IntBIIS is a composition of Palladio's simulator EventSim [MH11] and a business process simulator by extending the metamodel and simulation routines of Palladio by entities, scheduling policies, and simulation routines specific to business processes. Applying composition by extension in IntBIIS is possible as both simulators, Palladio's EventSim and the business process simulator, adhere to the same modelling paradigm and simulation formalism. IntBIIS extends the usage specification of the PCM by business process constructs. Both, the usage model and the business process model, rely on an activity diagram like modelling paradigm. They specify a certain workload to be processed by resources in the form of sequences of actions (possibly hierarchically nested) and intensity of action execution. Both, Palladio's EventSim and the business process simulator, build upon queuing theory concepts to simulate resources processing aforementioned workload.

An overview of the composed simulators of IntBIIS is given in Fig. 11.3. Blue elements with a stickman symbol indicate modelling constructs and simulation routines introduced as an extension of the original EventSim simulator. The remaining grey elements are those of the original EventSim simulator. A run of the composed simulators starts at the topmost layer with simulating workloads that originate

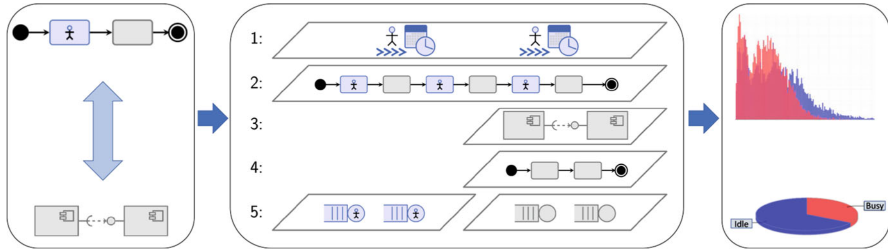

Fig. 11.3 Composition by extension in IntBIIS, after [Hei+17]

from the business process model. For each workload specification, a workload generator spawns a new business process instance in the simulation whenever a certain inter-arrival time has been passed [Hei+17]. A business process instance is the representation of a single enactment of the business process model [Hei+17]. Each business process instance is then simulated individually by traversing the corresponding sequence of actions specified in the business process model (layer 2). When the traversal procedure arrives at an action, basically two cases can be distinguished [Hei+17]: (i) the simulation encounters an actor step or (ii) it encounters a system step (i.e., system entry call).

In case (i), a suitable resource that represents a human actor is requested (layer 5, left) in simulation. If the selected actor is already busy, the actor step is enqueued in its waiting queue. This induces a waiting period not only for the actor step but also for the enclosing business process instance. Based on these concepts taken from queuing theory, we can simulate execution times of actor steps and the entire business process instance as well as utilisation of actor resources depending on a given workload. Simulation results can be visualised in the form of histograms and pie charts for engineers.

In case (ii), resource demands are not issued directly by the business process instance but emerge as the system request propagates through components (layer 3), their service effect specifications (layer 4), down to hardware resources (layer 5, right) [Hei+17]. Similar to actor resources, hardware resources may be busy and therefore block a request. This causes waiting time for the system step and the enclosing business process instance. Based on these concepts taken from queuing theory, we can simulate execution times of system steps and the entire business process instance as well as utilisation of hardware resources depending on a given workload. Simulation results can be visualised again in the form of histograms and pie charts for engineers.

PCA

Composition by result exchange between isolated simulators has been applied in the Palladio context to use Palladio simulator results in other analysis tools as a basis to reason about additional quality properties. The *Power Consumption Analyzer*

(PCA) [Sti18] uses the results of Palladio's simulator SimuLizar to forecast power consumption of software systems. The Power Consumption metamodel proposed in [Sti18] is used to specify consumption characteristics of servers, their components, and connected power distribution infrastructure. The performance simulation results of SimuLizar—utilisation of CPU and hard disk resources of servers—combined with the characteristics specified in instances of the Power Consumption metamodel are used to reason about the power consumption of software systems on architecture level. The analysis in [Sti18] supports the architecture-level examination of both, static and self-adaptive software systems. As shown in Fig. 11.4, the PCA uses measurements from the Palladio Runtime Measurement Model that have been produced by SimuLizar and calculates the power consumption based on its Power State Model. A Power State Model is a stateful power model in the form of a state machine that describes, for example, which servers are in on or off state. The results of the PCA are then accessible in the Palladio Runtime Measurement Model and can be used to trigger self-adaptations in SimuLizar.

OMPCM

An example of composition by co-simulation in the Palladio context is the OMPCM [HMR13] approach. Modelling and simulation of network communication are limited in Palladio. This weakens not only the prediction accuracy for network-intensive systems [KBH07, Ver+07] but also misses the opportunity to simulate different network configurations and topologies before implementing them. Extensive network communication arises especially within distributed systems, where software components deployed on different hardware nodes work together towards a common goal. OMPCM integrates the OMNeT++-based network simulation

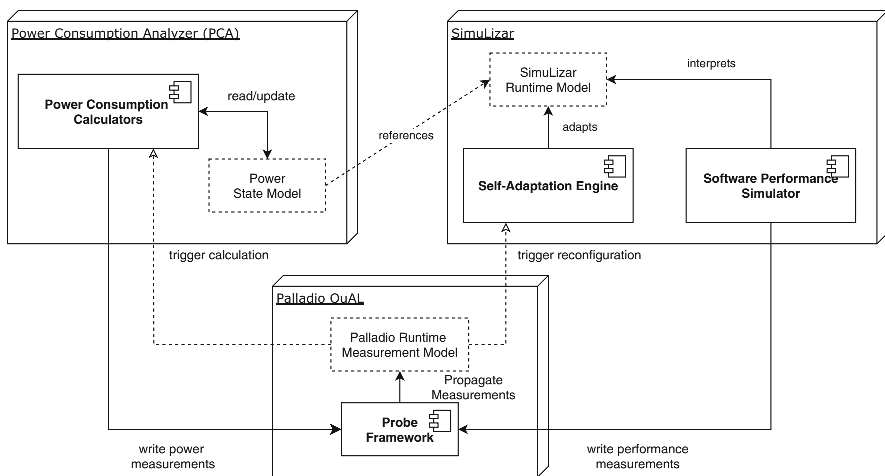

Fig. 11.4 Composition by result exchange in PCA, after [Sti18]

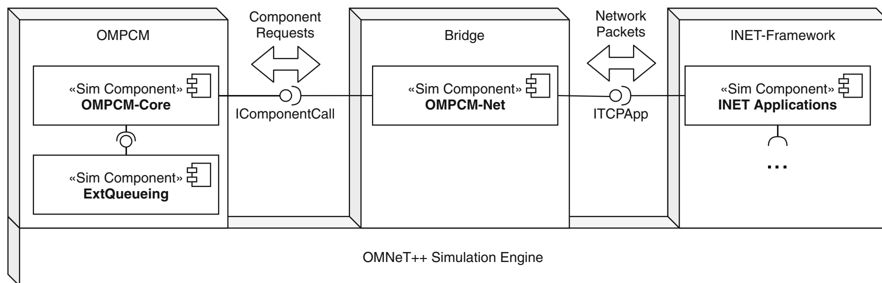

Fig. 11.5 Composition by co-simulation in OMPCM

framework INET with the Palladio architecture-level software performance prediction implemented in the OMPCM-Core and ExtQueuing components to enrich Palladio by more detailed network simulation. OMPCM applies composition by co-simulation by having a dedicated bridge (OMPCM-Net) to manage the translation of events between the OMPCM and the INET simulators. As shown in Fig. 11.5, the OMPCM-Net bridge component accepts events corresponding to the IComponentCall interface. The IComponentCall interface describes the sending and reception of requests and responses on software component level. The bridge component then translates the component-level events to network-level events by resolving remote software components to network nodes and requests/responses to TCP transfers. Implementing the ITCPApp interface of the INET-Framework, the bridge component then sends and receives network-level events to and from the network simulation.

Refactoring Simulator Design Smells

Next, we discuss how aforementioned design smells in the simulators can be resolved.

The Amateur Science smell can be refactored by the proposed simulator composition techniques. The individual simulator components can be developed independently by domain experts for the specific simulator components. The simulator components can then be composed to satisfy a certain analysis goal. In the Palladio context, for example, composition of OMPCM and the OMNeT++-based network simulator INET by co-simulation [HMR13] allows for including detailed network simulation in Palladio, while the INET network simulator has been developed by domain experts independent of Palladio.

The smell Analysis Paralysis can be addressed by decomposing unnecessarily complex or detailed metamodels and simulators and composing the metamodel components and simulator components, respectively, as described in this chapter on an appropriate level of complexity or detail.

The Everything but the Kitchen Sink smell can be addressed by decomposing metamodels and simulators by distinguishing relevant from irrelevant features

and composing only relevant metamodel components and simulator components, respectively, as described in this chapter.

The Living Flatland smell can be refactored by enabling the replacement and/or composition of simulator components to consider other and/or additional levels of abstraction in simulation. In Palladio, the simulator component responsible for processor scheduling needs to be replaced so that a new simulator component can provide the scheduling policies needed. Alternatively, the composition of additional simulator components that provide the needed scheduling policies with the existing simulator components is a solution in Palladio.

The Underdefined Semantics smell can be addressed by clearly defining the semantics of metamodel and simulator components and by purpose-oriented composition of only semantically compatible simulator components to satisfy a certain analysis goal. Compositionality of analyses and specific conditions of composition are discussed in Chap. 4 of this book [Hei+21].

The Excessive Events/Event Flooding smell can be refactored by avoiding unnecessary communication via events and using as little events as possible. This can be achieved by aggregating events that happen at the same time instead of sending each event individually. In addition, only time-dependent communication should happen via events, and the temporal resolution can be communicated before starting the simulation to reduce time synchronisation effort via events. Note, the Excessive Events/Event Flooding smell can be caused by simulator composition as each simulator component may have its own event management that needs to be synchronised with others. This synchronisation causes large event communication overhead. This communication overhead needs to be considered in simulator design and avoided as described before or by using a centralised event management like in [IEE10].

The Simulator Feature Scattering, Simulator Component Blob, and Simulator Monolith smells can be refactored by decomposing the simulator into simulator components along the features provided by the simulator following the decomposition techniques proposed in this chapter. The composition techniques described in this chapter enable the interaction between the different simulator components. Adequate decomposition of simulators allows for exchanging and purpose-oriented composition of simulator components.

The Global State Object and God Parameter smells can be resolved by decomposing the simulator into simulator components and following object-oriented design principles [Mar00] to reduce coupling between the simulator components.

The Intrusive Extension smell can be resolved by adequately decomposing the simulator into simulator components along the features it provides. This will lead to an extraction of intrusive simulator extensions into separate simulator components.

11.6 Conclusion and Outlook

This chapter gave insights into Palladio as a case study for evolution of a historically-grown approach to model-based analysis. We provided an overview of the Palladio approach and the associated tooling. We reported about design smells in the metamodel and simulators caused by evolution and growth over several years. We discussed how techniques for decomposition and purpose-oriented composition can help refactoring the metamodel and simulators to avoid these design smells and thus ease the evolution of the Palladio approach in the future.

Techniques for decomposition and composition of modelling languages and analysis tools need to be further investigated in the future to make the concepts discussed in this chapter applicable in a more general way. The application of the decomposition and composition techniques for grammar-based modelling languages would be interesting to investigate in the future. While in this chapter the techniques for decomposition and composition have been discussed in the light of the Palladio approach, we expect these techniques are independent of quality modelling and analysis and can be applied to modelling languages and analysis tools in general. Further, the dependencies between modelling languages and analysis tools on the level of their features and components need to be investigated in the future. Tool support is required for visualising dependencies between modelling languages and analysis tools on feature and component level to simplify working with and configuring large modelling languages and analysis tools.

References

[Bec08] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. Universitätsverlag Karlsruhe, 2008. https://publikationen.bibliothek.kit.edu/1000009095.

[BLB13] Matthias Becker, Markus Luckey, and Steffen Becker. “Performance Analysis of Self-adaptive Systems for Requirements Validation at Design-time”. In: *9th International ACM Sigsoft Conference on Quality of Software Architectures*. 2013, pp. 43–52. https://doi.org/10.1145/2465478.2465489.

[Bro+12] Franz Brosch, Heiko Koziolek, Barbora Buhnova, and Ralf Reussner. “Architecture-Based Reliability Prediction with the Palladio Component Model”. In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1319–1339. https://doi.org/10.1109/TSE.2011.94.

[Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[Gro13] Henning Groenda. *Certifying Software Component Performance Specifications*. Vol. 11. KIT Scientific Publishing, Karlsruhe, 2013. https://doi.org/10.5445/KSP/1000036063.

[GS13] Samarthyam Ganesh and Tushar Sharma. “Towards a Principle-based Classification of Structural Design Smells”. In: *Journal of Object Technology* 12 (2013). https://doi.org/10.5381/jot.2013.12.2.a1.

[Hap08] Jens Happe. *Predicting Software Performance in Symmetric Multi-Core and Multiprocessor Environments*. University of Oldenburg, Germany, 2008. http://oops.uni-oldenburg.de/827/1/happre08.pdf.

[Hei+17] Robert Heinrich, Philipp Merkle, Jörg Henss, and Barbara Paech. "Integrating business process simulation and information system simulation for performance prediction". In: *Software & Systems Modeling* 16.1 (2017), pp. 257–277. https://doi.org/10.1007/s10270-015-0457-1.

[Hei+18] Robert Heinrich, Dominik Werle, Heiko Klare, Ralf Reussner, Max Kramer, Steffen Becker, Jens Happe, Heiko Koziolek, and Klaus Krogmann. "The Palladio-Bench for Modeling and Simulating Software Architectures". In: *40th International Conference on Software Engineering: Companion Proceedings*. 2018, pp. 37–40.

[Hei+21] Robert Heinrich, Francisco Durán, Carolyn L. Talcott, and Steffen Zschaler (eds.) *Composing Model-Based Analysis Tools*. Springer, 2021. https://doi.org/10.1007/978-3-030-81915-6.

[HMR13] Jörg Henss, Philipp Merkle, and Ralf Reussner. "The OMPCM Simulator for Model-Based Software Performance Prediction: Poster Abstract". In: *6th International ICST Conference on Simulation Tools and Techniques*. 2013, pp. 354–357.

[HSR19] Robert Heinrich, Misha Strittmatter, and Ralf Reussner. "A Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis". In: *IEEE Transactions on Software Engineering* 47.4 (2019), pp. 775–800. https://doi.org/10.1109/TSE.2019.2903797.

[IEE10] IEEE. 1516-2010—IEEE *Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)*. Tech. rep. IEEE, 2010, pp. 1–38. https://doi.org/10.1109/IEEESTD.2010.5553440.

[KBH07] Heiko Koziolek, Steffen Becker, and Jens Happe. "Predicting the Performance of Component-Based Software Architectures with Different Usage Profiles". In: *Third International Conference on Quality of Software Architectures*. 2007, pp. 145–163. https://doi.org/10.1007/978-3-540-77619-2_9.

[KR08] Heiko Koziolek and Ralf Reussner. "A Model Transformation from the Palladio Component Model to Layered Queueing Networks". In: *SPEC International Performance Evaluation Workshop*. 2008, pp. 58–78. https://doi.org/10.1007/978-3-540-69814-2_6.

[Lan+11] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. "EMF profiles: A lightweight extension approach for EMF models". In: *Journal of Object Technology* 11 (2011), p. 8. https://doi.org/10.5381/jot.2012.11.1.a8.

[Leh14] Sebastian Lehrig. "Applying Architectural Templates for Design-Time Scalability and Elasticity Analyses of SaaS Applications". In: *2nd International Workshop on Hot Topics in Cloud Service Scalability*. 2014, 2:1–2:8.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. https://doi.org/10.1007/3-540-39538-5.

[Mar+10] Anne Martens, Heiko Koziolek, Steffen Becker, and Ralf Reussner. "Automatically Improve Software Architecture Models for Performance, Reliability, and Cost Using Evolutionary Algorithms". In: *First Joint WOSP/SIPEW International Conference on Performance Engineering*. 2010. https://doi.org/10.1145/1712605.1712624.

[Mar00] Robert C. Martin. *Design Principles and Design Patterns*. Vol. 1. Prentice Hall, 2000.

[MH11] Philipp Merkle and Jörg Henss. "EventSim—An Event-driven Palladio Software Architecture Simulator". In: *Palladio Days 2011*. 2011, pp. 15–22. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025188.

[MKK11] Philipp Meier, Samuel Kounev, and Heiko Koziolek. "Automated Transformation of Component-based Software Architecture Models to Queueing Petri Nets". In: *19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2011. https://doi.org/10.1109/MASCOTS.2011.23.

[Reu+16] Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures—The Palladio Approach*. MIT Press, 2016.

[Ros+15] Kiana Rostami, Johannes Stammel, Robert Heinrich, and Ralf Reussner. “Architecture-based Assessment and Planning of Change Requests”. In: *11th International ACM SIGSOFT Conference on Quality of Software Architectures*. 2015, pp. 21–30. https://sdqweb.ipd.kit.edu/publications/pdfs/rostami2015a.pdf.

[SHR18] Misha Strittmatter, Robert Heinrich, and Ralf Reussner. *Supplementary Material for the Evaluation of the Layered Reference Architecture for Metamodels to Tailor Quality Modeling and Analysis*. Tech. rep. 11. Karlsruher Institut für Technologie (KIT), 2018. 42 pp. https://doi.org/10.5445/IR/1000089243.

[SHR19] Stephan Seifermann, Robert Heinrich, and Ralf H. Reussner. “Data-Driven Software Architecture for Analyzing Confidentiality”. In: *IEEE International Conference on Software Architecture*. 2019, pp. 1–10.

[Ste12] Susan Stepney. “A pattern language for scientific simulations”. In: *Workshop on complex systems modelling and simulation*. 2012, pp. 77–103.

[Sti18] Christian Stier. *Adaptation-Aware Architecture Modeling and Analysis of Energy Efficiency for Software Systems*. 2018, p. 262. https://doi.org/10.5445/IR/1000083402.

[Str+16] Misha Strittmatter, Georg Hinkel, Michael Langhammer, Reiner Jung, and Robert Heinrich. “Challenges in the Evolution of Metamodels: Smells and Anti-Patterns of a Historically-Grown Metamodel”. In: *10th International Workshop on Models and Evolution*. 2016.

[Str19] Misha Strittmatter. *A Reference Structure for Modular Metamodels of Quality- Describing Domain-Specific Modeling Languages*. Universitätsverlag Karlsruhe, 2019. https://publikationen.bibliothek.kit.edu/1000098906.

[TH16] Emre Taspolatoglu and Robert Heinrich. “Context-based Architectural Security Analysis”. In: *13th Working IEEE/IFIP Conference on Software Architecture*. 2016, pp. 281–282.

[Ver+07] Tom Verdickt, Bart Dhoedt, Filip De Turck, and Piet Demeester. “Hybrid performance modeling approach for network intensive distributed software”. In: *6th International Workshop on Software and Performance*. 2007, pp. 189–200. https://doi.org/10.1145/1216993.1217026.