Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 107

Introducing Performance Awareness in an Integrated Specification Environment

Fabian Keller

Course of Study:	Softwaretechnik
Examiner:	Dr. André van Hoorn (Prof.-Vertr.)
Supervisor:	Dr. André van Hoorn (Prof.-Vertr.), Dr. Markus Völter (itemis AG, Stuttgart), Dr. Klaus Birken (itemis AG, Stuttgart)
Commenced:	May 17, 2016
Completed:	November 30, 2016
CR-Classification:	D.2.6, C.4

Abstract

With an increase in software complexity and modularization to create large software systems and software product lines it is increasingly difficult to ensure all requirements are met by the built system. Performance requirements are an important concern to software systems and research has developed approaches being capable of predicting software performance from annotated software architecture descriptions, such as the Palladio tool suite. However, the tooling when moving between specification, implementation and verification phase has a gap as the tools are commonly not linked, leading to inconsistencies and ambiguities in the produced artifacts. This thesis introduces performance awareness into the Integrated Specification Environment for the Specification of Technical Software Systems ($\mathrm{IET_S}^3$), which is a specification environment aiming to close the tooling gap between the different lifecycle phases. Performance awareness is introduced by integrating existing approaches for software performance prediction from the Palladio tool suite and extending them to cope with variability-aware system models for software product lines. The thesis includes an experimental evaluation showing that the developed approach is able to provide performance predictions to users of the specification environment within 2000 ms for systems of up to 20 components and within 8000 ms for systems of up to 30 components.

Kurzfassung

Mit zunehmender Software-Komplexität und Modularisierung zur Entwicklung großer Softwaresysteme und Software-Produktlinien ist es zunehmend schwierig, alle Anforderungen des eingebauten Systems zu erfüllen. Performanz ist eine wichtige Anforderung für Software-Systeme und aktuelle Forschungsarbeiten haben Ansätze entwickelt, die in der Lage sind, Software-Performanz von annotierten Software-Architekturen vorherzusagen, wie beispielswiese die Palladio Tool Suite. Jedoch hat beim Wechseln zwischen Spezifikations-, Implementierungs- und Verifikationsphase die bestehende Toolchain eine Lücke, da die eingesetzten Werkzeuge häufig nicht miteinander verknüpft sind. Dies führt zu Inkonsistenzen und Unklarheiten in den erzeugten Artefakten. Diese Arbeit führt Performanz-Bewusstsein in die Integrated Specification Environment for the Specification of Technical Software Systems ($\text{IET}_S{}^3$) ein - eine Spezifikationsumgebung, die die Werkzeuglücke zwischen den verschiedenen Phasen des Software-Lebenszyklus zu schließen versucht. Das Bewusstsein wird durch die Integration bestehender Ansätze zur Performanz-Vorhersage aus der Palladio Tool Suite hergestellt und um die Analyse von Produktlinien erweitert. Die experimentelle Evaluierung der Arbeit zeigt, dass der entwickelte Ansatz in der Lage ist, innerhalb von 2000 ms Systeme bestehend aus bis zu 20 Komponenten, und innerhalb von 8000 ms Systeme bestehend aus bis zu 30 Komponenten, zu analysieren.

Acknowledgment

I would like to express my most humble and sincere gratitude to...

... The supervisor of this thesis, **Dr. André van Hoorn** for the freedom to work on my own ideas and guidance whenever I felt the need for helpful discussions. His guidance in the numerous discussions and conversations helped me a lot for the realization and implementation of innovative ideas and to write this thesis.

... My advisor **Dr. Markus Völter** for his continuous support of my thesis. He has helped me shape the vision for this thesis and introduced me to the project this thesis is built on. His efforts in teaching me how to engineer domain-specific languages have had a profound impact on the realization of this thesis.

... My advisor **Dr. Klaus Birken** for the helpful discussions and guidance on the implementation of this thesis. He always had an open ear to discuss the ideas I had and decisions I made to implement the approach building on the existing project.

... All colleagues at itemis AG who have helped me with constructive discussions and by resolving technical difficulties. I would like to especially thank **Bernd Kolb**, **Tamás Szabó**, and **Alexander Rimer** for spending countless hours debugging classpath problems and other technical issues.

... My friends **Sabrina Kleiser**, **Sebastian Hesse**, **Tim Theil** and **Valentin Zickner** for proofreading this thesis and for the valuable comments and remarks.

... The members of the Palladio community, **Dr. Anne Koziolek** and **Prof. Dr. Steffen Becker**, for their help with understanding and using the Palladio tool suite.

Contents

List of Figures

List of Acronyms

API	Application programming interface
CBSE	Component-based software engineering
DSL	Domain-specific language
EMF	Eclipse modeling framework
IDE	Integrated development environment
$\text{IET}_S{}^3$	Integrated Specification Environment for the Specification of Technical Software Systems
LQN	Layered queueing network
MPS	JetBrains Meta Programming System
OCL	Object constraint language
PCM	Palladio component model
QoS	Quality of service
RDSEFF	Resource demanding service effect specification
SPL	Software product line
UML	Unified modeling language

Chapter 1

Introduction

1.1 Motivation

Software systems are steadily growing in size and are having increasingly complex interdependencies. Many software systems are built as software product lines where a specific software system is derived by configuring the features of the product line. Additionally, there is an increasing demand for reducing the time-to-market of software projects while improving the overall quality.

Tools have been built to help users tackle the complexity and maintain a high quality in the different lifecycle phases of the software development process. The lifecycle phases roughly boil down to: writing a correct and unambiguous specification, implementing the specification and finally, verifying whether the built software meets the specification. While the performance of a yet to be developed system is hard to assess, it inevitably is an important non-functional requirement to many applications that is often underspecified.

Software performance research has developed approaches to predict the performance before the systems are actually built (Koziolek, 2010). The predictions are based on the component architecture enriched with performance-specific annotations such as resource consumption and typical use cases (Becker et al., 2009). However, the existing tools for performance analyses are not integrated with other tools of the software lifecycle, thus, making it difficult to trace the fulfillment of requirements through the used tools. Moreover, as the tools are not linked inconsistencies and ambiguities may arise that stay unnoticed and lead to unforeseen breakdowns of the system in production.

Designing systems with performance in mind saves costs in the long run, as architectural changes are often expensive to carry out in an existing system. To be able to properly cope with software performance, recent research efforts focus on raising performance awareness, which captures the (real-time) availability of performance information in system development environments to inform and assist developers (Tůma, 2014). Existing approaches (e.g., (Horký et al., 2015)) neglect the system specification, particularly the specification phase and do not support software product lines (SPLs).

1.2 Main Goal of the Thesis and Research Questions

The main goal of this thesis is to introduce performance awareness support into IET$_S$^{3}, an "Integrated Specification Environment for the Specification of Technical Software-systems". IET$_S$^{3} provides a specification environment addressing the tooling gap between the software development lifecycle phases introducing various languages to express requirements, software product lines and component-based software systems. Based on the main goal, this thesis answers the following research questions:

Research Question 1: How can various model-based performance prediction approaches be abstracted to a uniform interface to leverage different analysis result characteristics?

To answer the research question, a framework to integrate different performance prediction approaches with a uniform analysis and results interface will be developed during this thesis. The framework should be capable of integrating performance predictions provided by the Palladio (Becker et al., 2009) tooling environment for model-based performance prediction as well as the existing Simbench performance analysis contained in IET$_S$^{3} (Birken et al., 2010). With such a framework in place, users are able to either choose an approach that yields fast results or leverage an approach that offers more precise and detailed results.

Research Question 2: How can model-based performance predictions be integrated in the IET$_S$^{3} specification environment to provide feedback to an architect while designing the software system?

To answer the research question, the performance analysis approaches are integrated into the IET$_S$^{3} platform, providing a user interface to assess the software performance of the designed system in real-time. To raise performance awareness, the performance analysis results should be integrated into the existing languages in IET$_S$^{3} by adding performance-related annotations.

Research Question 3: How can model-based performance predictions be used to provide feedback to an architect while designing software product lines?

As IET$_S$^{3} is capable of expressing software product lines, a further goal of this thesis is to leverage the performance analyses approaches to analyze variability-aware software systems. The variability analysis results should again be visualized at appropriate locations of the existing languages and allow the user to interact with the results to answer relevant questions.

Research Question 4: Is the approach capable of providing real-time performance awareness?

As the approach is integrated into an IDE-like editor for software specifications, the performance approaches execution time needs to be acceptably low. Ideally, the approach should be able to provide a performance prediction result from triggering the analysis to rendering the result in the user interface within ten seconds.

1.3 Summary of Contributions

The main contributions of this thesis are:

1. **Performance analysis abstraction**: An open-source framework for model-agnostic software performance analysis that fosters the integration of various performance analysis approaches and provides a common interface for defining performance analysis results.
2. **Performance awareness**: The approach developed in this thesis introduces performance awareness into the Integrated Specification Environment for the Specification of Technical Software Systems ($\text{IET}_\text{S}{}^3$) by providing real-time performance analyses to the users of the system. Hence, the users have the ability to automatically evaluate whether the designed system will meet the performance requirements from within the specification environment.
3. **Variability-aware performance analysis**: Research on performance analyses for variability-aware systems is in an early stage. The approach developed in this thesis adds the ability to define performance analyses for variability-aware systems to the $\text{IET}_\text{S}{}^3$ environment and defines user interaction concepts to help architects meet all performance requirements for a software product line.

1.4 Outline

This thesis is organized as follows:

Chapter 2 introduces the foundations and sets the context for this thesis. Chapter 2.1 introduces component-based software engineering which is required by the software performance prediction approaches in Chapter 2.2. Chapter 2.3 describes the $\text{IET}_\text{S}{}^3$ project, in which the presented performance prediction approach is integrated in the scope of this thesis. The vision for software performance engineering is presented in Chapter 2.5 and Chapter 2.6 surveys existing literature related to the approach developed in this thesis.

Chapter 3 describes the performance analysis framework developed during this thesis to integrate different performance analysis approaches. Chapter 3.1 explains the abstraction to conduct a single performance analysis for a system. The structure of the performance results are detailed in Chapter 3.2. Chapter 3.3 extends the analysis abstraction with the ability to conduct performance analyses for variability-aware system models.

Chapter 4 is concerned with the integration of the performance prediction in the $\text{IET}_\text{S}{}^3$ environment. Chapter 4.1 gives an overview of the performance analysis for a single system and Chapter 4.2 for a variability-aware system. Chapter 4.3 describes the tooling infrastructure built to accomplish the integration of the performance prediction. The user interface and interaction concepts for the developed approach are then detailed in Chapter 4.4.

Chapter 5 explains the model-to-model transformation implemented to transform the $\text{IET}_\text{S}{}^3$ system model to the Palladio component model for software performance prediction. An overview and architectural design decisions are discussed in Chapter 5.1. Chapter 5.2 provides the details of the implemented baseline model-to-model transformation, before Chapter 5.3 focuses on the support of an enhanced concept of the $\text{IET}_\text{S}{}^3$ model during the transformation.

Chapter 6 evaluates the performance of the presented approach as the approach aims to provide immediate feedback in an interactive specification environment. Chapter 6.1 analyzes the performance of the approach by scaling the number of components in the system, while Chapter 6.2 analyzes the performance of the approach using real-world software architectures.

Chapter 7 concludes the thesis by summarizing and discussing the contributions of the thesis and the approach taken. The chapter expounds the limitations of the approach and highlights future work.

Chapter 2

Foundations and Research Context

Software performance is a non-functional requirement and an important concern to all software systems being built. Without adequate performance a software system is likely to be rejected by its users for the task it should perform. Yet, software performance is often neglected when specifying software systems, causing issues after efforts have been spent to actually build the software. While it is not important (and in general not reasonable) to precisely define all performance requirements upfront it is still very important to continuously keep the expected software performance in mind (Ho et al., 2006).

This chapter provides the foundations and research context of this thesis. Section 2.1 introduces component-based software engineering as paradigm, as the used software performance evaluation methods leverage component-based software systems. Section 2.2 explains the Palladio component model (PCM), which is used to evaluate the performance of component-based systems. One such evaluation method is the layered queuing network analysis.

Section 2.3 introduces the MPS language engineering platform on which $\text{IET}_\text{S}{}^3$ is built on and as well as the $\text{IET}_\text{S}{}^3$ project itself. In addition to that, the existing performance evaluation approach called Simbench is described. As $\text{IET}_\text{S}{}^3$ supports the modeling of software product lines Section 2.4 introduces basic notations and formalisms related to variability. Section 2.5 details the status quo and vision of declarative performance engineering before Section 2.6 relates the work to the state of the art research.

2.1 Component-based Software Engineering

Component-based software engineering (CBSE) is a discipline centered around engineering software systems by composing individual, reusable software components. In addition to a high reuseability of components, an additional advantage of CBSE is the ability to design for changeability. With fast-paced changes to requirements and market demands the ability to change software quickly is an important competitive advantage (Fricke et al., 2005). Software components naturally allow to compose systems with changed behaviors by using different or updated components, or even by connecting components differently to alter their composed behavior.

2.1.1 Software Components

The use of the term *component* or *software component* in this thesis adheres to the definition by Taylor et al. (2009): "A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context". Hence, a key concept of software components are interfaces the component provides or requires. While a component can only be accessed through an interface it provides, the component itself may only access other components through interfaces it requires. With the stringent separation between the component specification comprising its interfaces and a specific component realization implementing the interfaces, two software component realizations can be exchanged as long as they provide the same interface.

Component realizations are free to choose the platform, programming language or programming paradigm they are built with, as long as they comply with their interfaces. Generally, component realizations are treated as black-box as clients should not know the inner workings of a component to not rely on implementation-specific behavior. However, to analyze the performance behavior it is inevitable to know the inner workings of the component to some degree, as the resource usage of a component is often highly dependent on the arguments the component is called with. Hence, the approach in this thesis requires at least a grey-box view of the component which reveals only parts of the implementation, or a glass-box view which provides a read-only view of the realization (Koziolek, 2008).

2.1.2 Component-based Software Development Process

Building a software system based on components differs from classical software engineering development processes. Taylor et al. (2009) define the development as: "A component-based software development is the development based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch".

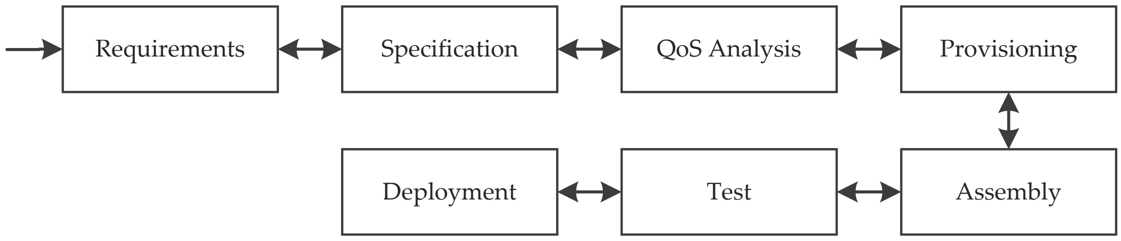


Figure 2.1: Simplified view of the component-based development process starting with the requirements phase. Originally defined by Cheesman et al. (2001) and augmented to include the QoS Analysis by Koziolek et al. (2006). The double-sided arrow indicates a change of activity in either direction.

The resulting system architecture emphasizes the component interactions as well as decisions to use or reuse a component.

Figure 2.1 shows the activities of the component-based development process as described by Cheesman et al. (2001) neglecting the flow of artifacts between the activities. The workflow is used by the Palladio tool suite (Koziolek, 2008) which powers the approach presented in this thesis. The workflow also applies to the $\text{IET}_S{}^3$ modeling environment and is thus important to this approach.

The *Requirements* activity elicits and documents the business requirements and use cases with the help of a domain expert. Based on the requirements, the component-based software architecture is designed in the *Specification* activity by a system architect. System architects may work with component developers to model new and reuse existing component specifications for the system architecture. Koziolek et al. (2006) added the *QoS Analysis* activity to explicitly account for the quality of service (QoS) analysis of the system architecture. The QoS analyst uses business requirements to enrich the system architecture with QoS-related measures and verifies whether the QoS requirements are fulfilled for a certain system deployment. The results of the analysis may in turn influence the system architecture as well as the deployment. The *Provisioning* activity comprises the “make-or-buy” decisions that need to be made for each individual component. During the *Assembly* activity the components are technically connected, which may involve configuring components, writing or bridging differing interfaces or integrating legacy components. The assembled application can then be tested in the *Test* activity, before being deployed in the *Deployment* activity.

2.2 Software Performance Prediction

One aspect of the QoS analysis activity in the component-based development process introduced in Section 2.1 is software performance. When designing the system architecture from the individual components it becomes increasingly important to determine whether the proposed architecture fulfills all requirements, ideally before any efforts are spent building the system. Research has developed methods to reason about the software performance of a system by specifying the performance behavior of its individual components and analyzing their interaction (Reussner et al., 2003).

Performance prediction leverages architectural models of a software system to predict its performance. Typically, the software system is in its design phase when performance prediction is useful. As performance prediction works on models of the system to be built, there is a risk that the predicted performance is not identical to the performance of the implemented system. To minimize the risk, experts build prototypes, proof-of-concepts, measure existing systems (e.g., by extracting typical workloads (van Hoorn et al., 2014)) and then integrate a number of refined assumptions into the models used for prediction. To predict the performance of a software system the following models are typically required (Becker et al., 2009):

- **Component Specifications**: Describes available components, their required and provided interfaces including supported operations as well as their expected performance demands.
- **Assembly Model**: The assembly model specifies component interactions and their connections to form the whole system. The assembly model specifies which component instances of a certain component are available to build the system.
- **Allocation Model**: The allocation model allocates the assembled components onto specific (hardware) resources that are capable of satisfying the performance demands of the components. The performance specification of the resources is also part of the allocation model.
- **Usage Model**: The usage model contains typical use-cases of the system for which the performance is relevant. The usage model is a reflection of the business QoS requirements and provides the entry point for the performance analysis.

Becker et al. (2009) introduce the Palladio component model (PCM) which is capable of modeling all of the above models. In the meantime, various analysis approaches for the PCM have been developed, including analytical analyses transforming the PCM to stochastic regular expressions or a queuing network model to predict the performance of the model.

The performance analyses used in this thesis are built on the Palladio component model, which is introduced in Section 2.2.1. The analytical PCM performance analysis leveraging layered queueing networks (LQNs) is then described in Section 2.2.2.

2.2.1 Palladio Component Model

Palladio (Becker et al., 2009) is a domain-specific language and corresponding tooling infrastructure for performance prediction of component-based software systems. The Palladio component model (PCM) is the modeling language used to model a system with its component specifications, assemblies, allocations and usage models. This section presents the different aspects contained in a PCM instance.

Repository

The PCM repository (Figure 2.2) stores component specifications (referred to as `RepositoryComponent`), interface specifications (referred to as `Interface`) and available data types (referred to as `DataType`). The component specification models which interfaces the component requires and which interfaces are provided by the component. For the sake of simplicity, the classes involved in establishing the provided and required interfaces of a component are not shown in the class diagram in Figure 2.2.

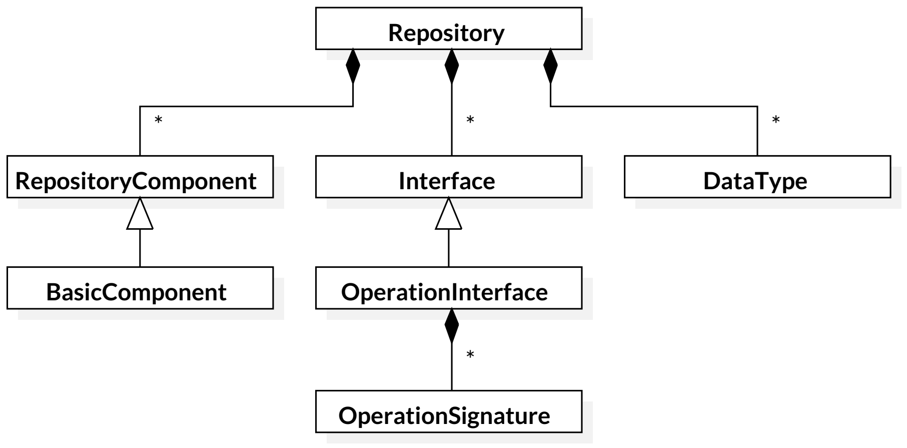


Figure 2.2: Important aspects of the PCM repository meta-model.

The approach developed in this thesis will only make use of the `BasicComponent` component type, which represents a simple component. Palladio features enhanced components such as composite components or subsystems, which are not relevant for the presented approach. The `OperationInterface` is an interface type representing a regular component interface owning a set of `OperationSignature` signatures that can be called. An operation signature may specify parameters and return types referencing the available data types of the repository.

Resource demanding service effect specification

Resource demanding service effect specifications (RDSEFFs) (Figure 2.3) are used to model the performance behavior of a component and are part of the repository model. A central part of the RDSEFF model is the `ResourceDemandingBehavior` that comprises a number of steps that define the resource demands.

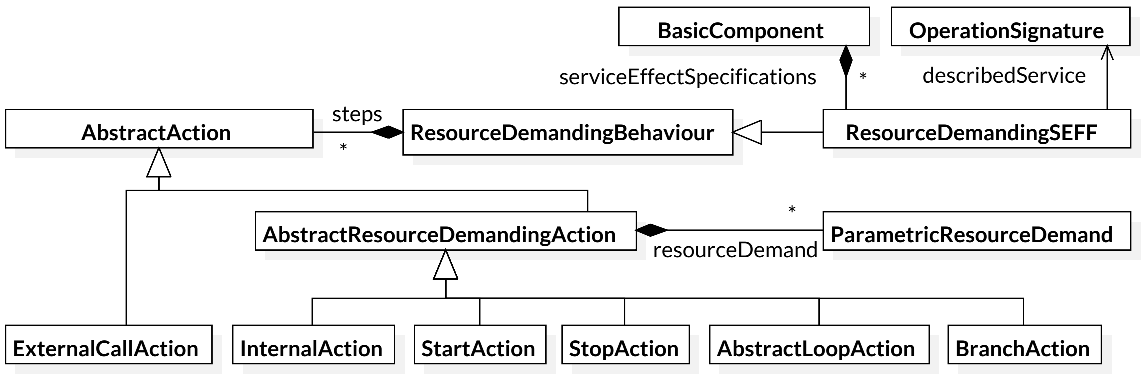


Figure 2.3: Important aspects of the Palladio resource demanding service effect specification meta-model.

The PCM ships with different steps extending `AbstractAction` to express resource demands. While PCM defines even more actions, the following are relevant for the approach developed in this thesis:

- **StartAction** and **StopAction**: The PCM requires them to be the first and last element, respectively, of the doubly-linked list of steps contained within a `ResourceDemandingBehaviour`.
- **InternalAction**: Models a resource demand affecting a specific resource specification (e.g., a CPU or HDD) of a resource container.
- **AbstractLoopAction**: Contains a nested `ResourceDemandingBehaviour` that is repeated as long as the loop condition holds.
- **BranchAction**: Allows to define conditional resource demands. Each conditional branch contains a nested `ResourceDemandingBehaviour`.
- **ExternalCallAction**: Calls an operation of another component through a required interface of the calling component.

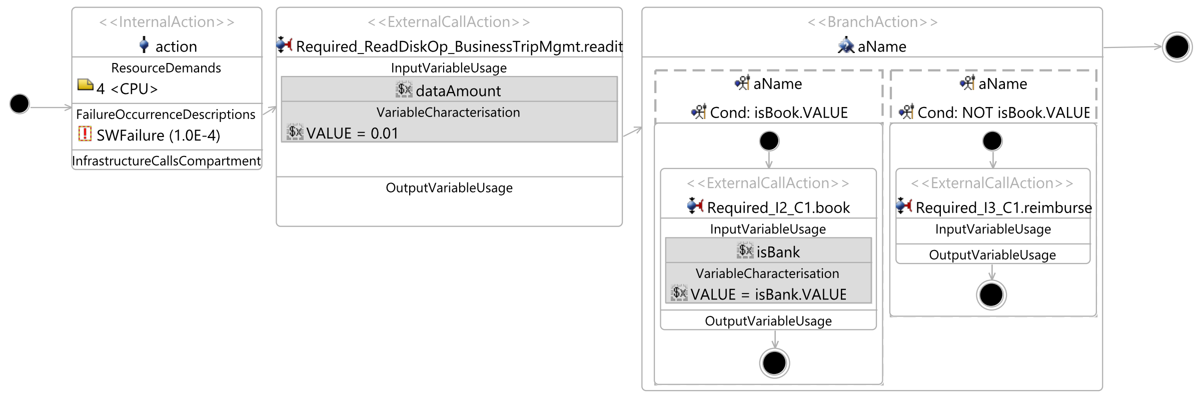


Figure 2.4: Example RDSEFF of an operation, first modeling a CPU demand of 4 units, then calling another component with signature parameter specifications before modeling a conditional resource demand depending on the value of a parameter of the called operation.

Figure 2.4 shows an example RDSEFF as visualized in the Palladio Bench tool. A black dot represents a `StartAction` and a black dot with surrounding circle denotes a `StopAction`. The arrows indicate the order of the steps in the RDSEFF. First, an `InternalAction` defines a CPU demand of 4. Then, an operation of another component is called with the `ExternalCallAction`, where a parameter of the called operation is characterized to be a certain value. Last, the behavior defines a `BranchAction` with two branches. Each branch defines a condition and if the condition is fulfilled, the nested RDSEFF is executed.

System

The system model assembles the components to a system by connecting them. In addition to that, the system model is capable of defining required and provided interfaces of the system itself. Figure 2.5 shows the PCM system model. A central concept of the system model is the `AssemblyContext`, which encapsulates any component of the repository and allows it to be connected to other components. Hence, multiple assembly contexts of the same component definition may coexist and can be connected differently. To connect two assembly contexts, an `AssemblyConnector` is used to define a unidirectional connection between the two encapsulated components. Hence, the `ExternalCall` actions defined in the RDSEFFs of a component requiring a service will use the component connected in the target assembly context. The system may also provide or require certain operations by defining the respective `*DelegationConnector`. The delegation connector refers to a specific provided or required role of a component in a assembly context.

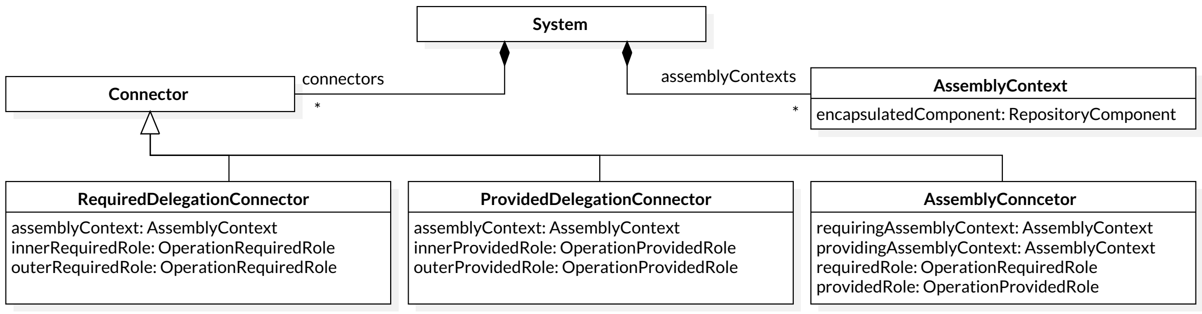

System
connectors
*
assemblyContexts
*
Connector
AssemblyContext
encapsulatedComponent: RepositoryComponent
RequiredDelegationConnector
assemblyContext: AssemblyContext
innerRequiredRole: OperationRequiredRole
outerRequiredRole: OperationRequiredRole
ProvidedDelegationConnector
assemblyContext: AssemblyContext
innerProvidedRole: OperationProvidedRole
outerProvidedRole: OperationProvidedRole
AssemblyConncetor
requiringAssemblyContext: AssemblyContext
providingAssemblyContext: AssemblyContext
requiredRole: OperationRequiredRole
providedRole: OperationProvidedRole

Figure 2.5: The PCM system model defines component connections.

Resource Environment

The resource environment defines the hardware the modeled system is allocated on. As shown in Figure 2.6, the resource environment consists of two concepts: resources denoted as `ResourceContainer` and communication links between the resources modeled as `LinkingResource`. A linking resource may connect an arbitrary number of resource containers and any resource demands affecting the communication of services running on connected containers are affected by the communication specification.

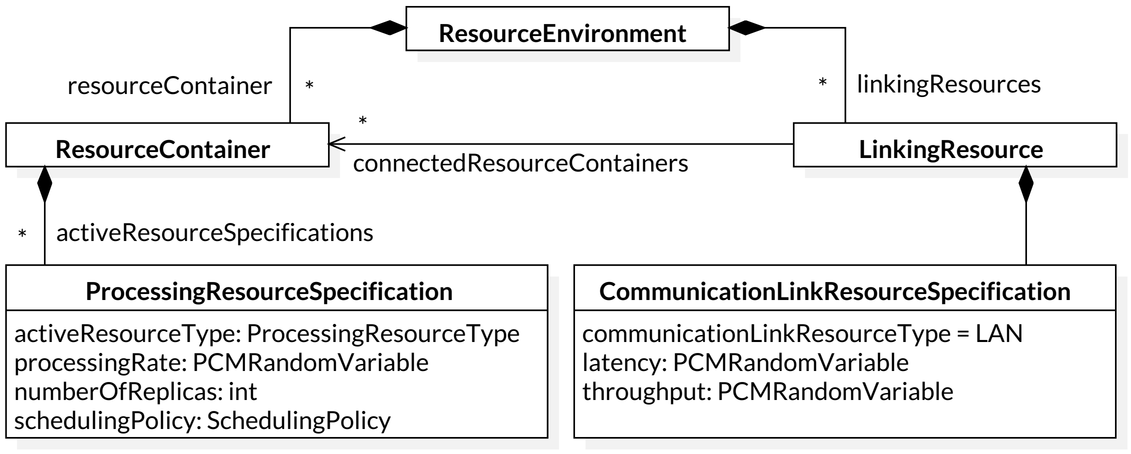

ResourceEnvironment
resourceContainer
*
*
linkingResources
*
ResourceContainer
connectedResourceContainers
LinkingResource
*
activeResourceSpecifications
ProcessingResourceSpecification
activeResourceType: ProcessingResourceType
processingRate: PCMRandomVariable
numberOfReplicas: int
schedulingPolicy: SchedulingPolicy
CommunicationLinkResourceSpecification
communicationLinkResourceType = LAN
latency: PCMRandomVariable
throughput: PCMRandomVariable

Figure 2.6: The PCM resource environment defines the available hardware the system is deployed on.

Both, resource containers and linking resources are parameterized by specifications attached to them. A `ResourceContainer` has a certain resource type, which in the scope of this thesis is either set to `CPU` representing a processor or `HDD` representing a data storage medium. The processing rate of the resource container specifies the processing speed of the CPU or the HDD and is set to a PCM random variable. The number of replicas models the number of cores the CPU has. Last but not least, the scheduling policy models in which order incoming requests are served. The available default scheduling policies are *first-come-first-serve*, *delay* and *processor sharing*. A `LinkingResource` may be parameterized by specifying the latency and the throughput, which are both PCM random variables.

Allocation

The purpose of the allocation model is to allocate the assembly contexts to actual processing resources of the resource environment. The `AllocationContext` as shown in Figure 2.7 establishes the mapping between a `AssemblyContext` and a `ResourceContainer` to which the context is deployed to. Each assembly context may only be deployed to a single resource container. The allocation model merely is a list of allocation contexts.

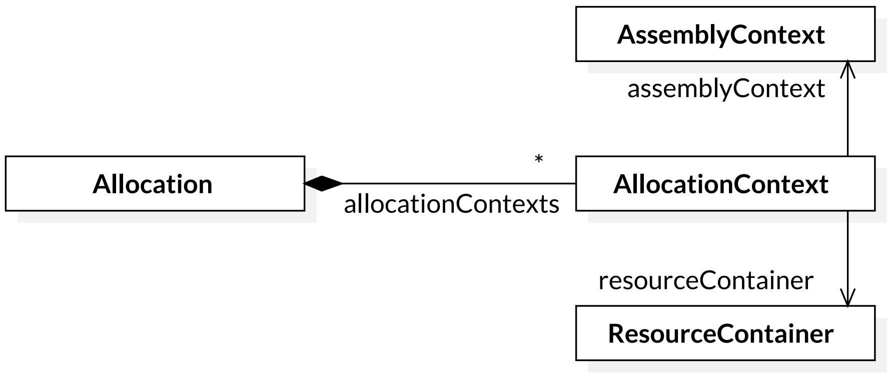


Figure 2.7: The PCM allocation maps the systems components to the processing capabilities defined in the resource environment.

Usage Model

The PCM usage model is used to specify an application workload scenario to be analyzed. Figure 2.8 shows the meta model of the usage model. Each scenario of the usage model has an associated workload and a behavior. The workload is either open or closed and can be parameterized accordingly. The scenario behavior is defined by an ordered list of steps, similar to the RDSEFF specification. The behavior steps have to start and end with the `Start` and `Stop` actions, respectively. The control flow of the scenario can be altered by defining a `Branch` action for conditional behavior or a `Loop` action for repeated behavior. The `Delay` action simply delays the execution of the next step by the specified amount of time and can for example be used to model think time between subsequent user requests.

To actually call the system, the `EntryLevelSystemCall` is used. The system call targets an operation provided by the system and is capable of modeling parameter

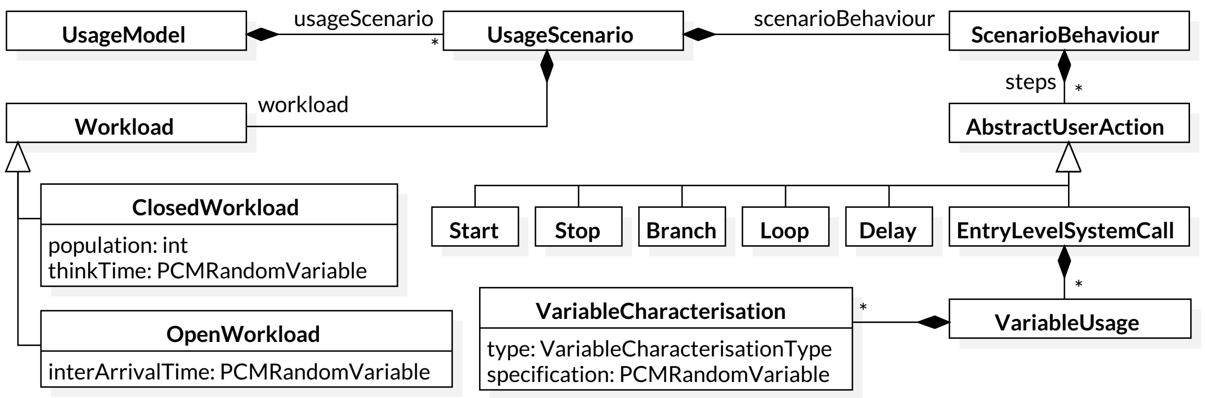


Figure 2.8: The PCM usage model defines a use case scenario for the application usage.

usages. If the called operation has parameters, the parameter can be parameterized by adding a `VariableUsage` declaration for the parameter. The `VariableUsage` then has a set of `VariableCharacterisations` containing the actual parametrization, which allow to precisely model performance-relevant parameter meta data. For example, an operation sorting an array expects an array as input, but the array contents itself do not impact the performance a lot. Instead, the characterization can specify the `NUMBER_OF_ELEMENTS` the array has, as that is the performance-relevant metadata for the sorting operation. Analogous to that, to model the performance of an operation writing data to a file the number of bytes written are relevant to the performance and as such, the characterization may specify the `BYTESIZE` as parameter meta data. With the variable usage and characterization concepts the usage model can precisely model performance-relevant behavior of the system.

2.2.2 Layered Queuing Network Analysis

Layered queueing networks (LQNs) are used to predict the performance of distributed software systems (Franks et al., 2009). Layered queueing networks extend the modeling capabilities of regular queuing networks by a hierarchical organization of queues. The hierarchical organization is important, as a server calling a second server remains in a blocked state until the call returns, which is what the LQN is capable of modeling. The concept of layered queueing networks has been extended by features important to real application systems and various solvers have been developed to cope with these additional features (e.g., Fontenot (1989), Menascé (2002), and Petriu et al. (1991)). The feature extensions include, but are not limited to, multithreaded servers, asynchronous messaging, issuing requests to all servers of lower layers, or modeling CPU demands of any variance.

Koziolek (2008) developed a transformation from the Palladio component model to a corresponding layered queueing network. The transformation resolves parameter dependencies of the Palladio components and turns the architectural concepts of the component model to an equivalent representation in the LQN modeling language. The Palladio tool suite uses the LQN solver developed by Franks et al. (2005) to solve the transformed models. While the LQN solver is only able to conduct a mean-value analysis, the analysis execution time is substantially lower compared to a simulated performance analysis of a PCM (Becker, 2008). Thus, the LQN approach perfectly fits the goal of this thesis to raise performance awareness by providing immediate analysis feedback to the user.

2.3 JetBrains MPS and the IET$_S$^{3} Project

The presented approach is integrated into IET$_S$^{3}, an Integrated Specification Environment for the Specification of Technical Software Systems. IET$_S$^{3} is built on top of the JetBrains Meta Programming System (MPS) (*Jetbrains MPS*), which is an open-source language workbench (Erdweg et al., 2013), i.e., a system for defining, composing and using languages and their IDEs. A discrete event simulation performance analysis tool called Simbench has already been integrated into IET$_S$^{3} and is integrated with the approach developed in this thesis.

2.3.1 JetBrains MPS

The JetBrains Meta Programming System (MPS) supports concrete and abstract syntax, type systems, transformations, as well as IDE aspects, such as syntax highlighting, code completion, find-usages, diff/merge, refactoring, and debugging. MPS uses a projectional editor, which means that a user's edit actions directly change the underlying abstract syntax tree of a program, which then is just projected to resemble the natural look and feel of a text-like editor. Hence, the approach does not require a parser to create the abstract syntax tree and thus the language is not limited by the constraints parsers impose on language design. Consequently, MPS supports a wide variety of notations (Voelter et al., 2014) as well as essentially unlimited language composition (Voelter, 2011). MPS is used widely in industry for languages in a wide variety of domains, including embedded software, insurance, health and medicine, as well as aerospace. MPS offers the following concepts to customize a domain-specific language:

- **Structure**: The structure defines the abstract syntax of the DSL. The abstract syntax comprises *concept* definitions, which feature inheritance, properties, children, and references to other concepts.
- **Editor**: Editors form the concrete syntax of a DSL that the user sees on the screen and interacts with. As the contents are projected in MPS a concept may have multiple editors (e.g., a textual notation and a graphical notation).
- **Constraints**: Constraints restrict the use of concepts to certain locations in the abstract syntax tree, e.g., a “can be child” constraint may limit the use of a certain concept only within a certain parent concept.
- **Typesystem**: Concepts have a defined type that is derived by typesystem rules. The typesystem allows to define complex validation scenarios to force a proper usage of the DSL.

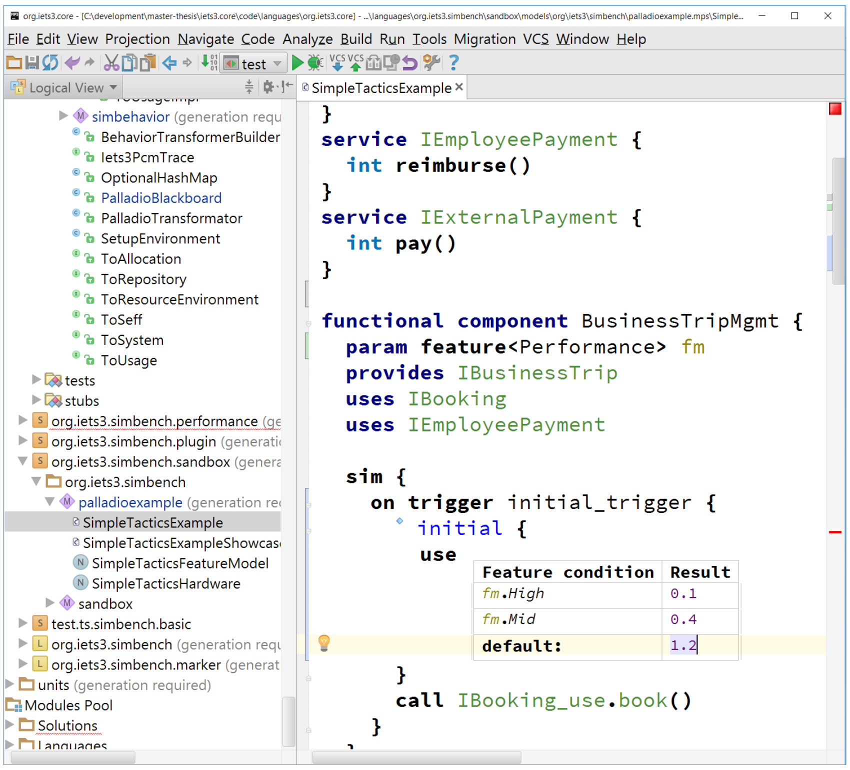
Next to the DSL definition, MPS also defines some common techniques for the user interaction with the DSL. A *plugin action* is able to provide custom entries to the context menu opened for a specific concept. Similar to a context menu, *intentions* are also defined for a specific concept. A keyboard shortcut then shows a list of defined intentions for the selected concept and the user may choose which intention to execute (i.e., change visibility of a Java property to public instead of changing the modifier itself).

Figure 2.9 shows a screenshot of the JetBrains Meta Programming System user interface. The user interface is very similar in terms of the look and feel to a regular IDE. The left pane shows all models belonging to the project and the right pane shows the currently opened model. The right pane is the projectional editor, which is why the feature specific use in the component definition can be rendered as a table.

2.3.2 The IET$_S$^{3} Project

IET$_S$^{3} is a research project striving to develop an IDE-like tool for writing specifications of software systems. Traditionally, the specification of software systems is scattered among various tools having different degrees of formal verifiability. Specifications are typically written in prose, for example using text editors like Word or Excel, or using requirements tools such as DOORS. To check these documents for internal consistency and correctness is nearly impossible due to the prose format not being suited for formal analysis methods. Additionally, the tools typically cannot reference artifacts in other tools, making it difficult to maintain consistent documents and to enable traceability.

To tackle the described problem, IET$_S$^{3} relies on the JetBrains MPS language workbench leveraging its facilities for language composition and notational flexibility. IET$_S$^{3} strives


org.iets3.core - [C:\development\master-thesis\iets3.core\code\languages\org.iets3.core] - ...\languages\org.iets3.simbench\sandbox\models\org\iets3\simbench\palladioexample.mps\Simple...
File Edit View Projection Navigate Code Analyze Build Run Tools Migration VCS Window Help
test
Logical View
SimpleTacticsExample

```
simbehavior (generation requ
  BehaviorTransformerBuilder
  Iets3PcmTrace
  OptionalHashMap
  PalladioBlackboard
  PalladioTransformator
  SetupEnvironment
  ToAllocation
  ToRepository
  ToResourceEnvironment
  ToSeff
  ToSystem
  ToUsage
tests
stubs
org.iets3.simbench.performance (ge
org.iets3.simbench.plugin (generati
org.iets3.simbench.sandbox (genera
  org.iets3.simbench
    palladioexample (generation re
      SimpleTacticsExample
      SimpleTacticsExampleShowcas
      SimpleTacticsFeatureModel
      SimpleTacticsHardware
    sandbox
test.ts.simbench.basic
org.iets3.simbench (generation requ
org.iets3.simbench.marker (generat
units (generation required)
Modules Pool
Solutions
Languages
```

```
}
service IEmployeePayment {
  int reimburse()
}
service IExternalPayment {
  int pay()
}

functional component BusinessTripMgmt {
  param feature<Performance> fm
  provides IBusinessTrip
  uses IBooking
  uses IEmployeePayment

  sim {
    on trigger initial_trigger {
      initial {
        use
          Feature condition  Result
          fm.High            0.1
          fm.Mid             0.4
          default:           1.2
      }
      call IBooking_use.book()
    }
```

Figure 2.9: A screenshot of the JetBrains Meta Programming System user interface.

to seamlessly integrate informal, semi-formal and formal specification languages using the features and capabilities of the language workbench. By using a language workbench, $\text{IET}_\text{S}{}^3$ is able to provide automatic checks with visual feedback in case of errors to help users write correct and coherent specifications. In addition to that, the languages allow the specification to be automatically transformed to other artifacts, such as generated code. To improve the user experience, interactions commonly used in IDEs by developers are added to the specification editor, such as refactorings to help users consistently change parts of the specification or quick fixes to solve common mistakes.

At the core, the $\text{IET}_\text{S}{}^3$ tool suite comes with four languages: expressions, components, requirements, and feature models to support software product lines (SPLs):

- **Expressions**: Expressions are a very versatile language construct to allow the definition of arbitrary, extensible expressions. Any additional language can contribute expressions that integrate seamlessly with existing expressions. A strict type system and checking rules help keep the expressions valid and form a solid foundation for using references to specification elements within expressions. For example, requirements can reference elements of a component definition by using expressions.

- **Components**: The component language allows to define components and their connections through specific ports. The components may contain instances of other components inside a *substructure*, allowing component hierarchies to be formed. The component language is used to model architectural specifications of the system and is extensible to have a solid foundation for component-based software engineering.
- **Requirements**: The requirements language allows to write requirements with a semi-formal language. While the requirements are generally written in prose text other requirements or term definitions can be referenced from within the natural language text. Requirements also contain metadata, such as tags, the person in charge, a priority or references to conflicting requirements.
- **Feature model**: Feature models are used to describe product lines and provide graphical editors showing the tree structure of the feature relationships. The feature model language supports constraints between features, parameters to parametrize a feature, and references between feature models. A specific variant can be modeled using a `Configuration`, which basically stores which features are enabled and sets parameters, if necessary. The feature model is tightly integrated into other languages of the $\text{IET}_\text{S}{}^3$ project to model feature-specific capabilities.

One of the first concrete tools built on top of the $\text{IET}_\text{S}{}^3$ core languages is Simbench (Birken et al., 2010), a tool for discrete performance simulation. Simbench extends and enriches the core languages of $\text{IET}_\text{S}{}^3$ in many places to provide the performance analysis capability:

- **Component language extension**: Simbench extends the component language to add performance analysis specific language concepts to components. A key part of the extension is the ability to specify resource demands for the services a component provides.
- **Hardware language**: A new language has been added to model hardware elements such as processors or passive resources (e.g., a hard disk). The components of the system can then be allocated on the modeled hardware elements and the performance analysis will use the processing speeds and capabilities of the hardware elements the components are deployed on to analyze the system performance.
- **Usage model language**: The usage model is able to model typical scenario work loads the system will encounter. The work loads are then used by the performance analysis to determine the system performance for the respective scenario. Performance requirements can be added to a scenario and are verified with the analysis

results. A deep integration with the IET$_S$³ requirements language is planned at this point in time, but has not yet been implemented.

- **Expression integration**: The expression language is used throughout all languages and extensions Simbench provides. Primarily, the expression language allows to reference parts of the Simbench model at appropriate places, for example a call to a required service from within a resource demand of a component. Typically, the defined expressions enable a hierarchical access of model elements (e.g., `root.child.grandChild`) and are strongly constrained to only allow semantically valid models.
- **Feature model integration**: The feature model language is tightly integrated into Simbench by means of language embedding. Resource demands, hardware allocations and scenarios may be feature-specific and can be enabled by adding a feature expression to the model elements. This integration allows to model performance behavior for software product lines and is an important foundation for this thesis.

Simbench provides performance-relevant modeling capabilities to the IET$_S$³ environment and even implements a performance analysis based on discrete-event simulation. Hence, the Simbench language and its extensions lay the groundwork for the approach developed in this thesis.

2.4 Software Product Lines

Software product lines (SPLs) allow software systems to be built by composing features with defined variability points to a deliverable system (Van Gurp et al., 2001). Product line software engineering is thus concerned with two aspects: developing the actual features (i.e., core asset development) and composing those features to craft a product (i.e., product development) (Northrop et al., 2001). A key ingredient to successfully applying product line software engineering is a strong feedback loop between those two activities, as the requirements for the core assets change with new products and in turn, better core assets allow to build better systems. With the guidance of a marketing-oriented perspective an organization is able to engineer systems using software product lines that meet business goals while satisfying the customer needs (Kang et al., 2002).

Tools have been developed to aid organizations in developing software product lines. Feature models are such a tool, allowing to model all possible variants of a product line (Lee et al., 2002). Feature models define required and optional features as well as their interdependencies by imposing constraints between features that must hold for all valid configurations.

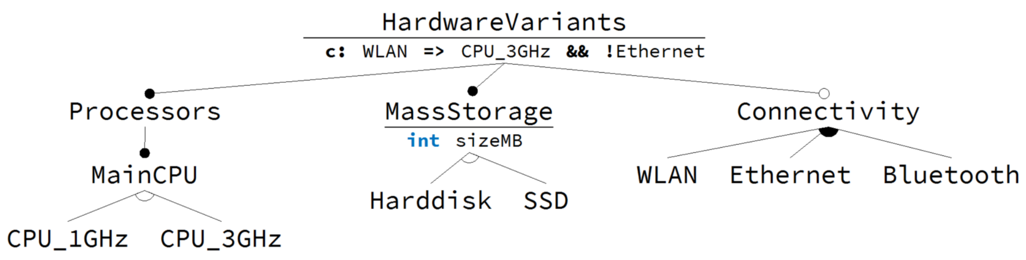


Figure 2.10: Example feature model as visualized in the IET$_S$^{3} editor.

Figure 2.10 shows an example feature model of a hardware product with a notation similar to the notation introduced by Kang et al. (1990). The relationships between the features have different semantics denoted by the respective icon. In general, the feature model is read top-down and the relationships indicate which child features can or need to be enabled if the parent feature is enabled. The `HardwareVariants` feature has three child features, of which the two with the filled dot are mandatory (i.e., `Processors` and `MassStorage`) while the unfilled dot of the `Connectivity` feature denotes an optional feature. Thus, each valid variant has to have a processor and a mass storage, but may or may not specify the connectivity feature. The unfilled arc of the `MassStorage` feature denotes a logical exclusive-or-relationship (XOR). Hence, if the `MassStorage` feature is enabled either the `Harddisk` or the `SSD` feature must be enabled, but not both at the same time. The filled arc below the `Connectivity` feature denotes a logical or-relationship (OR), where any child feature may be enabled. Accordingly, any of `WLAN`, `Ethernet` or `Bluetooth` may be enabled or disabled. While the relationship types allow to express a rudimentary set of interdependencies between the features, *constraints* allow to model more sophisticated interdependencies. For example, the `HardwareVariants` has a constraint stating that if `WLAN` is enabled, the `CPU_3Ghz` feature must be enabled and the `Ethernet` feature must be disabled to form a valid variant.

As features can be composed in a tremendous number of variations, it is challenging to analyze the performance impact of features in different product variations (Kolesnikov et al., 2013). Recent efforts in the field of SPL performance prediction leverage statistical techniques to try to cover the variability space. Zhang et al. (2016) aim to find performance-relevant feature interactions using Fourier transforms. Guo et al. (2013) train decision trees with a small, randomly selected set of variants and achieve a prediction accuracy of the software performance of over 94%. An interesting finding of the approach is that the built decision tree model is understandable by humans, as by definition a decision tree model embodies the feature interactions that lead to a certain performance. Thus, by analyzing the feature model itself engineers are be able to determine feature interactions that have a high performance impact. Valov (2014) explore different regression methods and parameters for the decision tree software performance prediction and found that bagging outperforms CART, random forests and SVM regression.

IET_S^3 provides a feature model and thus allows SPL performance analyses to be conducted on variability-aware models. The notation for feature models as shown in Figure 2.10 is part of IET_S^3 and deeply integrated with the component and performance modeling languages.

2.5 Declarative Performance Engineering

Declarative performance engineering is a new perspective on software performance engineering and emphasizes the user trying to reason about the performance of a system (Walter et al., 2016). Research has developed various tools and techniques to assess the performance of a software system, during design as well as during operation, and provides means for performance management (Brunnert et al., 2015). However, often an expert is needed to configure and parametrize the approach and to interpret the analysis results. Moreover, choosing the right approach to analyze a specific scenario is also a challenging task requiring experience of the analyst.

A key vision of declarative performance engineering is to distinguish between the actual concerns of the user and the task of selecting and parametrizing a performance analysis approach for the given concern. By expressing the concerns of the user in a (yet to be developed) declarative language, techniques can be developed to automatically select and parametrize appropriate methods, techniques or tools based on the user concern. It is planned that the declarative language supports *system element queries* to query the modeled system for its capabilities, *user-controlled queries* to query the modeled system for its performance, *sensitivity queries* to analyze the impact of certain model elements, *temporal queries* to analyze the system over a period of time, and *goal definitions* to enable users to specify goals the system should or must reach. The language is then processed and fed into a capability model and decision engine to decide which technique to use.

While the research in declarative performance engineering is still in its early stages, the decisions for the approach developed in this thesis are geared towards building a reusable platform for declarative performance engineering. Special care has been taken to expose extension points that are capable of integrating declarative performance engineering tools in the future.

2.6 Related Work

This section discusses the related work regarding the core contribution of this thesis: real-time performance awareness in the software development process. Related work primarily addresses performance awareness for developers during the implementation phase of a project.

Horký et al. (2015) use performance unit tests to generate performance documentation for the developers using the components to make informed decisions. The performance unit tests are executed with a certain workload and the performance measurement results are presented to the developer. The envisioned integration of the approach into the user interface of an IDE has not yet been implemented, as the proposed approach focuses on workload and parameter selection of the test cases.

Danciu et al. (2015) employ Palladio to predict the response time of Java EE component operations during their implementation within an IDE. The approach predicts the performance of the implementation by converting it to a parametrized Palladio component model modeling the implemented control flow structure and all calls to external services. The response times of external services under a certain workload are retrieved by instrumentation of a representative running system. Performance prediction results are then visualized next to the affected source code, especially highlighting operations where the predicted performance exceeds a predefined threshold.

Contrary to predicting the performance, Beck et al. (2013) integrate performance profiling results directly into the code. The profiling results are visualized as percentage of time consumed for a particular line of code. The approach also features user interaction concepts that all start from the small percentage indicator next to a line of code. Due to rendering the results near the affected location the approach helps developers keep focused on their task at hand and not distract them with complicated views or sidebars. A user study revealed that the compact display is helpful to developers, but that a traditional complex view is indeed helpful in certain situations.

The approach presented by Johnson et al. (2013) requires the software architecture to be modeled in UML and allows to define descriptive and predictive expressions using an extension to the object constraint language (OCL). The approach is capable of probabilistically analyzing the modeled architecture using an algorithm for probabilistic inference.

Performance awareness is also concerned with the real-time feedback loop for users involved in the software development process (Voelter et al., 2013). For real-time feedback incremental analysis can yield very fast response times (Szabó et al., 2016) and hence, a great user experience.

The approach developed in this thesis introduces performance awareness into the IET$_S$^{3} specification environment by transforming the IET$_S$^{3} model to a corresponding Palladio component model. The results of the analysis are lifted back to the IET$_S$^{3} model and are automatically attached to the specified performance requirements, similar to how Beck et al. (2013) visualize the profiling results. The developed approach provides a real-time feedback loop and is extensible to other performance analyses.

Chapter 3

Performance Analysis Abstraction

Performance analyzers typically analyze a single model and return the results depending on their tooling infrastructure. To increase performance awareness in the IET$_S$^{3} integrated specification environment the aim of this thesis is to be able to use multiple performance analyzers. A diverse range of performance analyzers in such a setting is of high value as they exhibit different characteristics: some may execute very fast, some may yield precise results and others may only be able to analyze a certain part of a model. Depending on the question the user is trying to answer, different performance analyzers may be better suited for the job.

To be able to easily execute different analysis approaches a reusable framework for software performance analysis called *fastpan*, which has been open-sourced, was developed (Keller, 2016a). The framework provides a set of interfaces and default implementations to write a performance analysis for a certain type of model. Most importantly, the framework provides a means to express performance results through a unified interfaces. A strictly-typed class hierarchy allows to express various performance measures, which are attached to parts of the model to form an analysis result.

Section 3.1 describes how a performance analysis approach is integrated into the fastpan framework. The uniform performance result interface is then presented in Section 3.2. Finally, Section 3.3 introduces concepts required to build variability-aware performance analyses.

3.1 Performance Analyzers

Using fastpan, a performance analysis for a specific system (e.g., the Palladio component model) is defined by implementing the `PerformanceAnalyzer` interface as shown in Figure 3.1. A `PerformanceAnalyzer` has to designate its specific capabilities (e.g., a

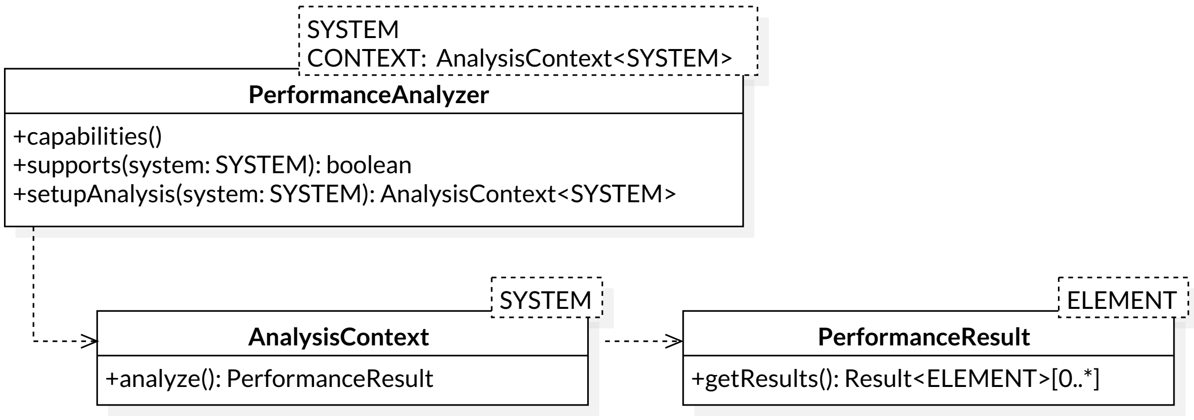

SYSTEM
CONTEXT: AnalysisContext<SYSTEM>
PerformanceAnalyzer
+capabilities()
+supports(system: SYSTEM): boolean
+setupAnalysis(system: SYSTEM): AnalysisContext<SYSTEM>
SYSTEM
AnalysisContext
+analyze(): PerformanceResult
ELEMENT
PerformanceResult
+getResults(): Result<ELEMENT>[0..*]

Figure 3.1: Required interfaces to define a performance analysis with the fastpan framework.

fast execution time) and can determine whether a particular instance of the system can be analyzed. The `PerformanceAnalyzer::setupAnalysis` method is a factory method that configures a specific `AnalysisContext` for the given system. The context instance is then able to execute the actual performance analysis and yields a performance result.

Figure 3.2 shows two performance analyzers built on top of the fastpan framework that were developed during this thesis. The `PcmLqnsAnalyzer` is an adapter wrapping the analysis provided by Palladio setting up and configuring the `PcmLqnsAnalyzerContext` which analyzes a system contained in a Palladio `PcmInstance`. To integrate the analyzer into IETS_S^3, the `PalladioLqnsAnalyzer` sets up and configures the `PalladioLqnsContext` for the IETS_S^3 model. The context performs a model-to-model transformation from the IETS_S^3 model to the PCM model and reuses the existing `PcmLqnsAnalyzer` to execute the LQN solver on the transformed PCM. As the results of the Palladio LQN analysis are wrapped with a fastpan performance result, the results are attached to a specific model element and comprise a certain performance measure. The IETS_S^3 `PalladioLqnsContext` keeps track of the tracing between the IETS_S^3 model elements and the transformed PCM elements and can thus lift the performance results to the IETS_S^3 model elements. The main advantage of using the fastpan framework in this scenario is that the IETS_S^3 model can be analyzed with any other approach capable of analyzing a PCM instance.

The framework has been designed to allow performance analyzers to solely consist of a model-to-model transformation to provide the input for another performance analyzer. Transforming to a system model that already has performance analyzers integrated into the fastpan framework allows these to be reused. When implementing a model-to-model transformation it is important to trace the transformed model parts to be able to lift

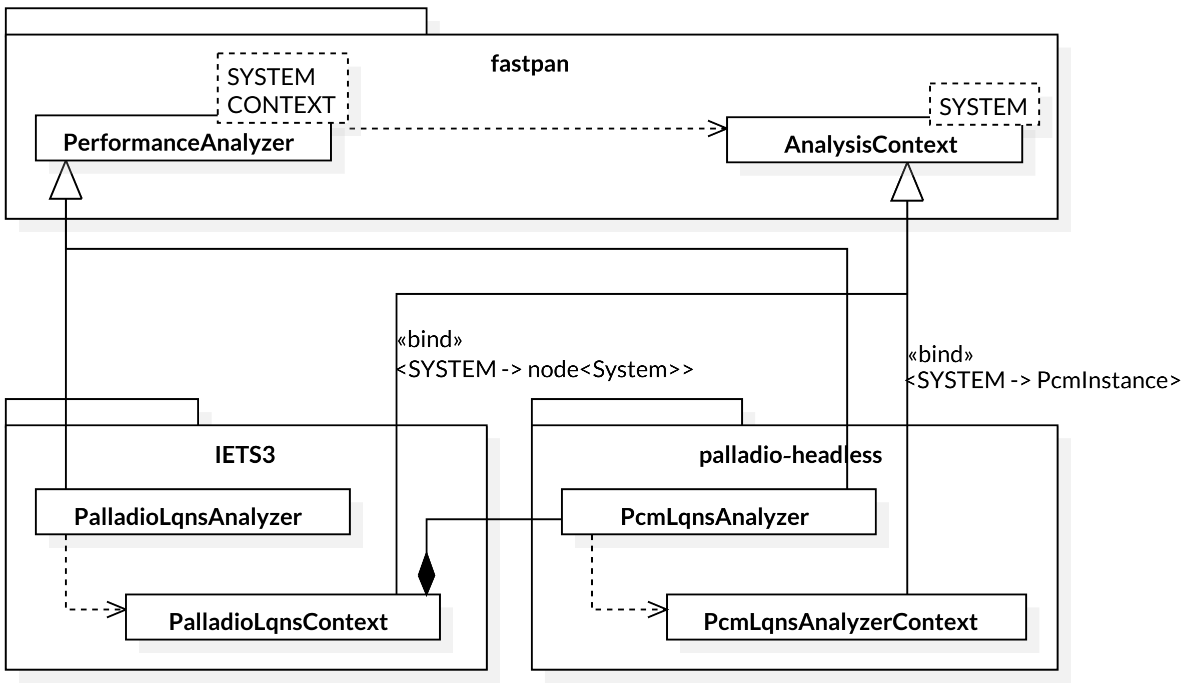


Figure 3.2: The fastpan interfaces used to setup the LQN analysis provided by Palladio for two different models: the Palladio component model itself and the $\text{IET}_\text{S}{}^3$ model, which reuses the PCM analyzer.

the performance analysis results of the underlying performance analyzer to the original model. As seen in Figure 3.1, `PerformanceResults` are attached to certain elements of the model, while the actual performance metric is encapsulated such that the actual result can easily be remapped to elements of another model. The developed approach in this thesis as described in Chapter 4 uses this technique to transform the $\text{IET}_\text{S}{}^3$ model to the Palladio component model and lift the analysis results back to the $\text{IET}_\text{S}{}^3$ model.

3.2 Performance Analysis Results

Performance measure definitions vary among literature and are used ambiguously. To be able to compare the results of different performance analyses, all relevant software performance measures are defined in this section and are based on Jain (1990). The following notations are required to formulate the measures:

- **Resource** R_i: Metrics can be related to a specific resource (e.g., a CPU), which is denoted as $R_i \in R$, $i \in \mathbb{N}$ being the i-th resource of the system.

- **Set of Resources** R_S: Metrics may be defined for a set of resources, which is denoted as $R_S \subseteq R$.
- **Observation time** T: The total observation time in a specific unit of time.

The names of the presented metrics are directly related to the `PerformanceMeasure` classes in the fastpan framework, which represent the semantic meaning as defined in this section.

3.2.1 General Time-based Performance Metrics

BusyTime B_i: The amount of time a processing resource R_i is busy, i.e., being used.

IdleTime I_i: The amount of time a processing resource R_i is idle, i.e., not being used.

By definition, the total observation time is the sum of the busy and idle time of a resource: $T = R_i + I_i$

Utilization: The proportion of a time span T that a single resource R_i is being busy B (i.e., being used).

$$\text{Utilization}(R_i) = \frac{B}{T} \tag{3.1}$$

By definition, $0 \leq Utilization(R_i) \leq 1$ always holds, as the utilization is defined for a single resource. To cover the utilization of multiple resource, the following term is defined:

MultiUtilization: The proportion of the observed time span T that a certain number of resources R_S are busy (i.e., being used).

$$\text{MultiUtilization}(R_S) = \frac{\sum_i B_i}{T}, \forall i \in \mathbb{N} : R_i \in R_S \tag{3.2}$$

3.2.2 Service-based Performance Metrics

Service-based performance metrics provide a top-level view on the performance of a system by specifying metrics for measureable quantities outside the actual system, e.g., the response time of a web server. Figure 3.3 shows an overview of the available service-based metrics and how they relate to each other.

ReactionTime: The time a service requires after a successful request has been made to start the actual execution.

ServiceExecutionTime: The time the service actually takes to execute the request.

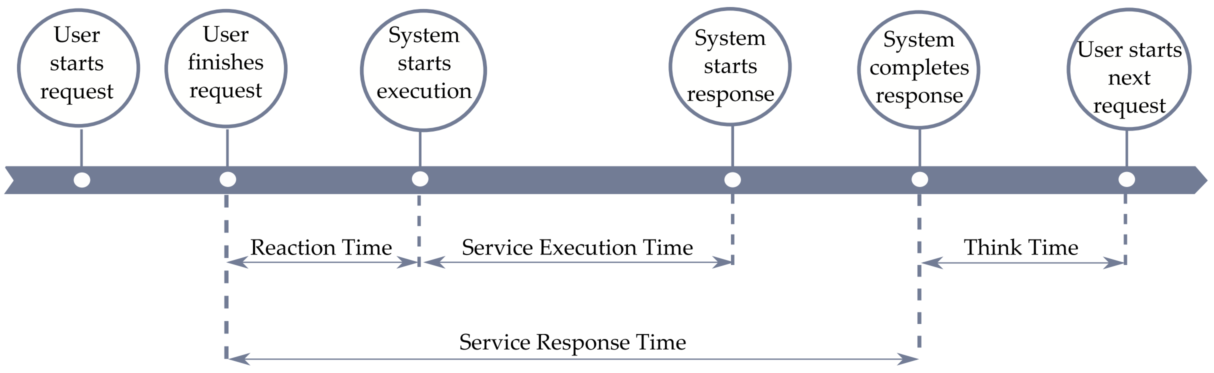


Figure 3.3: Performance measures based on the user and system/service activity phase adapted from Jain (1990).

ServiceResponseTime: The total time it takes a service to answer a user request.

ThinkTime: The time it takes a user to think before launching the next request. Typically, the user needs to process the response just received before being able to fire the next request.

3.2.3 Resource-based Performance Metrics

In contrast to service-based performance metrics, resource-based performance metrics focus on the inner workings of a system.

OperationExecutionTime: The time an operation spends actively computing something. The waiting time for other, called operations is excluded.

OperationResponseTime: The total time an operation requires to complete.

ResourceCount: A discrete number of resources to quantify the particular size of parts of the system.

Arrivals: The total number of arrivals a resource or system has during a time interval. The arrival count is of a certain unit, e.g., messages, requests or tasks.

CompletedArrivals: The number of arrivals that have been completed.

ArrivalRate: The rate at which arrivals reach the specific resource or system.

$$\text{ArrivalRate}(R_i) = \frac{\text{Arrivals}(R_i)}{T} \tag{3.3}$$

ArrivalCompletionRate: The proportion of arrivals that have been completed.

$$\text{ArrivalCompletionRate}(R_i) = \frac{\text{CompletedArrivals}(R_i)}{\text{Arrivals}(R_i)} \tag{3.4}$$

VisitCount: The number of visits a certain request has at a certain resource.

Throughput: The number of completed arrivals at a certain resource in a specific period of time.

$$\text{Throughput}(R_i) = \frac{\text{CompletedArrivals}(R_i)}{T} \tag{3.5}$$

3.3 Variability-aware Performance Analyzers

In many domains, such as embedded systems engineering, software product lines (SPLs) are used to build software systems. Traditional performance approaches neglect the needs of a software architect designing a software product line, as traditional approaches only allow a single software model to be analyzed. However, the software architect needs to ensure that the QoS requirements, or in particular performance requirements are met for all valid configurations.

The fastpan framework defines an abstraction for a variability-aware model analysis. The abstraction assumes that the system model to be analyzed has an associated feature model where configurations define which features are enabled or disabled. The system model then exposes variability points that alter the model depending on whether a feature is enabled or disabled. The fastpan abstraction provides a means to analyze these variability-aware system models.

The interfaces depicted in Figure 3.4 are used to define a variability-aware performance analysis. The entry point to the variability analysis is the `VariabilityAnalyzer` and analogously to the performance analysis of a single system, the variability analyzer is considered to be a stateless factory of a respective context class that conducts the actual analysis and is stateful. An implementation of the `VariabilityAnalyzer` needs to be parametrized with three different classes:

- **SYSTEM**: References the system model that is obtained when applying a specific configuration to a system with variability points.
- **CONTEXT**: Defines the `VariabilityContext` of the analyzer, that conducts the actual analysis and stores the analysis state.
- **FEATURE**: A class representing a single feature that can either be enabled or disabled for a specific configuration.

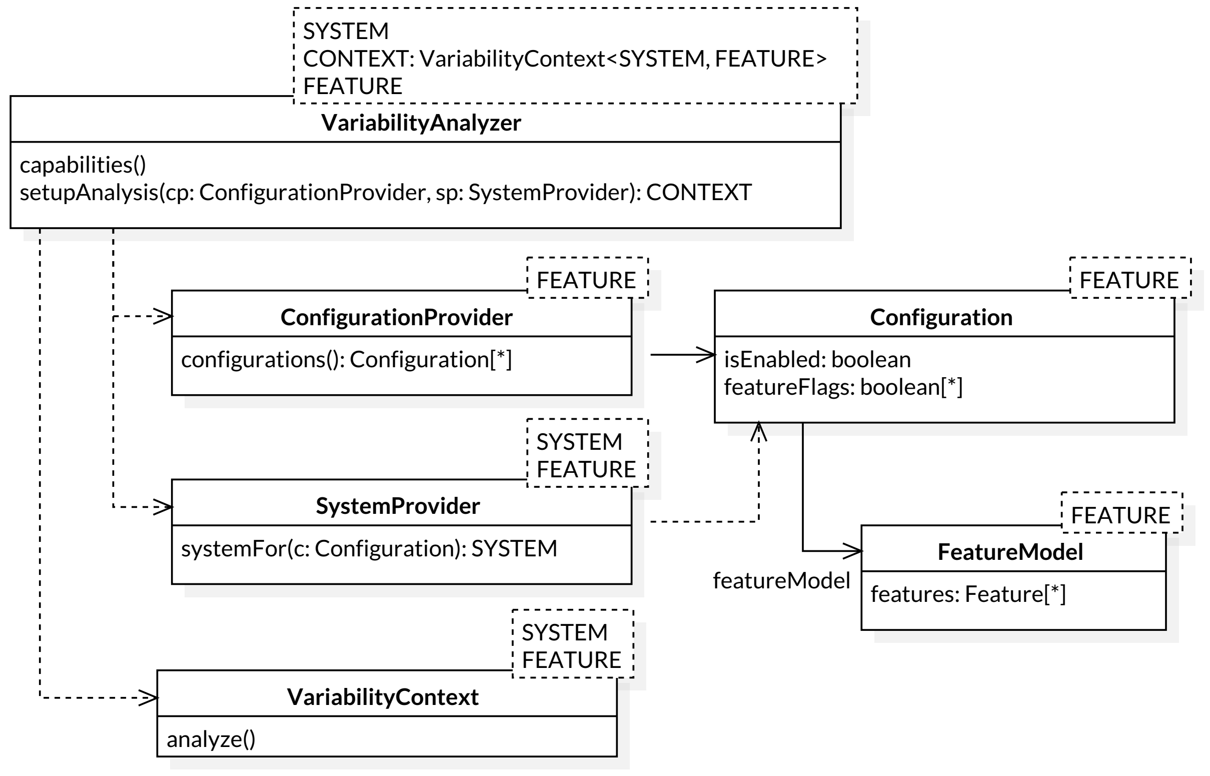


Figure 3.4: Required interfaces to define a variability-aware performance analysis with the fastpan framework.

To setup a `VariabilityContext` two parameters need to be supplied: A configuration provider and a system provider. The configuration provides supplies a list of configurations that should be analyzed by the variability analyzer. Currently, the fastpan framework does not provide a full-fledged feature model. The implemented feature model is a list of features and a configuration associates a boolean flag to each feature stating whether the feature is enabled or disabled. Users of the framework will most likely already have an existing feature model supporting constraints between features and enhanced analyses, such as satisfiability checks. However, the interface definitions of the fastpan framework are designed to integrate backing third-party feature models. In addition to the configuration provider, the analyzer requires a system provider. The system provider is used to derive a specific system for a given configuration allowing the variability points to be tailored with the settings provided by the configuration.

Using the two providers, the `VariabilityContext` can iterate all configurations to be analyzed and derive the specific system for each configuration. The context may then analyze the system and store any results in the context class. The context does not provide a uniform result interface similar to the regular performance analyzer, as the results a variability analysis may yield are very different. For example, fastpan provides a `BaseAnalyzer` which uses any existing performance analyzer for a single system to analyze the model derived for each configuration. The result of the `BaseAnalyzer` thus is a map assigning a fastpan performance result to each analyzed configuration. In contrast to that, another variability analyzer may analyze which feature combinations have the most adverse effect on the resulting performance. In this case, the result of the analysis would be a list of feature combinations.

Chapter 4

Introducing Performance Awareness in $\mathrm{IET_S}^3$

The goal of the approach presented in this thesis is to add performance awareness to the $\mathrm{IET_S}^3$ specification environment. Hence, the specification environment must treat software performance as first-class citizen for component developers, system architects, and domain experts already during the requirements elicitation and design phase of the software to be built.

This chapter conceptually describes the approach developed in this thesis. Section 4.1 outlines the approach for a regular performance analysis of a system model, while Section 4.2 outlines the approach for the variability-aware performance analysis. The tooling infrastructure implemented to achieve the outlined approach is described in Section 4.3. Finally, Section 4.4 explains how the existing $\mathrm{IET_S}^3$ editor was modified to introduce the performance awareness.

4.1 Regular Performance Analysis

A regular performance analysis evaluates a single system defined in the $\mathrm{IET_S}^3$ modeling language. In case feature modeling is used, the user has to choose the specific variant that should be analyzed. A general overview of the approach is depicted in Figure 4.1 and consists of the following steps:

1. **Create component model:** The architects and domain experts model the system and its requirements. Component developers supply reusable components with performance-related annotations to model the expected performance behavior. The

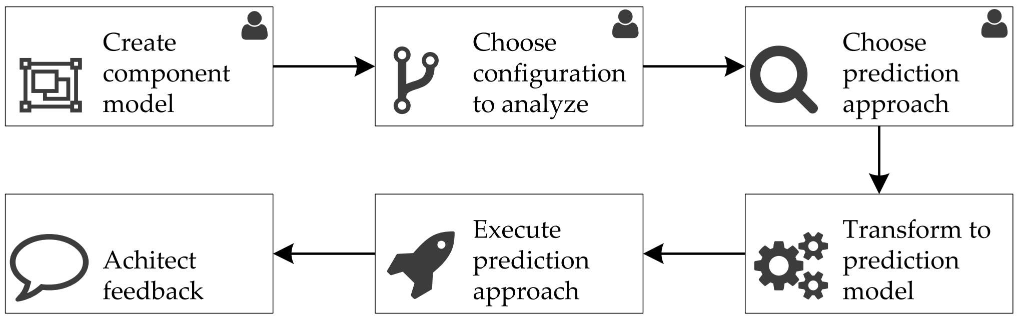

Create component model
Choose configuration to analyze
Choose prediction approach
Transform to prediction model
Execute prediction approach
Achitect feedback

Figure 4.1: Conceptual overview of the approach. A user icon in the top right of a step indicates user interaction.

architect may thus reuse existing components or use newly crafted components when designing the system to meet the requirements.

2. **Choose configuration to analyze:** If the model contains variability points leading to different configurations of a software product line, the user needs to choose a single configuration to be analyzed. The approach then automatically derives a system model for that particular configuration such that the succeeding steps of the presented approach do not need to care about the feature modeling specifics at all.

3. **Choose prediction approach:** The user needs to choose the performance analysis approach that shall be used as performance analysis approaches differ in their execution speed, their result precision, or other factors. Currently, the discrete event simulation implemented in SimBench (Birken et al., 2010) is integrated into the tooling, as well as the performance prediction infrastructure provided by Palladio — currently focusing on the Layered Queuing Network solver (Koziolek et al., 2008).

4. **Transform to prediction model:** Depending on the chosen analysis approach, the IET$_S$^{3} model is programmatically transformed into a model required by the particular approach. In case of the Palladio performance analyzers, the IET$_S$^{3} model is transformed into an equivalent instance of the Palladio component model. The concepts and semantics of the IET$_S$^{3} model are retained in the PCM although slight differences in the two modeling languages exist. The transformation is detailed in Chapter 5.

5. **Execute prediction approach**: The prediction approach is automatically executed on the transformed target representation of the system model. The results of the approach are then normalized leveraging the fastpan performance abstraction (see Chapter 3) before being lifted to the respective parts of the IET$_S$^{3} model.

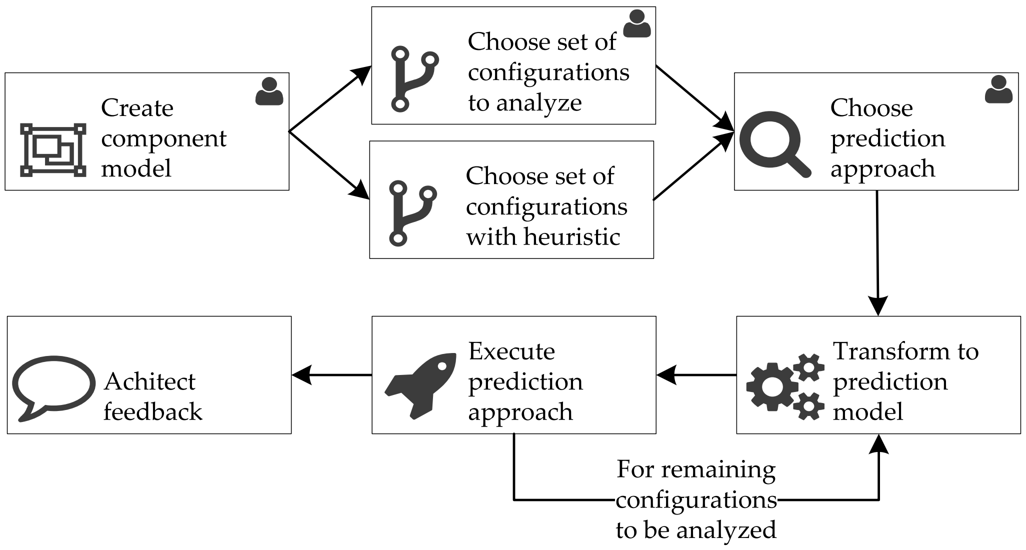

Create component model
Choose set of configurations to analyze
Choose set of configurations with heuristic
Choose prediction approach
Transform to prediction model
Execute prediction approach
Achitect feedback
For remaining configurations to be analyzed

Figure 4.2: Conceptual overview of the variability-aware performance analysis. A user icon in the top right of a step indicates user interaction.

6. **Architect feedback:** The results of the prediction are visualized near the affected parts of the model in $\text{IET}_S{}^3$. Depending on the analysis, the user may be able to see additional details upon interaction with the prediction result. Such an interaction may include showing additional details of a result of a single node or filtering certain results in order to provide the user with the precise information needed.

4.2 Variability-aware Performance Analysis

Feature modeling typically increases the number of valid configurations exponentially with the number of available features. Hence, the performance prediction can usually only be performed for a subset of all valid configurations. The configurations to analyze can either be chosen by the user or by leveraging heuristics. In either case, the presented approach is able to aggregate the performance analysis results in a way that is capable of answering the performance question the user posed in the first place.

Figure 4.2 depicts the workflow for variability-aware performance prediction. A notable change includes that instead of a single configuration to analyze a set of configurations has to be determined. While the user may manually choose a set of configurations to analyze, heuristics may be more suitable in helping the user find an answer to his question. If, for example, the user might wants to know: “Which configuration of the following incomplete feature model has the best response time for service X?”, a heuristic could pick promising feature configurations that fulfill the constraints the user asks for.

The present approach is easily extendable to implement such heuristics, albeit no such heuristics were implemented in the scope of this thesis.

When a set of prediction configurations to be analyzed is chosen, the user needs to choose and configure the analysis approach to use. The approach then analyzes all specified configurations and aggregates their individual results according to how the user configured the approach. The user might be interested in detailed performance results for each configuration, or might only be concerned about the utilization of a particular resource. The approach implemented in this thesis supports to either gather all performance results created by the individual configurations or to limit the scope to a single model element to be analyzed.

4.3 Tooling Infrastructure and Integration

To predict the performance of a system the Palladio approach to software performance prediction is employed (Koziolek et al., 2008). Palladio is built on the Eclipse platform and provides a graphical user interface to model and analyze the software system. The PCMs itself is built on the Eclipse modeling framework (EMF), inheriting the EMF tool and runtime support. IETS^3 is built on the JetBrains Meta Programming System platform, which is not built on the Eclipse platform nor has built-in support for the EMF. Hence, a fundamental step of the approach presented in this thesis is to technically integrate the Palladio tooling environment into the IETS^3 environment.

As MPS is based on a JetBrains IDE, the Eclipse environment required to run Palladio is not available. The Eclipse environment is very important to Palladio, as it is used to load and register required modules or to resolve files. To the best of our knowledge, Palladio does currently not provide a headless runner to run analyses without a graphical user interface. Hence, the ability to run Palladio analyses without a graphical user interface outside Eclipse environments was added to our open-source libraries.

To achieve this, several projects have emerged in this thesis as depicted in Figure 4.3:

- **Palladio Bridge**: The Palladio Bridge project provides all Palladio dependencies repackaged as Maven project. The project fetches the latest Palladio version from the Palladio Eclipse update site and bundles Palladio such that it can be declared as Maven dependency (*Palladio Headless Bridge Project*).
- **Palladio Environment**: The Palladio Environment project simulates an Eclipse environment tailored towards the Palladio project. It mocks certain services to ensure Palladio works as expected. The project contains technical quirks to be

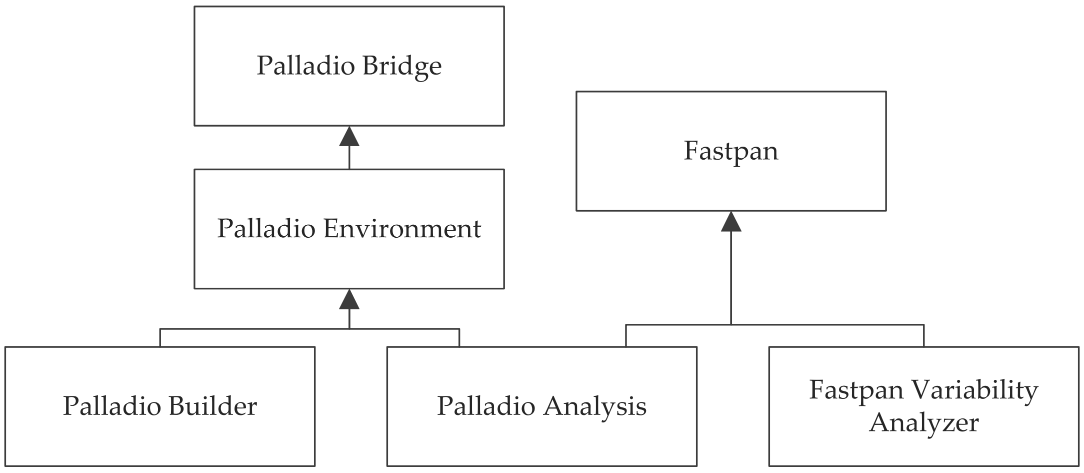


Figure 4.3: Overview of the projects developed for the approach presented in this thesis. The arrows indicate a dependency.

able to run Palladio without OSGi and without the Eclipse resource loaders (Keller, 2016b).

- **Palladio Builder**: The Palladio Builder provides a fluent Java builder API to construct PCM models on the fly. Essentially, the API creates the Palladio EMF models for a PCM instance and ensures the individual objects are properly referenced (Keller, 2016b).

- **fastpan**: The fastpan project provides the performance analysis abstraction presented in Chapter 3. The performance abstraction provides interfaces and classes to implement a performance analysis approach for a specific target model (Keller, 2016a).

- **Palladio Analysis**: The Palladio Analysis project provides an implementation of the fastpan abstraction for the Palladio component model. The Palladio analyses are executed headless, i.e., without the Eclipse environment running. Currently, only the Palladio PCM2LQN analysis leveraging layered queuing networks is supported but the approach is extensible to support other approaches as well.

- **fastpan Variability Analyzer**: The project implements an approach to analyze a variability-aware model by analyzing individual configurations. The implemented approach is based on decision trees (Valov, 2014) and is able to use any fastpan implementation (such as the Palladio Analysis project) to analyze the configurations (*fastpan Variability Analyzer Project*).

All of the above projects contribute to introducing performance awareness in $\text{IET}_S{}^3$. First, a fastpan performance analyzer is defined for the $\text{IET}_S{}^3$ model. The performance analyzer uses the *Palladio Builder* project to conduct a model-to-model transformation from the IET$_S$^{3} model to the Palladio component model. Once transformed, the Palladio component model is passed to the existing solver infrastructure with the help of the *Palladio Analysis* project, particularly using the simulations and the PCM2LQN transformation (Koziolek et al., 2008). The existing Simbench performance analyzer has also been wrapped in a fastpan performance analyzer. Both analyzers are capable of lifting the performance results of their analysis to the various model elements of the IET$_S$^{3} model. The strongly-typed performance results obtained by a fastpan performance analyzer are then used to render the results in the user interface of IET$_S$^{3}.

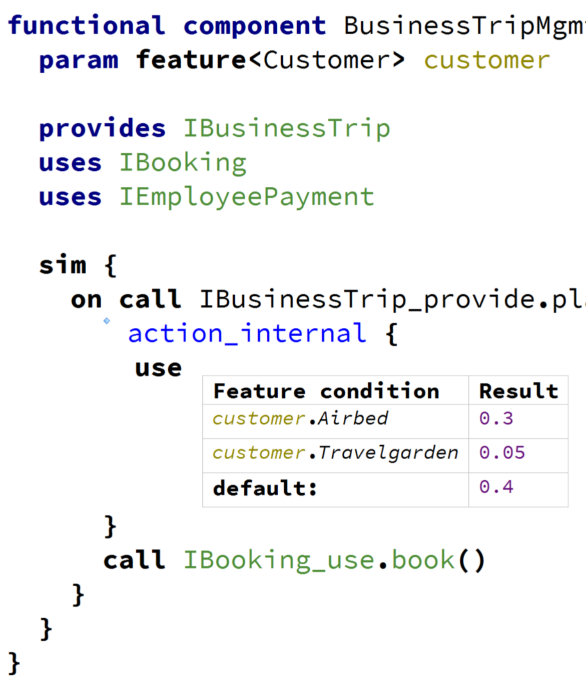
4.4 Performance Awareness in IET$_S$^{3}

With the Simbench language extension, IET$_S$^{3} treats the definition of performance behavior of components as first-class citizen. Simbench provides an extensive language to express performance demands of components. Figure 4.4 shows an example resource definition for a component. Resource demands are defined for certain triggers of the component denoted by the `on <trigger> <operation>` notation. The provided example defines a resource demand for a regular component `call` which are either provided or required by the component. The resources demands are then organized in milestones, which are named blocks that relate certain resource demands (`action_internal` in the example). Multiple milestones specify sequential resource demands. After all milestones complete, the resource demand may specify to call other triggers, as in the example the `plan` operation calls the `book` operation. Milestones contain actual resource demands, such as reading from or writing to resources, or processing time. The example contains a variant-specific processing time demand, yielding a different processor usage depending on which customer is selected in the configuration.

4.4.1 Defining performance requirements

Modeling the performance behavior is an important part of the modeling language, but equally important is the definition of performance requirements such that the fulfillment of the requirements can automatically be determined. The existing early-stage requirements specification of Simbench has been enhanced in this thesis to express performance requirements of any kind.

Figure 4.5 shows an example of how requirements are specified. A requirement may be specified by up to six aspects, of which currently the first three are implemented:



```
functional component BusinessTripMgmt {
  param feature<Customer> customer

  provides IBusinessTrip
  uses IBooking
  uses IEmployeePayment

  sim {
    on call IBusinessTrip_provide.plan {
      action_internal {
        use
```

Feature condition	Result
customer.Airbed	0.3
customer.Travelgarden	0.05
default:	0.4

```
      }
      call IBooking_use.book()
    }
  }
}
```

Figure 4.4: A sample component in the IET$_S$^{3} notation defining the performance behavior of a single operation with variant specific resource demands.

```
usecase defaultUsageScenario for MySystem {
  trigger BusinessTripMgmt.initial_trigger
}
scenario Scenario1 for MySystem {
  use cases
    defaultUsageScenario
  requirements
    The metric UTILIZATION at [ BusinessTripMgmt.initial_trigger() :: call(IBooking_use.book())
        ] must not exceed an average of 80.
    The metric UTILIZATION at [ Server3 ] must not exceed an average of 20.
}
```

Figure 4.5: A sample scenario and use case definition in the IET$_S$^{3} notation specifying performance requirements that need to be met for all use cases of the defined scenario.

1. A specific performance measure such as the utilization needs to be constrained. The available quantities relate to the quantities defined in the fastpan framework.

2. The requirement targets a specific model element. Of course, the selectable model elements should be elements that are annotated by performance analysis approaches.

3. A (numeric) target value of the requirement, such that it is possible to verify whether the requirement is met or not.

4. An indication of whether the requirement is classified as MUST-have, SHOULD-have, or NICE-to-have. Currently, all requirements are assumed to be MUST-haves.
5. Target a specific performance metric such as the average or median. Currently, all requirements are assumed to be an average metric.
6. Lifetime constraints that refine the requirement over time. For example, the requirement “allow a spike utilization above 60% only once in every X minutes”. Currently, all requirements are assumed to hold all the time.

With the given features, the requirements language is expressive enough to define realistic requirements. Enriching the requirements language with the further features is regarded future work. Also, not all performance analysis approaches support the evaluation of timing-specific behavior - a powerful requirements language limits potential analysis approaches from being able to verify the requirements.

4.4.2 Configuring the performance analysis

When the architect has finished modeling the components, the component connections, the allocation onto hardware and typical use cases, the performance analysis can be configured. The Simbench language provides a `system` concept to define performance analyses, as shown in Figure 4.6. The analysis is defined for a certain system (the root component instance) using a certain allocation onto hardware (the partitioning) under a certain scenario. If needed, a feature model to analyze can explicitly be set using the `feature models` attribute. In the scope of this thesis, the system concept was enhanced to support a set of configurations to analyze under the `configurations` attribute, as an integral part of the contribution of this thesis is to support the performance analysis of software product lines. After configuring the analysis, a performance analyzer can be started from the context menu of the system concept.

Two performance analyzers were implemented in the scope of this thesis: a performance analyzer based on Palladio that analyzes a single system and visualizes the result thereof, and a variability-aware analyzer being capable of analyzing and visualizing a set of system configurations. The implemented visualizations completely rely on the performance result abstraction of the fastpan framework (Chapter 3). Hence, any performance analysis approach can be implemented and easily integrated into the existing eco system. The result visualization and user interactions implemented are detailed in the following sections.

```
system SimpleTacticsSimulation {
  feature models
    feature<SimpleTacticsVariants> fm = Expedia_HighPerformance;

  configurations
    configuration Expedia_High {
      fm = Expedia_HighPerformance
    }
    configuration Airbnb_Mid {
      fm = Airbnb_MidPerformance
    }
    configuration Expedia_Entry {
      fm = Expedia_EntryPerformance
    }

  root instance MySystem(fm = fm)

  partitioning Allocation
  scenario Scenario1
}
```

Figure 4.6: The `system` concept is used to define a performance analysis.

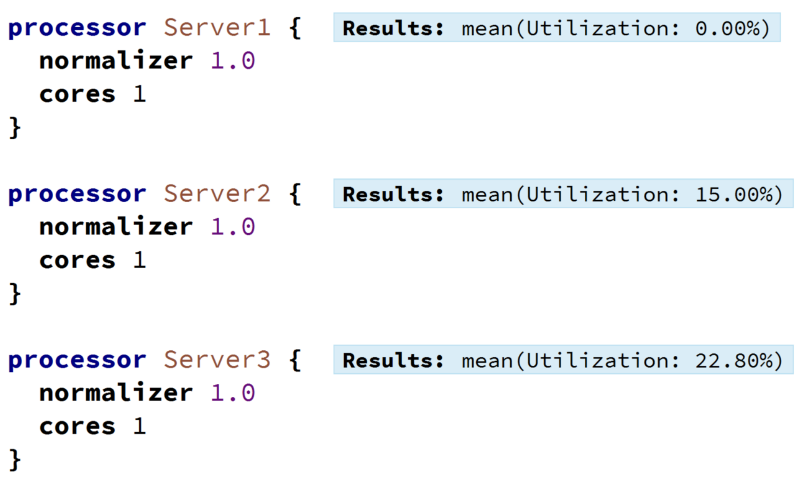
4.4.3 Regular Analysis Results

A regular performance analysis analyzes a single system which is obtained by a single configuration if feature modeling is used. In case of the Palladio analysis, the system is transformed to a Palladio component model (detailed in Chapter 5) and then analyzed by the Palladio LQN solver. The results of the Palladio analysis are expressed with the fastpan framework and are lifted to the IETS^3 model with tracing information stored during the model-to-model transformation.

Figure 4.7 shows the visualization of the performance results of a single analysis. The results are rendered to the right of the affected element and are highlighted with a background color to make it easy for users to visually distinguish parts of the model and the analysis results. The results show the actual performance measure of the result, which in this case is the utilization, and the performance metric the measure is expressed in; in this particular case the shown utilization is a mean value.

Requirements Fulfillment

The approach presented in this thesis allows to automatically determine whether requirements are fulfilled or not. The current implementation of the requirements verification checks whether the specified model element has an attached performance result of the specified metric (currently only the `Average` metric) and of the specified type. Hence, the verification of a single requirement may yield three different states: the requirement

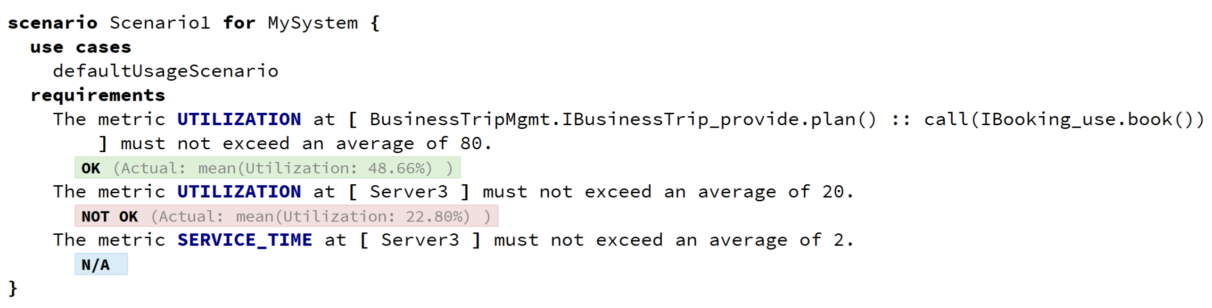


```
processor Server1 {    Results: mean(Utilization: 0.00%)
  normalizer 1.0
  cores 1
}

processor Server2 {    Results: mean(Utilization: 15.00%)
  normalizer 1.0
  cores 1
}

processor Server3 {    Results: mean(Utilization: 22.80%)
  normalizer 1.0
  cores 1
}
```

Figure 4.7: Single performance result visualization. Shows the mean utilization that is predicted for each of the processors.



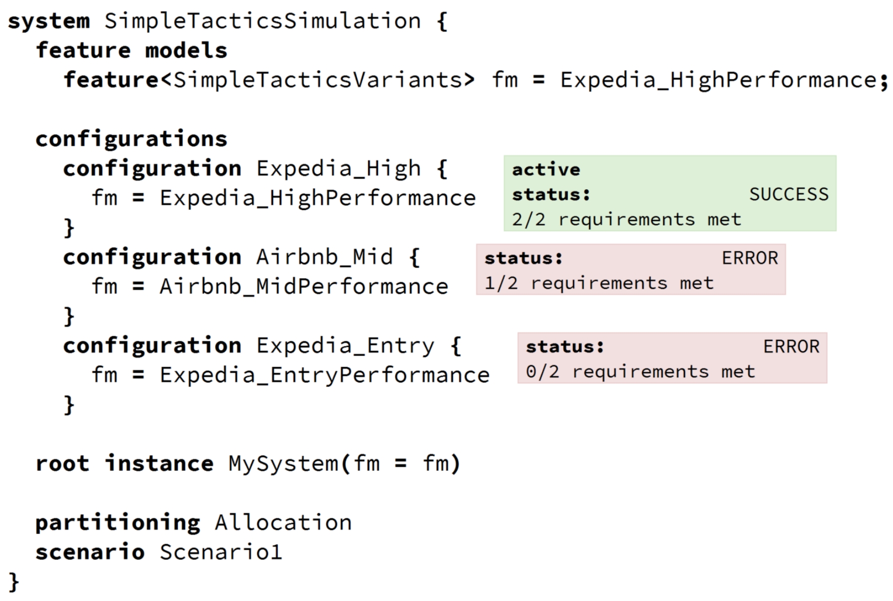
```
scenario Scenario1 for MySystem {
  use cases
    defaultUsageScenario
  requirements
    The metric UTILIZATION at [ BusinessTripMgmt.IBusinessTrip_provide.plan() :: call(IBooking_use.book())
        ] must not exceed an average of 80.
      OK (Actual: mean(Utilization: 48.66%) )
    The metric UTILIZATION at [ Server3 ] must not exceed an average of 20.
      NOT OK (Actual: mean(Utilization: 22.80%) )
    The metric SERVICE_TIME at [ Server3 ] must not exceed an average of 2.
      N/A
}
```

Figure 4.8: The performance analysis result allows to automatically verify whether each requirement is fulfilled or not.

is fulfilled, the requirement is not fulfilled, or it is not decidable whether the requirement is fulfilled or not.

Figure 4.8 shows an example how all three requirement fulfillment states are visualized below the requirements definition. If available, the actual result is rendered next to the state so that the user immediately sees the difference between the requirement and the performance result. The actual value gives the user an impression of how close the requirements are fulfilled or not fulfilled.

Future work may include a more sophisticated requirements checker. For example, a performance result specifying a result using a maximum metric that is lower than the requirement specified as average metric surely is fulfilled, but the current approach does not handle these cases.



```
system SimpleTacticsSimulation {
  feature models
    feature<SimpleTacticsVariants> fm = Expedia_HighPerformance;

  configurations
    configuration Expedia_High {      active
      fm = Expedia_HighPerformance    status:                SUCCESS
    }                                 2/2 requirements met
    configuration Airbnb_Mid {      status:              ERROR
      fm = Airbnb_MidPerformance    1/2 requirements met
    }
    configuration Expedia_Entry {     status:              ERROR
      fm = Expedia_EntryPerformance   0/2 requirements met
    }

  root instance MySystem(fm = fm)

  partitioning Allocation
  scenario Scenario1
}
```

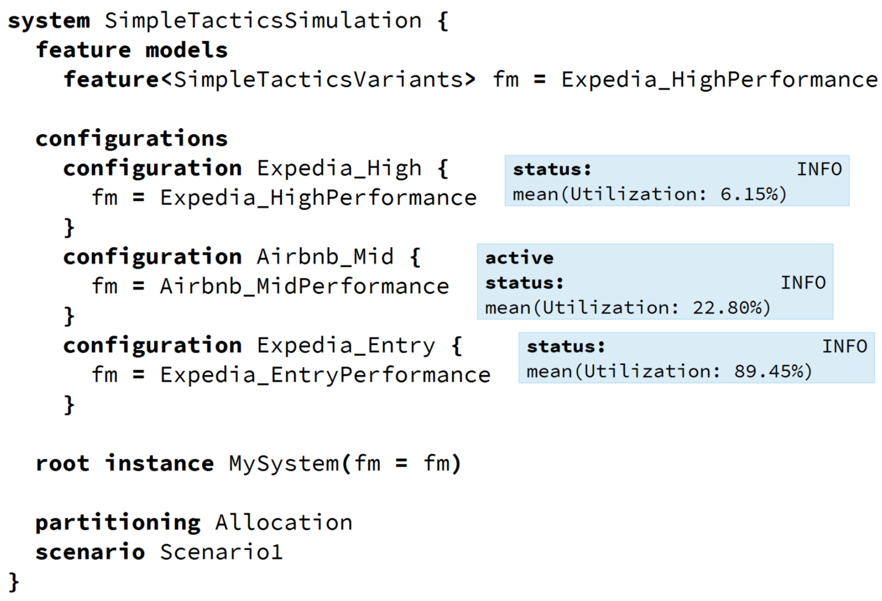
Figure 4.9: Requirements fulfillment overview for all configurations of the performance analysis. The boxes can expose a state (e.g. good) to help the user with the summary interpretation.

4.4.4 Variability Analysis Results

The variability analysis typically analyzes a set of configurations. Annotating the results of all configurations to all model elements would clutter the user interface with results and be of no use to the user. Furthermore, a variability analyzer may not even produce specific performance results for the configurations, but instead determine feature combinations that lead to a low/high performance. As the visualization of the variability analysis result may be highly specific to the analysis approach, in most cases a tailored visualization makes the most sense. However, the approach implemented in this thesis defines a common, reusable and flexible visualization format and user interaction concept for variability analysis results.

Results for a single configuration are presented next to the configuration definition as depicted in Figure 4.9. The displayed box provides the following information to the user:

1. **Description**: The description contains a human-readable summary of the analysis result. The description should not be too long, but contain the most important aspects of the result.
2. **State**: The box can show a state helping the user interpret the summary. Currently, the state supports the following severities: `INFO` for indicating an informational



```
system SimpleTacticsSimulation {
  feature models
    feature<SimpleTacticsVariants> fm = Expedia_HighPerformance;

  configurations
    configuration Expedia_High {        status:                    INFO
      fm = Expedia_HighPerformance      mean(Utilization: 6.15%)
    }
    configuration Airbnb_Mid {        active
      fm = Airbnb_MidPerformance      status:                      INFO
    }                                 mean(Utilization: 22.80%)
    configuration Expedia_Entry {        status:                      INFO
      fm = Expedia_EntryPerformance      mean(Utilization: 89.45%)
    }

  root instance MySystem(fm = fm)

  partitioning Allocation
  scenario Scenario1
}
```

Figure 4.10: Performance result of a single node in comparison for all analyzed configurations. The result being displayed is selected by the user.

result, WARNING to indicate that the users attention is required, ERROR to indicate that the analysis has revealed an error or SUCCESS to indicate that the analysis has revealed no error.

3. **Active**: The box has an active flag indicating whether details of the selected configuration are currently visualized elsewhere.

As such, the box result visualization imposes no assumptions on the analyzer and is thus reusable. An additional benefit is that the user of the system is accustomed to the particular graphical representation of variability results and has a smaller learning curve when using other analyzers.

The default variability analysis of the presented approach uses any existing performance analyzer to analyze the system for each configuration. Hence, the analysis in fact yields a regular performance result for each configuration. Using those results, Figure 4.9 shows the visualization of the summarized requirements fulfillment state of all analyzed configurations. The box has the SUCCESS state if all requirements are fulfilled, an ERROR state if at least one requirement is not fulfilled and a WARNING state if the fulfillment of at least one requirement is not decidable for the configuration.

An intention[1] can be used to show the result details of a single configuration, which loads the visualization of the system as if it was analyzed with a regular performance

[1]In MPS, an intention is like a keyboard-triggered context menu

analysis (as described in Section 4.4.3). However, the visualization then has an additional user interaction: when a performance result of a single node is selected (see Figure 4.7) the user can focus the result of the specific node in the configuration overview by using an additional intention. Figure 4.10 visualizes the performance result of a selected node for all analyzed configurations. With that, the user can easily compare the performance result of a single node (e.g. the utilization of a processor) across all analyzed configurations.

Chapter 5

Model-to-Model Transformation

IETs$_S^3$ provides several language concepts to express component specifications, system architectures built with the components, planned allocations onto hardware, and usage scenarios for the designed system. On top, IETs$_S^3$ provides a feature model that tightly integrates with the aforementioned concepts to express variability points on all aspects of the model. As the IETs$_S^3$ DSL provides all information required by Palladio to conduct a performance prediction for the system, a model-to-model transformation of the IETs$_S^3$ meta-model to the Palladio component model has been implemented to leverage the performance prediction features Palladio provides.

This chapter highlights selected parts of the current implementation of the approach, in particular the model-to-model transformation from the IETs$_S^3$ model to the Palladio component model. The transformation is built using the MPS Java language, which contains language enhancements tailored to building DSLs with MPS. An open-source builder API[1] to construct Palladio component models from Java has been built to ease the writing of the transformation. As the models of IETs$_S^3$ and Palladio are not semantically identical, a mapping to retain the semantic meaning of IETs$_S^3$ in the transformed PCM instance was implemented. The source code of the implementation is available in the provided supplementary material (Keller, 2016c).

5.1 Transformation Overview

The model-to-model transformation is split into multiple steps as shown in the step dependency graph in Figure 5.1. The steps correspond to the top-level entities defined in the `PCMInstance` class, which serves as the root object of a PCM instance. Due to

[1]https://github.com/DECLARE-Project/palladio-headless

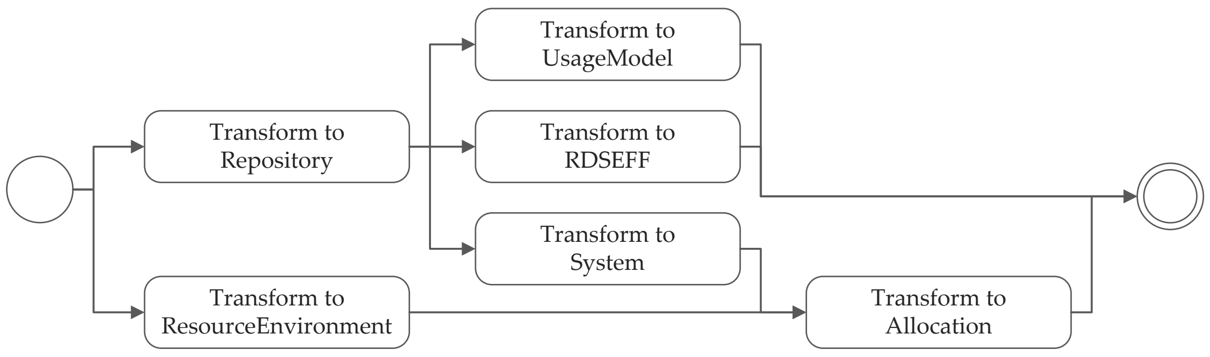


Figure 5.1: The different steps of the $\text{IET}_S{}^3$ to PCM transformation and their dependencies.

the complexity of the repository and RDSEFF transformation the two transformations have been split into separate steps, but are both part of a `Repository` in the PCM instance.

The dependency graph indicates that a parallel transformation of the different steps is possible. As the transformation has not yet been a limiting factor w.r.t execution time the current implementation does not leverage parallelized processing of the steps. However, during the design of the transformation architecture special care has been taken to allow a parallelization of the steps in the future.

The transformation itself is built using the blackboard design pattern. Transformed elements are stored in shared context objects relating to the different transformation steps inside a "blackboard" object. By accessing the context objects through the blackboard, the transformators can read from and write to the respective context objects as required. After all steps have completed the transformed PCM instance can be retrieved from the blackboard.

For each step, multiple transformators can be registered which are then executed in the sequence of their registration. Thus, the chosen architecture enables to define a basic mapping to support a limited subset of $\text{IET}_S{}^3$ that is very similar to the PCM meta-model which is detailed in Section 5.2. The limited subset comprises enough modeling concepts to conduct a valid LQN analysis in Palladio. Advanced modeling concepts of $\text{IET}_S{}^3$ are then supported by chaining additional transformators, such as the support of the `SimTrigger` concept as described in Section 5.3 shows.

Table 5.1: Mapping of $\text{IET}_\text{S}{}^3$ data types to corresponding data types of Palladio.

$\text{IET}_\text{S}{}^3$ Data Type	Mapped Palladio Data Type
BooleanType	Bool
StringType	String
RealType	Double
IntegerType	Int

5.2 Basic Transformation

As a baseline, a basic transformation of $\text{IET}_\text{S}{}^3$ concepts that are very similar to the PCM meta-model has been implemented. The transformation yields a valid and analyzable Palladio instance, as the transformation to all important modeling concepts of a PCM instance are supported. The following sections detail each transformation step, respectively.

5.2.1 Transform to Repository

The PCM repository hosts the component definitions, including the operations the components provide or require. In $\text{IET}_\text{S}{}^3$, components are connected by ports of a certain type. The `ServicePortType` is the most basic port type and represent a connection between components connecting components through `ServiceDefinitions`. A `ServiceDefinition` is an interface that describes a collection of available operations with their parameters and a return type. Though the object hierarchy seems fairly complex, a one-to-one mapping of the concepts between the $\text{IET}_\text{S}{}^3$ model and the PCM model is possible as seen in Figure 5.2.

$\text{IET}_\text{S}{}^3$ ships with a sophisticated and extensible expression language that has various primitive and complex data types. All operation definitions in $\text{IET}_\text{S}{}^3$ may use all available types to define parameters or return types, such as functions, maps, records, or enums. As a proper mapping to Palladio's data types is impossible for some of the types, only a subset of all available data types is currently supported. In particular, Table 5.1 summarizes the mapping of all supported primitive types.

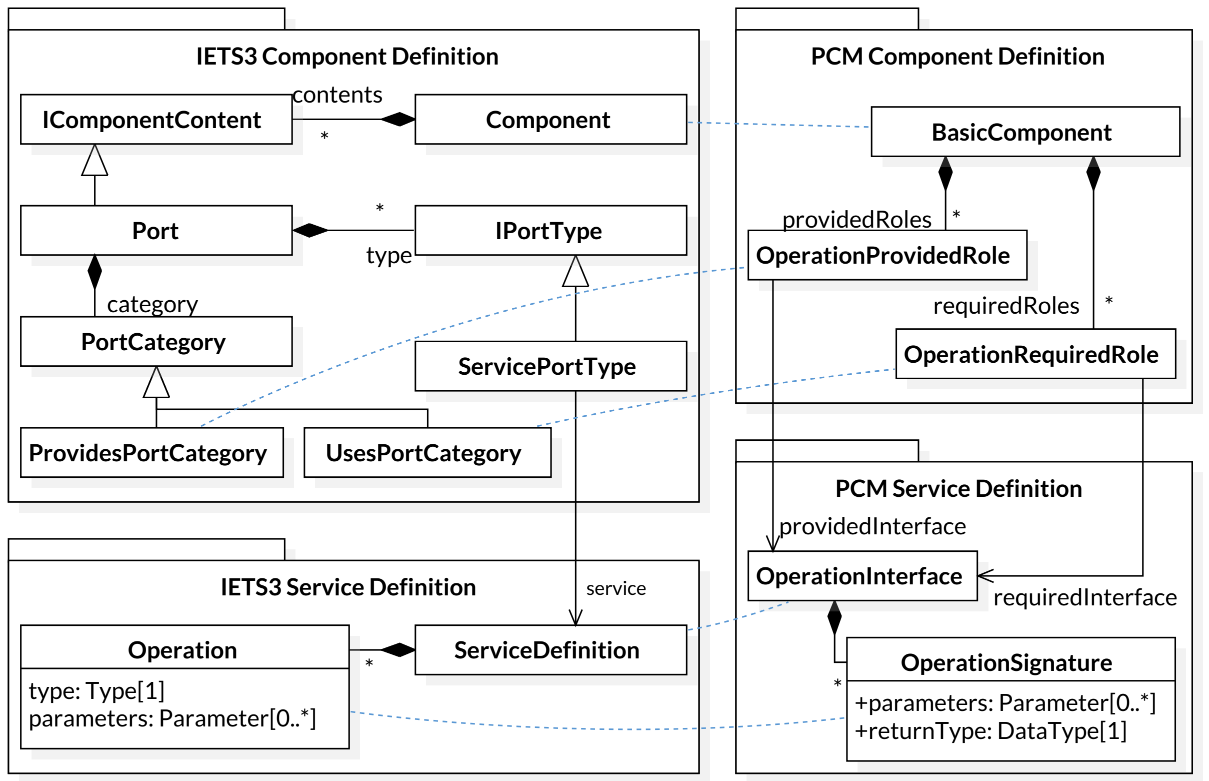


Figure 5.2: Mapping of ordinary components defined in IETS3 to the Palladio component model.

5.2.2 Transform to ResourceEnvironment

Transforming the resource environment is a crucial part of the model-to-model transformation, as retaining the semantics of the IETs^3 resources in a PCM is required to gain useful results from the performance analysis. IETs^3 currently has three different concepts to express a resource environment: *processors* to model computational resources, *resources* to model bandwidth-limited resources (e.g., a hard disk) and *pools* to model resources with a fixed upper usage limit (e.g., RAM).

Processors

A processor in IETs^3 is defined by two values: a normalization factor (i.e., the processing speed) and a discrete number of cores that are capable of processing tasks in parallel. As depicted in Figure 5.3, each processor is mapped to a Palladio `ResourceContainer` with a `ProcessingResourceSpecification` having the `numberOfReplicas` set to the corresponding value of the IETs^3 model. The `processingRate` is fixed to 1 despite the normalization factor modeled in the IETs^3 model, as the factor is already

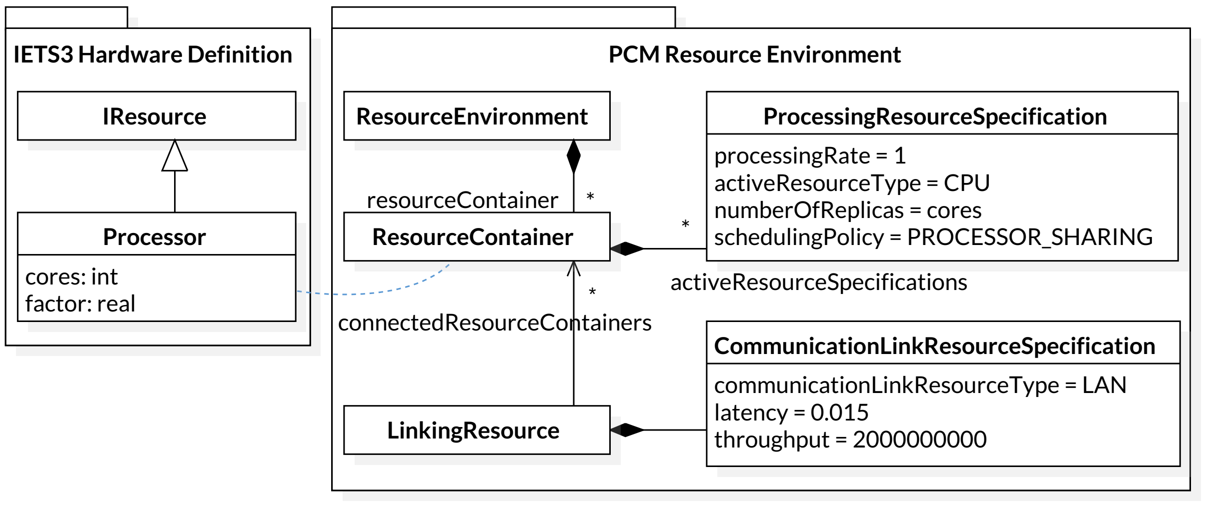


Figure 5.3: Mapping of a CPU in IET$_S$^{3} to a Palladio `ResourceContainer`.

taken into account when calculating the actual resource demands of the IET$_S$^{3} model. Hence, all IET$_S$^{3} resource demands are normalized w.r.t. the factor of the processor they are deployed on and thus, the corresponding Palladio resource containers must all have the same `processingRate`. By default, the `activeResourceType` is set to `CPU` and the `schedulingPolicy` to `PROCESSOR_SHARING` as the IET$_S$^{3} model assumes that processing power is evenly split between all pending tasks. Note that the IET$_S$^{3} model assumes that all components can communicate with each other, regardless on which processor they are deployed. To transform this assumption to the PCM model, an artificial `LinkingResource` with the default values specified in Figure 5.3 is created that connects all transformed `ResourceContainer`s.

Resources

IET$_S$^{3} resources are currently not supported by the model-to-model transformation, as the IET$_S$^{3} resources have modeling capabilities that cannot be mapped to Palladio resource containers:

- **Induced load**: IET$_S$^{3} resources model an induced load as percentage, that induces extra load on the processor that is currently accessing the resource. The induced load is an important modeling concept and used in real-world models to account for extra processing power required for resources access, e.g., to model access to a resource with on-the-fly compression of the data. While such a modeling capability is missing from the Palladio component model, it would be possible to support the induced load by generating intermediate components to handle the resource access, that create the induced load by specifying respective resource demands.

- **Context switching time**: $\text{IET}_S{}^3$ resources model a context switching time as percentage, that induces extra load on accessing processors if and only if the resource is being accessed concurrently. Palladio currently has no such modeling capabilities and a model construct accounting for the context switching time is not feasible in the PCM.
- **Multiple interfaces**: $\text{IET}_S{}^3$ supports multiple interfaces to a single resource, each specifying its own bandwidth, induced load and context switching time. Semantically, the multiple interfaces access the same underlying resource and thus compete for the processing power of the resource. Multiple interfaces are used for example, to model a flash memory resource with a *standard* interface, a *direct IO* interface and a *compressed* interface, each parametrized differently to take their actual speeds into account. Palladio currently has not capabilities to model multiple interfaces for a single resource and hence, this feature could only be supported by generating intermediate components to handle the resource access by offering operations corresponding to the different interfaces and issuing the required resource demands.

While it would be possible to account for some of the $\text{IET}_S{}^3$ resource modeling capabilities in the corresponding Palladio component model, the transformation would create complex generated model elements in the PCM that are still not able to fully retain the semantics. Due to this, the transformation was not yet implemented during this thesis, as it is more beneficial to investigate the use cases of the resources in real-world projects and see how the transformation can help architects in making good decisions. If feasible, a viable alternative would be to use a simplified transformation of the $\text{IET}_S{}^3$ resources and accept some loss in precision of the result.

5.2.3 Transform to RDSEFF

Palladio resource demanding service effect specification (RDSEFF) allow to model the resource consumption and behavior of the system components. Hence, when constructing the RDSEFF it is important to preserve the semantic behavior of the components defined in $\text{IET}_S{}^3$. Figure 5.4 provides an overview of the mapping described in this section.

In $\text{IET}_S{}^3$ the performance-related behavior of a component is described in the `SimulationBehavior`, which is denoted by the `sim` keyword as seen in the example in Figure 5.5. Each simulation behavior may contain an arbitrary number of behaviors, which are the corresponding elements to the `ResourceDemandingSEFF`s of Palladio. A behavior comprises four aspects: *triggers* that determine when the behavior

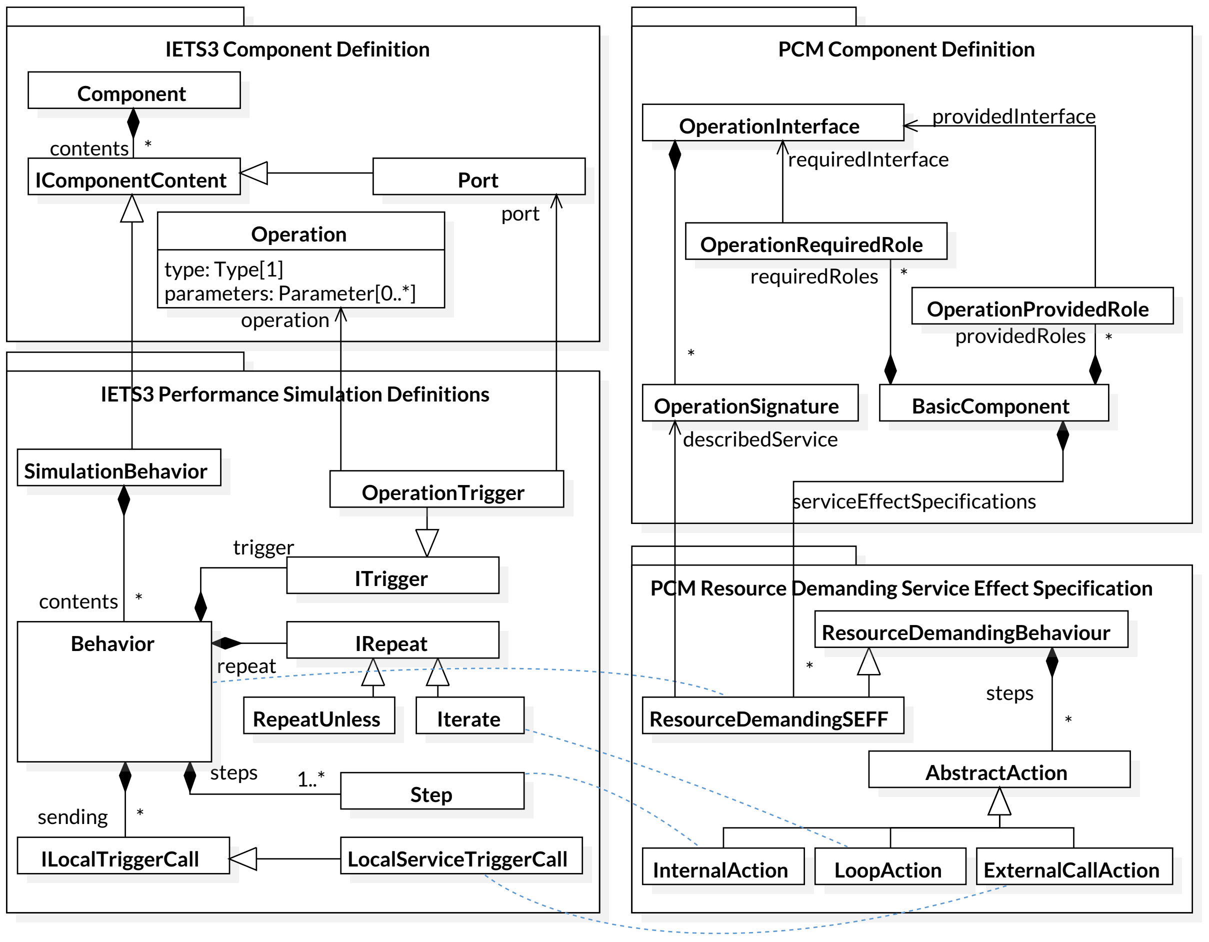


Figure 5.4: Mapping of simulation behavior in IET$_S$^{3} to a Palladio RDSEFF. The dashed lines denote corresponding model elements.

is executed, an optional *repeat* declaration for looped behaviors, *steps* modeling the actual resource demands, and *sending* declarations to invoke other trigger definitions.

Behavior Triggers

The simulation behavior is triggered by a certain type of trigger, with the `OperationTrigger` representing a regular one-directional call to an operation of a service definition.

Behavior Repeats

The simulation behavior specification in IET$_S$^{3} supports loops only on a complete behavior definition. The `Iterate` loop definition iterates for a specified number of times and can directly be mapped to the Palladio `LoopAction`.

```
functional component BookingSystem {
  provides IBooking
  uses IExternalPayment

  sim {
    on call IBooking_provide.book {
      action_internal {
        use 0.3
      }
      call IExternalPayment_use.pay()
    }
  }
}
```

Figure 5.5: Example IET$_S$^{3} resource demand, specifying a processor demand and a call to another component.

The `RepeatUnless` declaration repeats the modeled behavior until a certain predefined trigger was called. As there is no `AbstractAction` in Palladio available that has a similar behavior, the `RepeatUnless` loop is currently not supported.

Behavior Steps
The `Step` concepts contain the actual procedural control flow logic of the behavior. Most aspects defined in the steps can be mapped to `InternalActions` with a resource demand, for example the `UseAction` models CPU consumption, and `ReadAction` or `WriteAction` model the access to a bandwidth-limited resource, such as a hard disk.

Behavior Sending
A sending declaration is a call to another trigger, i.e., it invokes the behavior modeled in another simulation behavior. A `LocalServiceTriggerCall` is a call to another `OperationTrigger` which can be mapped to an `ExternalCallAction` in the Palladio RDSEFF.

The example in Figure 5.5 defines a behavior step with a processing power demand of 0.3 and a behavior sending call to another component through the used interface of the component.

5.2.4 Transform to System

The system structure models the connections between the actual instances of components. In IET$_S$^{3} the instances and their connections are defined in the `ComponentSubstructure` inside of a component. Any component may contain a substructure which allows to model a hierarchical component structure. The transformed

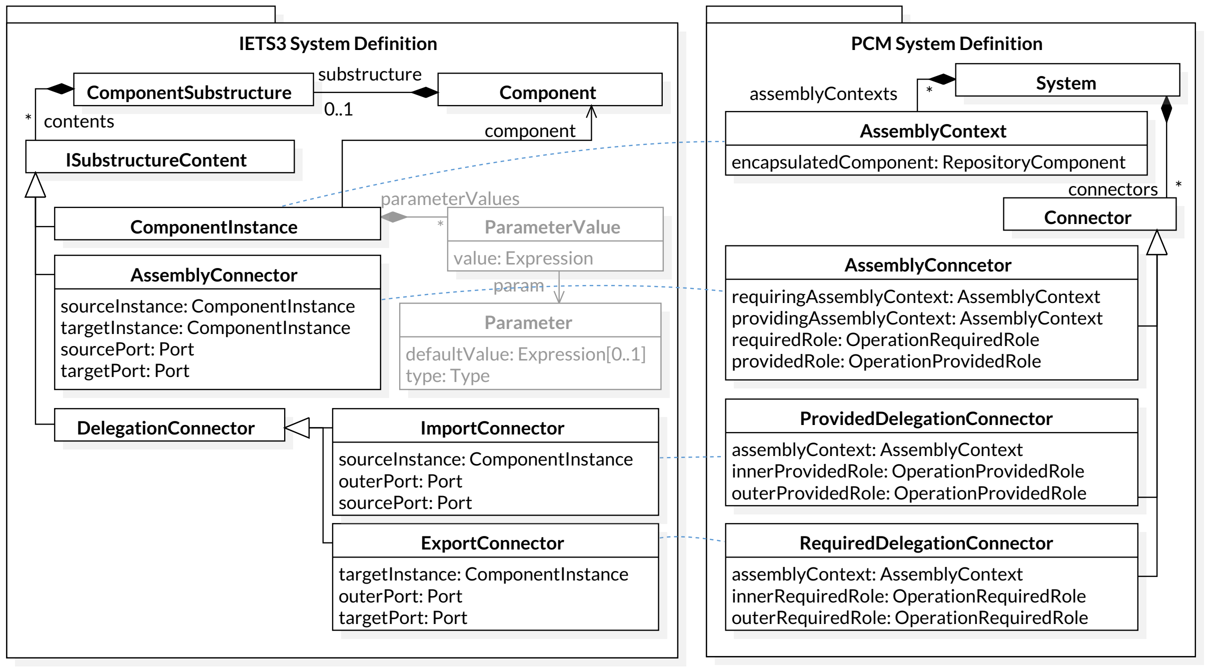

IETS3 System Definition
ComponentSubstructure
substructure
0..1
Component
* contents
component
ISubstructureContent
ComponentInstance
parameterValues
*
ParameterValue
value: Expression
param
AssemblyConnector
sourceInstance: ComponentInstance
targetInstance: ComponentInstance
sourcePort: Port
targetPort: Port
Parameter
defaultValue: Expression[0..1]
type: Type
DelegationConnector
ImportConnector
sourceInstance: ComponentInstance
outerPort: Port
sourcePort: Port
ExportConnector
targetInstance: ComponentInstance
outerPort: Port
targetPort: Port
PCM System Definition
assemblyContexts
*
System
AssemblyContext
encapsulatedComponent: RepositoryComponent
connectors
*
Connector
AssemblyConncetor
requiringAssemblyContext: AssemblyContext
providingAssemblyContext: AssemblyContext
requiredRole: OperationRequiredRole
providedRole: OperationProvidedRole
ProvidedDelegationConnector
assemblyContext: AssemblyContext
innerProvidedRole: OperationProvidedRole
outerProvidedRole: OperationProvidedRole
RequiredDelegationConnector
assemblyContext: AssemblyContext
innerRequiredRole: OperationRequiredRole
outerRequiredRole: OperationRequiredRole

Figure 5.6: Mapping of the IET$_S$^{3} component instances to PCM's assemblies.

PCM does not retain the hierarchical structure and instead has a flattened view of the components.

Figure 5.6 show the corresponding system modeling concepts between IET$_S$^{3} and the PCM. An IET$_S$^{3} component featuring a substructure has an arbitrary number of `ComponentInstance`s that are mapped to a PCM `AssemblyContenxt`. Both concepts reference their respective component definition. In IET$_S$^{3}, a component definition may define required or optional typed parameters, that can be accessed from various expressions from within the component. The value of the parameters are either set to a default value or can be specified when defining the component instance using the `ParameterValue` specification. Currently, the parameters are primarily used to pass feature models to the component such that feature-specific behavior can be expressed within the component definition. However, the parameter concept is not limited to that: it is certainly possible to use the parameters within resource demand expressions. While such a parametrized resource demand could easily be mapped to the PCM model, there are expressions that are not easily ported to the PCM, such as `LambdaExpressions` to define lambda functions. To support such a versatile parameter concept it would be necessary to know whether the parameters can and should be evaluated before transforming to the PCM or whether the parametrization concept should be retained in the transformed PCM. As this depends on the intended use cases, the parameter concept is not yet supported in the model-to-model transformation.

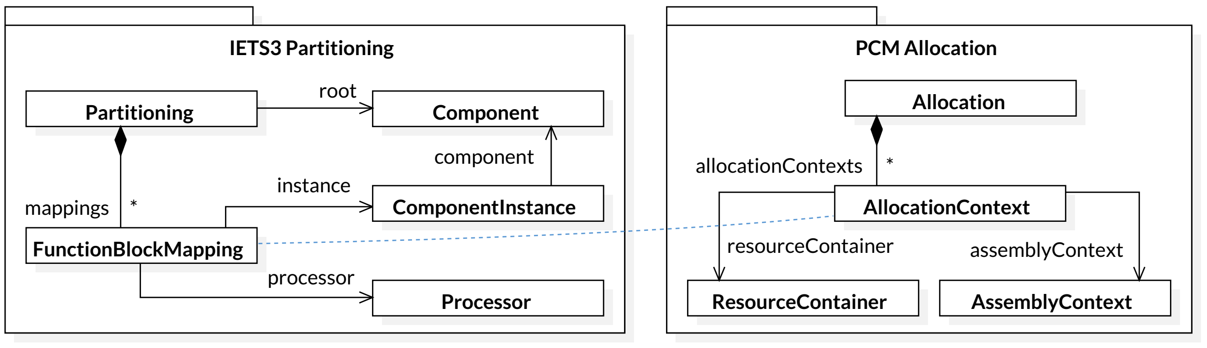


Figure 5.7: Mapping of the IET$_S$^{3} partitioning to the PCM allocation concept.

The IET$_S$^{3} component substructure needs to further define the connections between the component instances. All connections correspond to a `ServiceDefinition` representing the interfaces that need to match to make a connection. To connect two components over a common interface an `AssemblyConnector` is used both in IET$_S$^{3} and the PCM. The component hosting the component substructure can provide or require services definitions itself. These provided or required services are provided to or required from the “outer” system and can thus be mapped to the corresponding concepts of the PCM.

5.2.5 Transform to Allocation

The IET$_S$^{3} partitioning maps component instances to the available processors. Due to the hierarchical structure of the component model, the partitioning may also assign any nested components to a different processor than any of its parents. As the component model in the transformed PCM is flattened, the Palladio allocation can easily be mapped as shown in Figure 5.7.

An IET$_S$^{3} `FunctionalBlockMapping` assigns a single component instance to a single processor. The concept can be mapped to a Palladio `AllocationContext` referencing the respective `AssemblyContext` of the component instance and `ResourceContainer` of the processor.

5.2.6 Transform to UsageModel

The usage model defines system usage scenarios for which the performance is relevant. In IET$_S$^{3} there are two relevant concepts to define the behavior. First, a `UseCase` is attached to the system’s root component and defines triggers that invoke certain behavior

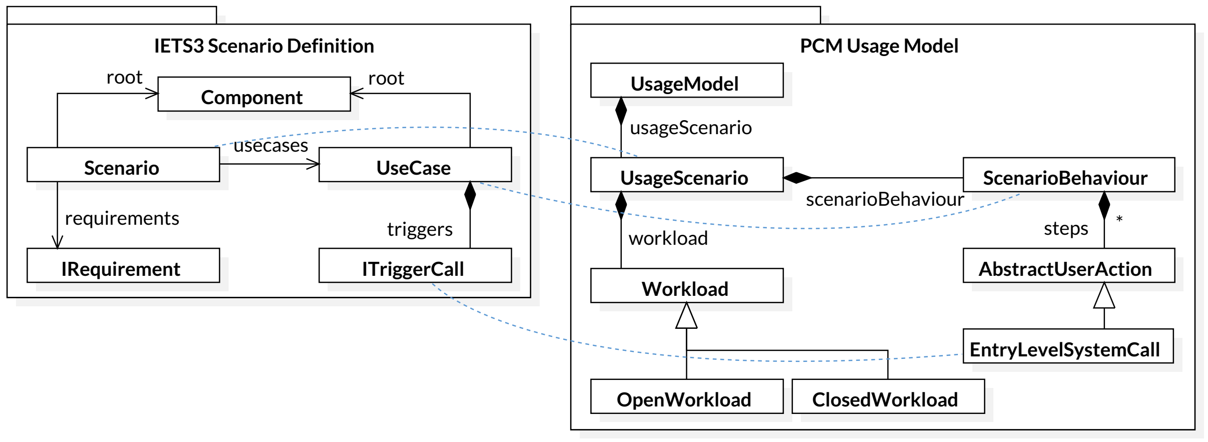


Figure 5.8: Mapping of the $\mathrm{IET_S}^3$ scenario to the PCM usage model concept.

of the system. A set of use cases are then grouped to a scenario that defines performance requirements which need to be met for all use cases.

Figure 5.8 shows the conceptual mapping of the $\mathrm{IET_S}^3$ usage model to the PCM usage model. PCM supports equivalent concepts for the scenario and uses cases, which are directly mapped to. The PCM `ScenarioBehaviour` is more powerful in terms of describing the scenario behavior, as next to calling services it also supports loops, branches and delays. As those are not present in the $\mathrm{IET_S}^3$ model, all $\mathrm{IET_S}^3$ trigger calls are directly mapped to a PCM `EntryLevelSystemCall`. In order to construct a valid PCM, the operations referenced by the `EntryLevelSystemCall` need to be properly provided to the system in the system transformation step.

5.3 Supporting SimTriggers

Resource demands of regular component interactions are defined with the *on call* keyword and are directly mapped to Palladio's RDSEFF. In addition to that, the *trigger* concept allows components to define arbitrary communication and invocation paths regardless of their structure and hierarchy in order to model environmental events and their performance impact (such as the sudden re-routing while playing radio in a car infotainment system). During the transformation the triggers are mapped to artificial PCM interfaces which are connected appropriately in the resulting PCM system. The variable resource demand is evaluated for a specific variant before transforming to the PCM, such that the feature-specific resource demand is a regular numeric value during the transformation.

```
functional component DB {
  sim {
    on trigger scaleOut {
      slowerQueries {
        use 20
      }
      << ... >>
    }
  }
}
functional component Payment {
  sim {
    on trigger highFrequencyPayments {
      incoming {
        use 1000
      }
      << ... >>
    }
  }
  instance DB
}
functional component Booking {
  sim {
    on trigger christmasSale {
      highLoad {
        use 12000
      }
      trigger Payment.highFrequencyPayments
      trigger Payment.DB.scaleOut
    }
  }
  instance Payment
}
```

Figure 5.9: Example usage of the IETS^3 SimTrigger concept with a hierarchical component organization.

The transformation of the SimTrigger concept is done in four steps that are intermingled with the steps of the basic transformation described in Section 5.2. Figure 5.9 is used as running example in this section to describe the transformation. Figure 5.9 contains three components with a hierarchical organization: `DB`, `Payment`, and `Booking`. As seen from the example trigger calls in the `Booking` component, the SimTrigger concept can call nested components of arbitrary depths. Notably, the triggers do not require any services to be able to call other triggers. As the SimTrigger concept is transformed to a regular Palladio component model, several interfaces and their required and provided roles need to be generated by the transformation.

Step 1: Transform SimTrigger declarations to PCM interfaces

The first step of the transformation involves the generation of an interface for each SimTrigger definition (represented by `on trigger <name>` in the example). The generated interface contains a single operation without arguments called `trigger`, which merely allows the interface to be called by other components. Finally, the component containing the SimTrigger definition provides the generated interface as if it was a regular interface of the component. In the example shown in Figure 5.9, the triggers `scaleOut`, `highFrequencyPayments`, and `christmasSale` yield a generated interface.

Step 2: Transform SimTrigger calls to PCM required roles

This transformation step is looking for the components that need to require the interfaces generated in the first step. As the triggers are implicitly available everywhere, this step scans all SimTrigger bodies of all components for `trigger` calls (e.g., the `Booking` component calls a nested SimTrigger via `trigger Payment.DB.scaleOut`). As the target of the call is a SimTrigger definition, the transformation retrieves the generated interface of step 1 for the SimTrigger and adds the interface as required interface to the component issuing the trigger call. The transformation takes care to require the interface only once for each component. In the example shown in Figure 5.9, only the `Booking` component has trigger calls and hence, requires the two generated interfaces referenced by the call.

Step 3: Transform each SimTrigger body to a PCM RDSEFF

The body content of each SimTrigger has the same structure and semantics as a regular call definition. Hence, the contents of the SimTriggers are transformed to RDSEFFs identical to a regular component call. As in this step, all component already provide and require the generated interfaces, the trigger calls can be transformed to regular Palladio `ExternalCallActions`.

Step 4: Connect the generated required and provided interfaces in the assembly

As a last step, the modeled system needs to have all the required and provided interfaces generated during the SimTrigger transformation to be connected. Hence, the transformation adds a connection between each valid pair of required and provided interfaces. In the example shown in Figure 5.9, the transformation connects the `Booking` component to the `Payment` component, as well as the `Booking` component to the `DB` component through the generated interfaces.

Chapter 6

Quantitative Evaluation

To evaluate research question RQ4 a set of experiments was conducted. The experiments measure the execution time of the approach, from triggering the analysis to rendering the results in the user interface. The following times were measured:

- **Total time**: The total time it takes for the approach from triggering the analysis to rendering the results in the user interface.
- **SimModelWrapper time**: Prior to the model-to-model transformation, a `SimModelWrapper` class prepares the IET$_S$^{3} model for the transformation. As the preparation is non-trivial, it's execution time was measured.
- **Model-to-model transformation time**: The time it takes for the IET$_S$^{3} model to be transformed to a PCM instance by the developed model-to-model transformation.
- **Analysis time**: The time it takes for the underlying performance prediction approach (e.g., the Palladio LQN solver) to predict the performance of the model.

Each individual experiment was repeated twelve times in a row and the first two results were discarded to prevent a possible ramp-up time from skewing the measurement results. Table 6.1 shows the hardware and software specifications of the laptop on which the evaluation was conducted.

The models generated for the analysis are all generated using the same procedure from an architecture definition. The architecture definition models a component with three parameters: the name, the number of occurrences and a list of dependencies (other components names). A list of those definitions forms the evaluation architecture. From such a list, the actual IET$_S$^{3} model is constructed as follows:

Table 6.1: Hardware and software specifications of the laptop the evaluation was conducted on.

Evaluation Environment Specifications	
Hardware	
Processor (CPU)	Intel(R) Core(TM) i7-4700MQ CPU at 2.40 GHz per core
Memory (RAM)	16 GB
Hard Disk Drive	256 GB SSD
Software	
Operating System (OS)	Windows 10
Java	1.8.0
JetBrains MPS	3.3.5

1. **Generate components**: Generate a new component for each occurrence. That is, if there is a component named `db`, with an occurrence of three times, then three artificial components are generated: `db_01`, `db_02`, and `db_03`. Each component is then allocated exactly once.
2. **Generate service**: Create a new service with a single operation for each required dependency of a component. Then the component defining the dependency requires the service while the dependency itself provides the service. This is repeated for all occurrences of the component. For example, a component `app` with two occurrences depending on the component `db` with two occurrences yields six different services.
3. **Create service calls**: The components are connected by defining resource demands for the provided services by first, adding a CPU resource demand of 0.01 and then calling all dependent services of all occurrences.

In addition to the components, the evaluation architecture defines a single entry component that is being called first.

6.1 Linear Evaluation

The linear evaluation uses an artificially generated chain of components that is parametrized by the number of components in the chain. Hence, the number of components, the number of services and the number of service calls scales linearly with the parameter. Clearly, the modeled system does not resemble the architecture of a realistic

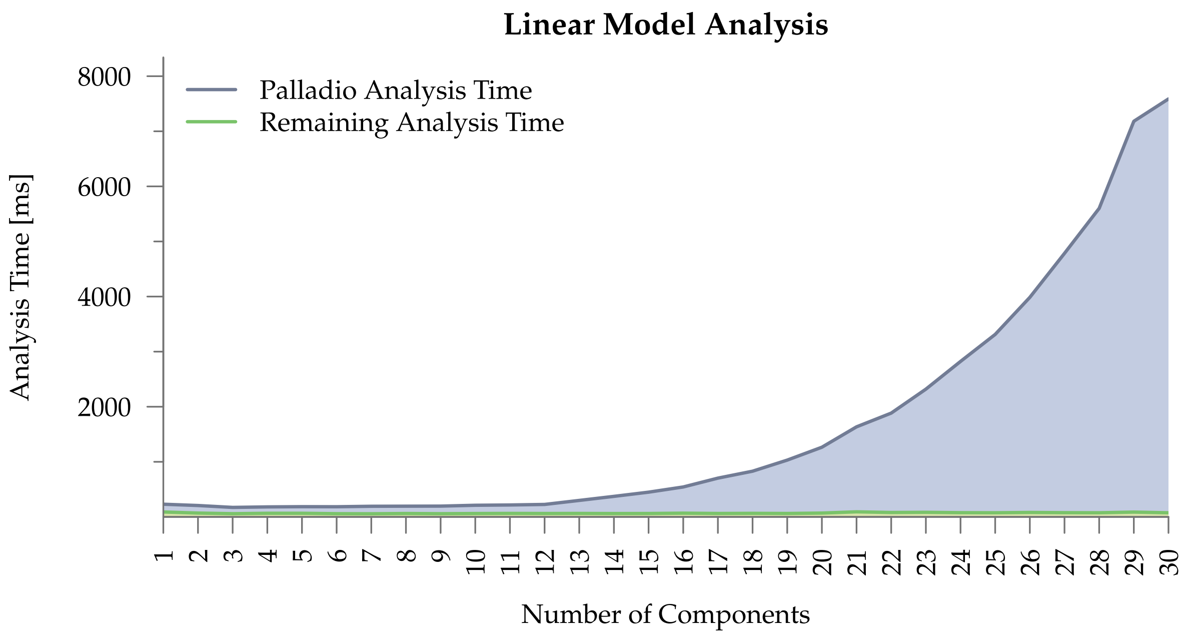


Figure 6.1: Median execution time of ten repeated runs of the linear evaluation model.

system in an appropriate way. However, the evaluation determines the impact of the model complexity on the execution time of the approach developed in this thesis and is thus suitable as an indicator whether the performance awareness approach is executing fast enough.

Figure 6.1 shows the median execution time of the approach developed in this thesis for the linear evaluation model parametrized by the number of components contributing to the model. The execution times shown are the median values of all ten measured execution times. The plot shows a stacked line chart, where the blue, upper line is the fraction of the total time spent in the Palladio LQNS solver and the bottom, green line shows the remaining time spent in the developed approach, comprising the SimModelWrapper time and the model-to-model transformation time.

The plot indicates, that the performance awareness approach developed during this thesis has a constant overhead for the tooling infrastructure and model-to-model transformation. With an increasing model size, the Palladio analysis shows a more than linear increase in processing time until the model consists of 30 unique components. With the given analysis configuration, the Palladio LQNS solver was not able to analyze linear models consisting of more than 30 components.

To answer research question 4, the developed approach is able to execute a performance analysis within ten seconds, but the execution time depends on the model size, strongly increasing for more than 20 components.

6.2 Simianviz Evaluation

In contrast to the linear evaluation, the Simianviz evaluation aims to evaluate the scalability of the approach developed in this thesis using realistic architectures. The architectures used for the evaluation are taken from the Simianviz[1] project, which aims to simulate monitoring data for realistic microservice architectures. By seeing each microservice as a component in the context of this thesis, the architectures of the Simianviz project minimize the threat to external validity as they have been manually curated by a domain expert.

The Simianviz architecture definitions have a similar format to the architecture definition used for this analysis. Hence, the architecture definitions can directly be mapped and preserve their connection characteristics. To analyze the scalability, the architecture definitions were parameterized by a single parameter n, indicating the maximum number of occurrences of a single component. Hence, all components having a higher number of occurrences than the parameter n are scaled down to the parameter.

Figure 6.2 shows the performance analysis times of the different architectures for different values of n. The architecture names refer to the names of the architecture JSON files contained in the Simianviz tool. Each name is followed by a bracket (a, b) where a denotes the number of components in the derived system b denotes the number of connections (i.e., service definitions) between the components.

The plot shows that the median analysis time of all architectures is below 500 ms, while the slowest two analysis runs take around 10 seconds to complete. In general, the time it takes to analyze a certain system has a very low variance and thus, the execution time of the approach is very reliable.

The parameter has been set to $n = 1$, $n = 2$ and $n = 3$, but not all architectures derived from the parameters were analyzable. For $n = 1$ all eleven architectures were analyzable, for $n = 2$ only 6 architectures were analyzable, and for $n = 3$ there were 5 architectures left. When increasing n, the architecture definitions contain more components and connections leading to a model that is too large for the Palladio LQN solver, similar to the upper bound detected in the linear evaluation (see Section 6.1). Hence, some of the generated models are not analyzable and are thus not included in the evaluation.

[1]https://github.com/adrianco/spigo

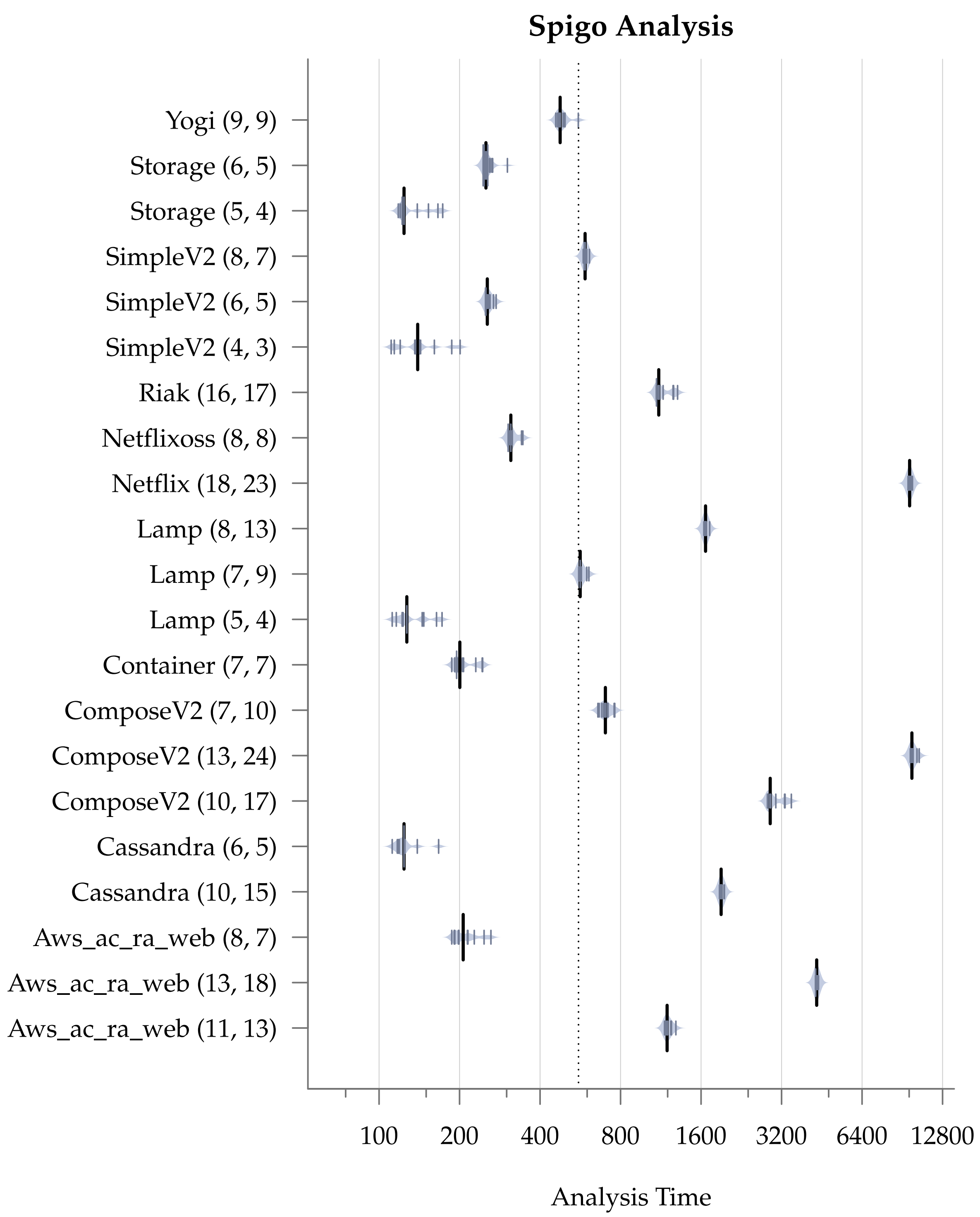


Figure 6.2: Execution times of ten repeated runs of the Simianviz evaluation models.

Chapter 7

Conclusions and Future Work

7.1 Summary and Discussion

This thesis provides a proof-of-concept implementation for introducing performance awareness in the IET$_S$^{3} integrated specification environment. This section summarizes the findings of this thesis and discusses the research questions posed in the introduction.

Research Question 1: How can various model-based performance prediction approaches be abstracted to a uniform interface to leverage different analysis result characteristics?

The research questions aims to bridge the technical gap between different performance analyzers in order to build a technical foundation for declarative performance engineering. Chapter 3 describes the technical abstraction of the developed abstraction that is used to integrate different performance analyzers. The developed abstraction provides a means for integrating regular performance analyzers, as well as variability-aware performance analyzers. To reason about performance results, the developed abstraction provides a set of well-defined performance measures allowing performance results to be strictly typed.

While the performance measures are straightforward to express analysis result of the performance analysis for a single system, the variability-aware part of the abstraction does not define a uniform result interface. This limitation is due to the fact, that the results of a variability-aware analysis are manifold: the result could be a collection of individual analysis results for each configuration, but could also be a set of features that have a strong negative impact on the performance of the system.

As a proof-of-concept implementation of the abstraction the Palladio performance analysis tool suite is integrated into the $\text{IET}_S{}^3$ environment leveraging the abstraction. Chapter 5 describes the model-to-model transformation implemented to transform the $\text{IET}_S{}^3$ model to the Palladio component model in order to leverage the Palladio performance analyses.

Research Question 2: How can model-based performance predictions be integrated in the $\text{IET}_S{}^3$ specification environment to provide feedback to an architect while designing the software system?

Providing feedback to the architect is a very important aspect of performance awareness, as visual interfaces help people to perceive and filter relevant information. Chapter 4 details how the performance analyses are integrated into the $\text{IET}_S{}^3$ specification environment. The result visualization for a single performance analysis enriches two model elements with result annotations. First, the affected model elements are annotated with their respective performance results, similar to the profiling result visualization described by Beck et al. (2013). Second, the performance requirements are automatically verified with the actual performance results and are annotated to visually distinguish met and unmet performance requirements. The visualizations provide the required information at the exact location of the affected model elements. Hence, a user of the system has a minimal cognitive overhead in locating and associating the results and instead can focus on important tasks, such as interpreting the results or finding viable alternatives.

Currently, the expressiveness of the requirements is limited to only specifying *average* thresholds. While the average value of a performance measure is certainly not able to cover the requirements of a real-world system, future work should investigate typical requirements of real-world projects. However, performance analyzers may only be capable of providing certain metrics, such as the average value, in their analysis results. Thus, the analysis results may not be able to be deduced to verify overly complex requirements.

Research Question 3: How can model-based performance predictions be used to provide feedback to an architect while designing software product lines?

Software product lines (SPLs) define a set of software systems, namely one for each valid configuration. A key difficulty of designing software product lines is that only some of the valid configurations may not fulfill performance requirements (e.g., due to unforseen side-effects of feature interactions). Hence, it is of high interest to help architects design SPLs that meet all performance requirements. Chapter 4 shows how variability-aware performance analyzers are integrated into the user interface of $\text{IET}_S{}^3$ and how user interactions are used to raise the performance awareness in the specification environment. The approach includes a language extension to the $\text{IET}_S{}^3$ modeling language to define a set of configurations of the SPL to analyze. The list of the configurations to analyze is then used to visualize the results of the variability-aware analysis. By default, the visualization shows how many requirements are met for each configuration. Through user interaction, it is possible to show a single performance measure of a model element for all configurations. For example, the architect is then able to see the mean utilization of a processor for all analyzed configurations and can immediately see which configurations may have a concerning performance.

Research Question 4: Is the approach capable of providing real-time performance awareness?

As performance awareness is concerned about fast feedback cycles a fast execution time of the analysis is preferred. Chapter 6 evaluates the performance and scalability of the approach developed in this thesis. Chapter 6.1 evaluates the scalability on a controlled model of linearly chained components, with 20 components analyzed in less than 2 seconds and 30 components analyzed in under 8 seconds. Chapter 6.2 minimizes threats to external validity by evaluating real-world component architectures provided by Simianviz. The median of median analysis times of all analyzed architectures is around 0.5 seconds.

The turnaround time for the approach developed in this thesis is perfectly suited for real-time analysis that retriggers by user-intended changes to the model. A limitation of the approach is the small number of components (less than 30) a system may consist of to remain analyzable. The limitation may originate from the way the evaluation models are built or from the chosen parametrization of the Palladio tool suite. Nevertheless, the underlying LQN solver by Franks et al. (2005) can only solve models containing up to 1000 entries, tasks, processors and entries per task, respectively. Typically, the architectural model elements defined in $\text{IET}_S{}^3$ and the transformed PCM expand to multiple LQN model concepts, leading to models that quickly reach the LQN solver limits. To improve the scalability either the solver can be improved to cope with larger models or techniques can be designed to solve only relevant parts of the model, e.g., by hierarchical decomposition.

7.2 Future Work

This section provides pointers to possible future work directions starting from the research done in this thesis. The future work section is divided into short-term future work and long-term future work.

7.2.1 Short Term Future Work

Short-term future work focuses on improving the existing approach.

Model-to-model transformation: The model-to-model transformation currently supports a basic set of $\text{IET}_S{}^3$ and PCM features. The model-to-model transformation can be enhanced to support more sophisticated feature of both models to better retain the semantics of the $\text{IET}_S{}^3$ model in the transformed version.

Analysis Results: The user interface to display and interact with analysis results can be improved by additional views and interactions. The views may include tabular or graphical displays of the results with enhanced filtering and browsing concepts. In addition to that, the results could be integrated with existing approaches to modeling software performance measures (e.g., MAMBA by Frey et al. (2011)).

Performance Requirements: The performance requirements currently have a limited expressiveness. Future work can improve the expressiveness of the results language by supporting enhanced notations (e.g., for distributions), and also improve the algorithm to determine whether a requirement is fulfilled or not.

7.2.2 Long Term Future Work

Long-term future work focuses on extending the existing approach by new concepts and ideas.

Industry-scale user study: The evaluation conducted during this thesis was performed by the author of the thesis and is inherently limited in it's validity. A systematic evaluation of the approach w.r.t. performance and scalability using an industry-scale user study is desirable. In addition to that, a controlled experiment can reveal how users actually benefit from the developed approach.

Variability analyzers: The developed approach provides only a rudimentary variability analysis. To improve the performance evaluation and awareness support for software product lines existing techniques from other contexts may be leveraged (Guo et al., 2013; Valov, 2014; Zhang et al., 2016).

Automatic Selection: A transparent selection of model-based and measurement-based evaluation techniques are of high interest to improve the usability (Walter et al., 2016). Heuristics can help choosing an approach to optimize the end-user experience in terms of the trade-off between prediction precision and analysis runtime (Walter et al., 2016).

Incremental analysis: To improve the real-time feedback loop supporting incremental performance predictions is of high interest, as users typically only change small parts of the model (Szabó et al., 2016) and require immediate feedback for the changed parts.

Appendix

Bibliography

Beck, Fabian, Oliver Moseler, Stephan Diehl, Günter Daniel Rey (2013). "In situ understanding of performance bottlenecks through visually augmented code." In: *Proc. ICPC '13*, pp. 63–72 (cit. on pp. 22, 23, 68).

Becker, Steffen (2008). "Coupled model transformations for QoS enabled component-based software design." PhD thesis. Universität Oldenburg (cit. on p. 15).

Becker, Steffen, Heiko Koziolek, Ralf H. Reussner (2009). "The Palladio component model for model-driven performance prediction." In: *J. Syst. Software* 82.1, pp. 3–22 (cit. on pp. 1, 2, 8, 9).

Birken, Klaus, Daniel Hünig, Thomas Rustemeyer, Ralph Wittmann (2010). "Resource Analysis of Automotive/Infotainment Systems Based on Domain-specific Models: A Real-world Example." In: *Proc. ISoLA'10, Part II*. Heraklion, Crete, Greece, pp. 424–433. ISBN: 3-642-16560-5, 978-3-642-16560-3 (cit. on pp. 2, 18, 34).

Brunnert, Andreas, André van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, et al. (2015). "Performance-oriented devops: A research agenda." In: *arXiv preprint arXiv:1508.04752* (cit. on p. 21).

Cheesman, John, John Daniels (2001). "UML components." In: *EUA: Addison-Wesley* (cit. on p. 7).

Danciu, Alexandru, Alexander Chrusciel, Andreas Brunnert, Helmut Krcmar (2015). "Performance Awareness in Java EE Development Environments." In: *Proc. EPEW '15*, pp. 146–160 (cit. on p. 22).

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, et al. (2013). "The State of the Art in Language Workbenches." In: *Proc. SLE '13* (cit. on p. 15).

Fontenot, Michael L. (1989). "Software congestion, mobile servers, and the hyperbolic model." In: *IEEE Transactions on Software Engineering* 15.8, pp. 947–962 (cit. on p. 14).

Franks, Greg, Peter Maly, Murray Woodside, Dorina C Petriu, Alex Hubbard (2005). “Layered queueing network solver and simulator user manual.” In: *Dept. of Systems and Computer Engineering, Carleton University (December 2005)* (cit. on pp. 15, 69).

Franks, Greg, Tariq Al-Omari, Murray Woodside, Olivia Das, Salem Derisavi (2009). “Enhanced modeling and solution of layered queueing networks.” In: *IEEE Transactions on Software Engineering* 35.2, pp. 148–161 (cit. on p. 14).

Frey, Sören, André van Hoorn, Reiner Jung, Wilhelm Hasselbring, Benjamin Kiel (2011). “MAMBA: A Measurement Architecture for Model-Based Analysis.” In: *Department of Computer Science, University of Kiel, Germany, Tech. Rep. TR-1112, Dec* (cit. on p. 70).

Fricke, Ernst, Armin P Schulz (2005). “Design for changeability (DfC): Principles to enable changes in systems throughout their entire lifecycle.” In: *Systems Engineering* 8.4 (cit. on p. 6).

Guo, Jianmei, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, Andrzej Wasowski (2013). “Variability-aware performance prediction: A statistical learning approach.” In: *Proc. ASE ’13*, pp. 301–311 (cit. on pp. 20, 70).

Ho, Chih-Wei, Michael J Johnson, Laurie Williams, E Michael Maximilien (2006). “On agile performance requirements specification and testing.” In: *AGILE 2006*. IEEE (cit. on p. 5).

Hoorn, Andre van, Christian Vögele, Eike Schulz, Wilhelm Hasselbring, Helmut Krcmar (2014). “Automatic Extraction of Probabilistic Workload Specifications for Load Testing Session-Based Application Systems.” In: *8th International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2014)* (cit. on p. 8).

Horký, Vojtech, Peter Libic, Lukás Marek, Antonín Steinhauser, Petr Tuma (2015). “Utilizing Performance Unit Tests To Increase Performance Awareness.” In: *Proc. ICPE ’15*, pp. 289–300 (cit. on pp. 2, 22).

Jain, Raj (1990). *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons (cit. on pp. 27, 29).

Jetbrains MPS. http://www.jetbrains.com/mps (cit. on p. 15).

Johnson, Pontus, Johan Ullberg, Markus Buschle, Ulrik Franke, Khurram Shahzad (2013). “P2amf: Predictive, probabilistic architecture modeling framework.” In: *International IFIP Working Conference on Enterprise Interoperability*. Springer, pp. 104–117 (cit. on p. 22).

Kang, Kyo C, Sholom G Cohen, James A Hess, William E Novak, A Spencer Peterson (1990). *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document (cit. on p. 20).

Kang, Kyo C, Jaejoon Lee, Patrick Donohoe (2002). “Feature-oriented product line engineering.” In: *IEEE software* 19.4, p. 58 (cit. on p. 19).

Keller, Fabian (2016a). *DECLARE-Project/fastpan v1.0.0*. URL: https://doi.org/10.5281/zenodo.182153 (cit. on pp. 25, 37).

– (2016b). *DECLARE-Project/palladio-headless v1.0.0*. URL: https://doi.org/10.5281/zenodo.183038 (cit. on p. 37).

– (2016c). *Introducing Performance Awareness in an Integrated Specification Environment Supplementary Material*. URL: https://doi.org/10.5281/zenodo.183057 (cit. on p. 47).

Kolesnikov, Sergiy, Alexander von Rhein, Claus Hunsen, Sven Apel (2013). “A comparison of product-based, feature-based, and family-based type checking.” In: *ACM SIGPLAN Notices*. Vol. 49. 3. ACM, pp. 115–124 (cit. on p. 20).

Koziolek, Heiko (2008). “Parameter dependencies for reusable performance specifications of software components.” PhD thesis. Universität Oldenburg (cit. on pp. 6, 7, 15).

– (2010). “Performance evaluation of component-based software systems: A survey.” In: *Perform. Eval.* 67.8, pp. 634–658 (cit. on p. 1).

Koziolek, Heiko, Jens Happe (2006). “A qos driven development process model for component-based software systems.” In: *International Symposium on Component-Based Software Engineering*. Springer, pp. 336–343 (cit. on p. 7).

Koziolek, Heiko, Ralf H. Reussner (2008). “A Model Transformation from the Palladio Component Model to Layered Queueing Networks.” In: *Proc. SIPEW ’08*, pp. 58–78 (cit. on pp. 34, 36, 38).

Lee, Kwanwoo, Kyo C Kang, Jaejoon Lee (2002). “Concepts and guidelines of feature modeling for product line software engineering.” In: *International Conference on Software Reuse*. Springer, pp. 62–77 (cit. on p. 19).

Menascé, Daniel A (2002). “Two-level iterative queuing modeling of software contention.” In: *Modeling, Analysis and Simulation of Computer and Telecommunications Systems, 2002. MASCOTS 2002. Proceedings. 10th IEEE International Symposium on*. IEEE, pp. 267–276 (cit. on p. 14).

Northrop, Linda, P Clements (2001). “Software product lines.” In: 4.05 (cit. on p. 19).

Palladio Headless Bridge Project. https://github.com/DECLARE-Project/palladio-headless-bridge-mvn (cit. on p. 36).

Petriu, Dorina C, C Murray Woodside (1991). “Approximate MVA from Markov model of software client/server systems.” In: *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on*. IEEE, pp. 322–329 (cit. on p. 14).

Reussner, Ralf H, Iman H Poernomo, Heinz W Schmidt (2003). “Reasoning about software architectures with contractually specified components.” In: *Component-Based Software Quality*. Springer, pp. 287–325 (cit. on p. 8).

Szabó, Tamás, Sebastian Erdweg, Markus Voelter (2016). “IncA: A DSL for the definition of incremental program analyses.” In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, pp. 320–331 (cit. on pp. 22, 71).

Taylor, Richard N, Nenad Medvidovic, Eric M Dashofy (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing (cit. on p. 6).

Tůma, Petr (2014). “Performance Awareness.” In: *Proc. ACM/SPEC ICPE ’14*. Dublin, Ireland, pp. 135–136. ISBN: 978-1-4503-2733-6 (cit. on p. 2).

Valov, Pavel (2014). “Variability-Aware Performance Prediction: A Case Study.” In: (cit. on pp. 20, 37, 70).

Van Gurp, Jilles, Jan Bosch, Mikael Svahnberg (2001). “On the notion of variability in software product lines.” In: *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. IEEE, pp. 45–54 (cit. on p. 19).

Voelter, Markus (2011). “Language and IDE Development, Modularization and Composition with MPS.” In: *GTTSE*. LNCS. Springer (cit. on p. 15).

Voelter, Markus, Daniel Ratiu, Bernd Kolb, Bernhard Schätz (2013). “mbeddr: Instantiating a language workbench in the embedded software domain.” In: *Autom. Softw. Eng.* 20.3, pp. 339–390 (cit. on p. 22).

Voelter, Markus, Sascha Lisson (2014). “Supporting Diverse Notations in MPS’Projectional Editor.” In: *GEMOC MoDELS*, pp. 7–16 (cit. on p. 15).

Walter, Jürgen, Andre van Hoorn, Heiko Koziolek, Dusan Okanovic, Samuel Kounev (2016). “Asking What?, Automating the How?: The Vision of Declarative Performance Engineering.” In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, pp. 91–94 (cit. on pp. 21, 70).

Zhang, Yi, Jianmei Guo, Eric Blais, Krzysztof Czarnecki, Huiqun Yu (2016). “A mathematical model of performance-relevant feature interactions.” In: *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, pp. 25–34 (cit. on pp. 20, 70).

fastpan Variability Analyzer Project. https://github.com/DECLARE-Project/fastpan-variability-analyzer (cit. on p. 37).

All links were last followed on November 10, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature