Enabling Consistency between Software Artefacts for Software Adaption and Evolution

Master's Thesis of

David Monschein

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr. Anne Koziolek
Advisor:	Dr. rer. nat. Robert Heinrich
Second advisor:	M.Sc. Manar Mazkatli

27. January 2020 – 2. October 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Hohberg, 1. October 2020

. . . . . .

(David Monschein)

Abstract

Nowadays, software systems are evolving at a pace never seen before. In addition, there are increasingly more adaptations of the software at run-time. Consequently, many challenges arise, especially when architecture models are used to analyze the software. As a result, emerging inconsistencies between different software artifacts are almost inevitable. The effort to eliminate these inconsistencies is high and grows due to faster and faster changing software systems. Furthermore, a profound knowledge about the architecture is necessary to maintain the consistency permanently. Another problem in this context is that this knowledge is not available centrally, rather it is usually distributed among several teams or people (e.g. operators and developers).

Currently, there are several approaches for automated consistency maintenance between source code and architecture models. Furthermore, there are methods that use monitoring data to update and enrich the architecture models at run-time. However, these approaches do not consider the system composition and changes to it. Moreover, they do not perform automated self-validations or have integrated them only prototypically so far. This leads to uncertainty about the quality of the architecture models.

Therefore, in this thesis, we present a comprehensive approach for supporting the consistency preservation between software artifacts with special focus on software evolution and adaptation. It combines features and concepts of different existing approaches. At design time, source code analysis and consistency rules are used, while at run-time, monitoring data is used as input for a transformation pipeline. In contrast to already existing approaches, the automated derivation of the system composition at run-time and a semi-automated extraction at design-time are supported. Ultimately, self-validations were included as a central component of the approach. These allow to dynamically adapt the behavior based on detected inaccuracies of the models. The goal is to update the architecture models as accurately as possible and with the highest possible degree of automation.

In a case study based evaluation the accuracy of the models was measured. Reference models and monitoring data were used as a basis for comparison. It was shown that the derived models remain accurate even when considering evolution and adaptation scenarios. Subsequently, the performance of the transformation pipeline for the case studies was examined and it became clear that they are adequate for practical use. In this context, we also examined the overhead that was caused by the monitoring. It was shown that a good balance between the overhead and the accuracy of the monitoring was achieved. Finally, the scalability of the transformations within the pipeline was investigated by using synthetically generated monitoring data as input. The results showed that the scalability of the transformations is sufficient for the majority of practical use cases.

Zusammenfassung

Heutzutage entwickeln und verändern sich Softwaresysteme in einem immer schneller werdenden Tempo. Außerdem kommt es häufiger zu Adaptionsszenarien zur Laufzeit der Software. Daraus ergeben sich viele neue Herausforderungen, insbesondere wenn Architekturmodelle zur Modellierung der Software verwendet werden. Infolgedessen sind auftretende Inkonsistenzen zwischen verschiedenen Software-Artefakten nahezu unvermeidlich. Der Aufwand zur Beseitigung dieser Inkonsistenzen ist erheblich und wächst aufgrund der immer schneller evolvierenden und adaptierenden Softwaresysteme weiter. Darüber hinaus ist ein fundiertes Wissen über die Architektur notwendig, um die Konsistenz dauerhaft zu erhalten.

Es gibt bereits verschiedene Ansätze die die automatisierte Konsistenzhaltung zwischen Softwareartefakten. Des Weiteren wurden verschiedene Strategien vorgestellt, wie Monitoring Daten verwendet werden können, um die Architekturmodelle zur Laufzeit zu aktualisieren und anzureichern. Jedoch binden diese Ansätze die System-Zusammensetzung bei der Konsistenzhaltung nicht ein. Außerdem führen diese keine Selbst-Validierung durch oder haben diese bisher nur prototypisch integriert. Dadurch besteht eine Unsicherheit über die tatsächliche Genauigkeit der Modelle.

In dieser Thesis präsentieren wir einen umfangreichen Ansatz zur Konsistenzhaltung zwischen Softwareartefakten. Ein besonderes Augenmerk liegt dabei auf der Unterstützung von Evolutions- und Adaptionsszenarien. Dieser Ansatz kombiniert Funktionen und Konzepte von verschiedenen existierenden Strategien. Um die Inkonsistenzen zwischen den Modellen zu vermeiden, werden zur Entwurfszeit Quellcode-Analysen und Konsistenzregeln verwendet, während zur Laufzeit Monitoring-Daten als Input für eine Transformations-Pipeline dienen. Im Vergleich zu bereits existierenden Ansätzen wird die Ableitung von Modellen der System-Zusammensetzung zur Lauf- und Design-Zeit unterstützt. Außerdem wurden Selbst-Validierungen als zentraler Baustein in den Ansatz integriert. Auf Basis der Ergebnisse dieser Validierungen ist es möglich, bestehende Inkonsistenzen gezielt zu eliminieren.

In einer Fallstudien-basierten Evaluation wurde die Genauigkeit der Modelle ausgewertet. Dafür wurden Referenz-Modelle und die Monitoring-Daten als Vergleichsbasis herangezogen. Es wurde gezeigt, dass die abgeleiteten Modelle akkurat sind, auch wenn Evolutions- und Adaptionsszenarien betrachtet werden. Anschließend wurde die Performance der Transformations-Pipeline evaluiert. Es wurde deutlich, dass diese für die meisten praktischen Anwendungsfälle adäquat ist. In diesem Zusammenhang wurde auch der Monitoring-Overhead gemessen. Es wurde gezeigt, dass ein gutes Balancing zwischen der Genauigkeit der Monitoring-Daten und dem entstehenden Overhead erreicht wurde. Abschließend wurde die Skalierbarkeit der Transformationen untersucht. Die Ergebnisse ergaben, dass diese für die Mehrzahl der praktischen Anwendungsfälle ausreichend ist.

Contents

List of Figures

List of Tables

1. Introduction

Modern software systems become increasingly more complex over time and development iterations tend to get shorter and shorter. In most cases the software system is based on various artifacts that represent different abstractions or aspects. An example for such a pair of artifacts is the source code and related architecture models. Another problem results from the fact that in many cases the artifacts are managed and developed by different teams. This creates an overhead due to the mandatory interaction between the teams. Especially due to the continually rising pace of software evolution and adaptation, maintaining consistency between software artefacts becomes increasingly more costly and complex. An example of a scenario that leads to inconsistencies between the architecture model and the software system is the replication of a database. Even if an accurate model was built at design-time, it is mandatory that it needs to be updated. In this case, the operator who performs the replication has to adjust the model manually.

Therefore it is desirable that the effort for consistency maintenance is minimized as much as possible. In this thesis we focus on the consistency between architecture models, source code and the executed application in the run-time environment. The architecture model that is used is the Palladio Component Model (PCM) [6], which emphasizes model-based performance predictions. Since the improved planning possibilities are an important advantage of the models, it is not sufficient to derive them from running software. Rather, it is desirable to build an approach that interweaves the evolution of software and models. Moreover, the adaptation of the software at run-time should also be considered and taken into account. The goal is a phase-spanning approach to maintain consistency throughout the evolution and adaptation of the software.

There are already a number of approaches whose goal is to maintain consistency between software artifacts with the highest possible degree of automation. There are static approaches such as ArchLint [53] and Just-In-Time Tool for Architecture Consistency (JITTAC) [11], which aim to detect inconsistencies between source code and architecture models during development. Langhammer et al. presented a conceptually similar approach [50]. It extracts architecture models and usage models from source code and test cases. However, no monitoring events are taken into account at run-time. For this reason, it is impossible to detect and eliminate any inconsistencies that may arise through software adaptations. In contrast, there are also some approaches that deal with the derivation, maintenance and updating of models at run-time. A good summary can be found in [72]. In addition to the approaches mentioned in the summary, we highlight iObserve here [29]. The idea of iObserve is to update PCM models at run-time using monitoring data.

All already mentioned approaches have one thing in common: all of them do not consider the system composition, i.e. the structure and the arrangement of the components in the system. But this is also an important part of the architecture models and can be derived from information about the software (source code and monitoring data). Another

limitation of most existing approaches is that they do not validate the resulting models. This leads to uncertainty about the actual accuracy of the models. The CIPM approach [55] takes these self-validations into account, but in the current implementation these are only integrated prototypically. Moreover, most approaches focus on subtopics and often only cover evolutionary *or* adaptation aspects.

In the following section, the most important goals of the thesis are listed and described.

1.1. Goal of the Thesis

The main goal of the thesis is to design an approach which supports the consistency maintenance between software artefacts and architectural models. It should span the design-time, but also the run-time of the software. In particular, common evolution and adaptation scenarios should be supported (e.g. replication of a component at run-time). As a basis for this, the conception and features of iObserve [29], Vitruvius [46], and CIPM [55] are combined and used. Vitruvius should be used as a central building block to eliminate inconsistencies through consistency rules and to establish a mapping between elements of different models. At run-time, a transformation pipeline (based on iObserve and CIPM) should be used to derive up-to-date architectural models. So far, iObserve used the run-time architecture correspondence model (RAC) [62] to provide a mapping between model elements of different metamodels. In the course of the thesis, the RAC should be replaced by the correspondence model of Vitruvius. Furthermore, the self-validations that are already integrated in the CIPM approach should be extended and the results should be entered into the transformation pipeline. The pipeline can then address and eliminate existing inaccuracies.

The main contributions of the thesis can be summarized as follows:

(i) Extraction of the system composition at run-time and design-time. At design-time the derivation is based on the analysis of the source code. Since the exact behavior of the code at run-time cannot be predicted [18], an overestimation must be made here. At run-time, the system composition is automatically extracted from the monitoring data. This is based on the analysis of service call relationships.

(ii) Extension of the self-validations, based on the CIPM approach. The scope of self-validations is increased, several validations are performed within the transformation pipeline to directly estimate the impact on the accuracy. In CIPM, up to now only one validation was performed after the calibration of the architecture models and the detected inaccuracies were not used directly as input for the calibration [55, 56].

(iii) Fusion of different approaches to cover a variety of aspects of consistency maintenance, including iObserve, CIPM, Vitruvius and Langhammer's co-evolutionary approach. For example, the features of the iObserve transformation pipeline and the model calibration of the CIPM approach should be merged.

(iiii) Complete automation and expansion of the instrumentation process. Based on preliminary work in the CIPM context, the instrumentation should be extended in

such a way that the extent of monitoring can be controlled at run-time. This allows to effectively manage the arising overhead.

1.2. Structure of the Thesis

Chapter 2 presents the basics for understanding the thesis. Next, Chapter 3 explains the objectives of the thesis in detail and describes assumptions as well as limitations of our work. Subsequently, Chapter 4 outlines the conception of the developed approach. Chapter 5 presents the structure and the results of the case study based evaluation. Afterwards, Chapter 6 discusses the related work in the area of consistency maintenance between software artifacts and view based consistency. In Chapter 7, areas for future work are pointed out and finally, Chapter 8 summarizes the complete thesis and describes the most important insights.

2. Foundations

The thesis relies on several concepts and approaches, which are introduced in this section. First, Section 2.1 introduces the basics of modeling and model-driven development. Then, Section 2.2 introduces the Palladio Component Model, which can be used to create models of component-based software systems. Section 2.3 describes Vitruvius, an approach to ensure consistency between models. The principle of application monitoring is introduced in Section 2.4. Subsequently, the last three Sections 2.5, 2.6, and 2.7 describe existing approaches which address the consistency preservation between software systems and architecture models.

2.1. Model-Driven Software Development

Models are used in software development for various purposes. According to Stachowiak's definition, a model has three properties: a model is an representation of an original (*mapping feature*), a model does not reflect all properties of the original (*reduction feature*), and a model can only replace the original under certain constraints and in certain contexts (*pragmatic feature*) [68]. In the context of software development, this is the most common and widely used definition.

Model-driven software development (MDSD), also known as Model-driven development (MDD), refers to procedures that generate executable code from formal models [69]. These formal models describe an aspect of the software system entirely, for example the structure or the behavior. The main goal is to generate the source code of a software from a model whose complexity is lower than the complexity of the resulting source code. This results in many advantages, such as a higher development speed, a higher degree of reusability and the separation of concerns [69]. An example for a modeling language is the Unified Modeling Language (UML), which can be used for specification, construction, documentation and visualization of software parts and other systems [63]. In the context of model-driven development, so-called domain specific languages (DSLs) are frequently used. In contrast to general-purpose languages such as Java and C, these are optimized for special tasks and domains. Consequently, DSLs are less expressive, but more precise and compact.

For MDSD it is common that several different models are used. These models can have different levels of abstraction and are adapted or optimized for a particular purpose. Typically, the models contain redundant or dependent information. During the evolution of the models this leads to the fact that changes affect several models. These adjustments must then be done either manually or automatically. Otherwise, inconsistencies between the models may occur. This process is very complex and requires a high degree of knowledge about the dependencies and redundancies between the models.

2.1.1. Metamodels

Metamodels are also models that describe the possible structure of models, in particular the constructs of the modeling language, the relationships between them, the constraints and modeling rules [69]. A well known metamodel is the UML which has already been mentioned earlier. In order to be able to describe these meta models it is necessary to establish a meta-meta model. The Object Management Group (OMG) has introduced the Meta-Object Facility (MOF) standard, which describes a special metadata architecture [60]. The core component is a self-describing meta-meta model, which provides a general basis for metamodels. The data is arranged in four meta levels [60]:

M0 Concrete data

M1 Models, for example a concrete instance of UML that represent the M0 level

M2 Metamodels, which describe the structure and composition of the models on M1 level (e.g. UML)

M3 Meta-meta models used to define the M2 plane. These must be self-describing, i.e. the M3 layer must be describable by means of the M3 layer.

A metamodel must cover four different aspects: abstract syntax, concrete syntax, static semantics and dynamic semantics [69].

The *abstract syntax* describes the entities that make up the models, their properties and relationships. However, it does not make any statements about the representation of these entities, properties and relationships. [69]

For each abstract syntax at least one *concrete syntax* must be defined. However, several concrete syntaxes can be defined for an abstract syntax, there is no upper limit. A concrete syntax describes the representation of the components, properties and relations, which are defined in the abstract syntax. Typical forms for a concrete syntax are the graphical or textual representation. [69]

Static semantics describes restrictions and rules that cannot be expressed using abstract syntax [69, 1]. It is important that these rules are verifiable. An example of a language for describing static semantics is the Object Constraint Language (OCL) [16].

The dynamic semantics specifies the meaning of the elements. Often this is expressed in natural language. However, it is also possible to map the meta model to a formal language (e.g. Java or Petri nets), which can be used to formally define the dynamic semantics. [69]

2.1.2. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is, as the name suggests, a Java framework intended for modeling purposes [12]. It is a project of the Eclipse open source community and supports the generation of Java code from models. The underlying meta-meta model is called Ecore and forms the basis for modeling [12]. It is based on a subset of the MOF standard for metamodeling, which has already been briefly described.

Using the EMF offers many advantages. The entire ecosystem covers many aspects of modeling. Java source code can be generated directly from the Ecore models. In addition,

there is the possibility to include an editor directly in the generation. With the help of this editor instances of the model can be created and edited. Furthermore, test cases are generated which check the generated Java source code. The models can be directly integrated as a plug-in into the Eclipse platform, but they can also be used as standalone Java code [22, 12]. The serialization and de-serialization of models is also covered by EMF. The models can be converted into the XML Metadata Interchange (XMI) format and stored in files [74].

The EMF also offers many features for the creation of Ecore models. There are two different editors, one diagramm-based and one tree-based. This allows the implementation of structures, relationships and rules. It is also possible to directly enter OCL expressions, which depict the static semantics of the Ecore models. The goal of the EMF is that high-quality Ecore models can be designed and deployed without much effort.

The EMF contains a number of other tools, including implementations of model transformation languages such as QVT [49] and ATL [36], the Java Virtual Machine (JVM) based programming language Xtend [7] and Xtext [23, 7], which is designed for DSL development.

In this thesis the EMF was used to establish new model types and to adapt existing models.

2.2. Palladio Component Model

The Palladio Component Model (PCM) is a model-based approach to the analysis of software architectures [6]. The PCM focuses on the evaluation of performance aspects, such as the detection of bottlenecks and scalability problems. However, it is also possible to analyze reliability and other quality characteristics, such as privacy issues [31, 59]. The advantage of the model-based approach is that the software does not have to be implemented and then tested, rather it is possible to make statements about the impact of design decisions at an early stage of development. As a result, high costs for wrong decisions in development can be avoided and scalable software can be designed from scratch.

The PCM divides the specification of the software architecture into five different model parts [5]:

1. Repository Model
2. System Model
3. Resource Environment Model
4. Allocation Model
5. Usage Model

These models are designed for different user groups according to the separation of concerns principle. An overview of the building blocks of a PCM instance is given in Figure 2.1. It also shows the roles and users that are responsible for each model. The Allocation Model

and the Resource Environment Model are combined in this figure and both are part of the system deployer's scope.

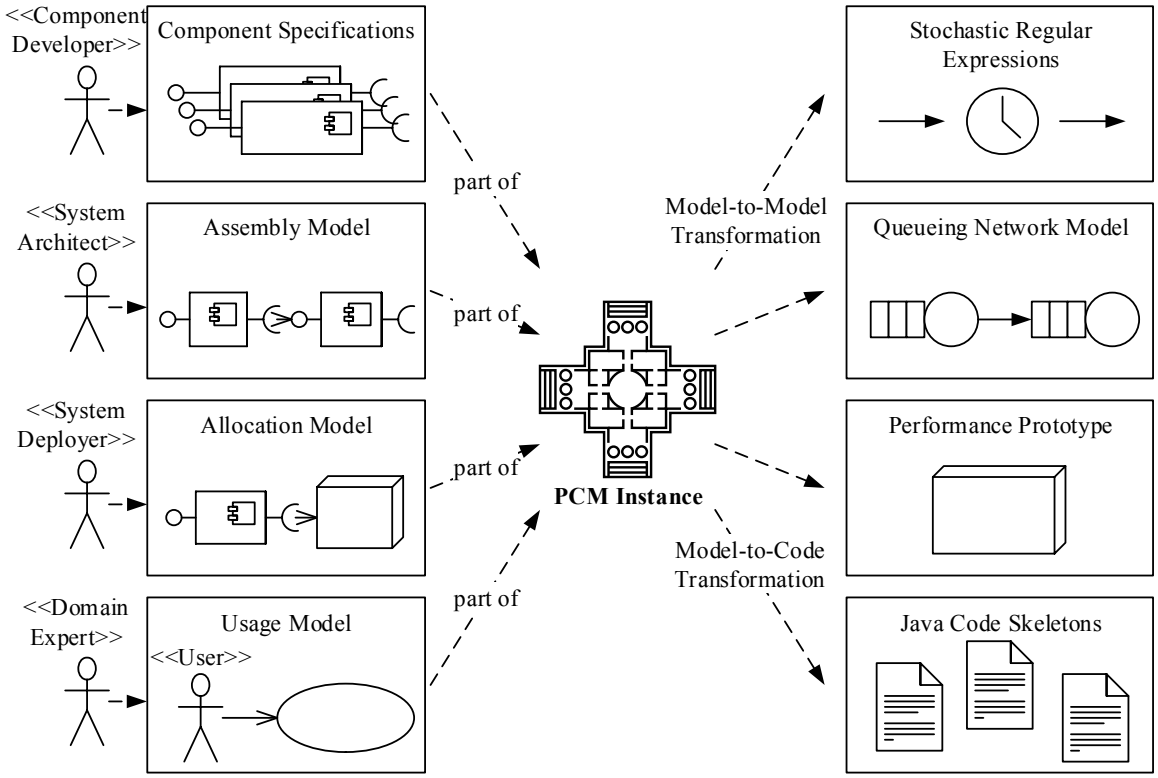


Figure 2.1.: Overview of the Palladio Component Model [27]

A fully specified PCM instance can be used to perform various analyses and to generate artifacts. For example, it is possible to transfer the PCM instance into a Queueing Network Model [19], which can then be used to make statements about the performance. Furthermore, it is possible to generate a basic skeleton of Java code that reflects the PCM. It should be noted that the PCM is on a higher level of abstraction than the Java source code. Therefore, the generated code should only be seen as a framework and not as a functionally complete implementation.

In the following paragraphs, the model parts of a PCM instance are described in more detail.

Repository Model Within the *Repository Model*, the components and the interfaces of the software architecture to be modeled are specified. The interfaces contain signatures with parameters and return types that characterize the services used or provided by the components. The required and provided interfaces for the components can be specified via so-called *Provided Roles* and *Required Roles*. Another important part of the repository model are the *Service Effect Specifications* (SEFFs) within the components. These describe the behavior of a service provided by the respective component. The structure is similar to UML activity diagrams [2], but the SEFFs are specifically designed so that *Resource Demands* can be defined for the actions. These *Resource Demands* define the hardware resources

required for an action. To ensure the accuracy of the simulation results, adequate values for the resource demands are essential. These values can be described using stochastic expressions [43], so that distributions can be represented as well. Furthermore, it is possible to relate the demands to input parameters, so that parametric dependencies can be modeled. Figure 2.2 shows the visual representation of an exemplary SEFF. The *Component Developer* is responsible for creating the components, interfaces and SEFFs in the Repository.

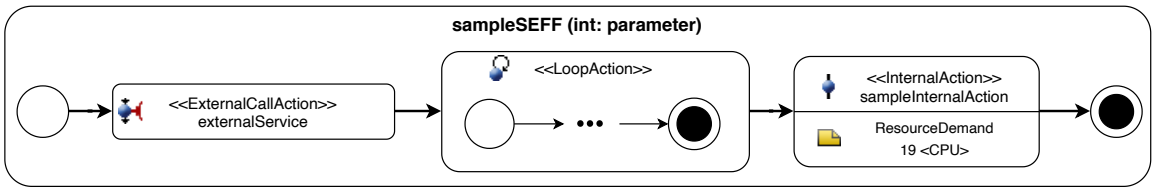


Figure 2.2.: Presentation of an exemplary Service Effect Specification (SEFF)

System Model The *System Model* describes the composition of the software architecture, based on the components and interfaces that have already been created in the Repository Model. The *System Architect* is responsible for performing the modeling tasks that concern the System Model. For the system as a whole, the provided and the required interfaces must be defined. These must then be connected via *Delegation Connectors* to instances of components within the system. These instances of components are called *Assembly Contexts*. Using *Assembly Connectors*, provided and required interfaces of Assembly Contexts can be connected to each other and thus be satisfied. The interplay of repository and system model makes it possible to create composite components. They consist of assembly contexts and their connection via assembly connectors and are ultimately a part of the repository model. These composite components can then be "instantiated" again via Assembly Contexts and therefore allow reusability on a higher level.

Resource Environment and Allocation Model The *Resource Environment Model* reflects the actual hardware environment of the software system. The main elements are the *Resource Containers*, which represent a computer with specified available resources. These resources embody performance-related hardware, such as the hard disk or the CPU. In other words, no exact clock speed or cache size is specified for a CPU, but only an expression that describes how many tasks can be executed per time unit. In the same abstract way, the resource demands are specified in the SEFFs, so that the actually required times for a specific action can be calculated during the simulation. Connections between the Resource Containers are realized by so-called *Linking Resources*. For these links properties like latency and throughput have to be configured.

The structure of the *Allocation Model* is very simple. It consists of Allocation Contexts, which realize the mapping from Assembly Contexts to Resource Containers. It is necessary that all Assembly Contexts within the System Model are mapped to Resource Containers.

Both the Resource Environment Model and the Allocation Model are created by the *System Deployer*.

Usage Model The *Usage Model* defines the behavior of users, i.e. the way in which they interact with the system. For this purpose, different *Usage Scenarios* can be created, which represent the behavior of individual user groups. For these scenarios, it is possible to specify the workload, i.e., how many users possess this behavior within a certain period of time. The structure of a scenario is very similar to the SEFFs within the Repository Model. Loops and branches can also be integrated. The creation of the Usage Model is the task of the *Domain Expert*.

2.3. Vitruvius

Vitruvius (**VI**ew-cen**TR**ic engineering **U**sing a **VI**rtual **U**nderlying **S**ingle model) is a view-based approach which is designed to keep instances of different metamodels consistent [14, 44]. It uses a Virtual Single Underlying Model (VSUM) [50] which contains the models. These models can only accessed via views. Consistency between the models is ensured by so-called consistency preservation rules [45]. They are specified on metamodel level and consist of triggers, retrievals and actions [45]. Therefore, it is possible to propagate a change in a certain model through all models. Furthermore, Vitruvius also provides a bidirectional language, which allows us to specify mappings between metaclasses of different metamodels [45].

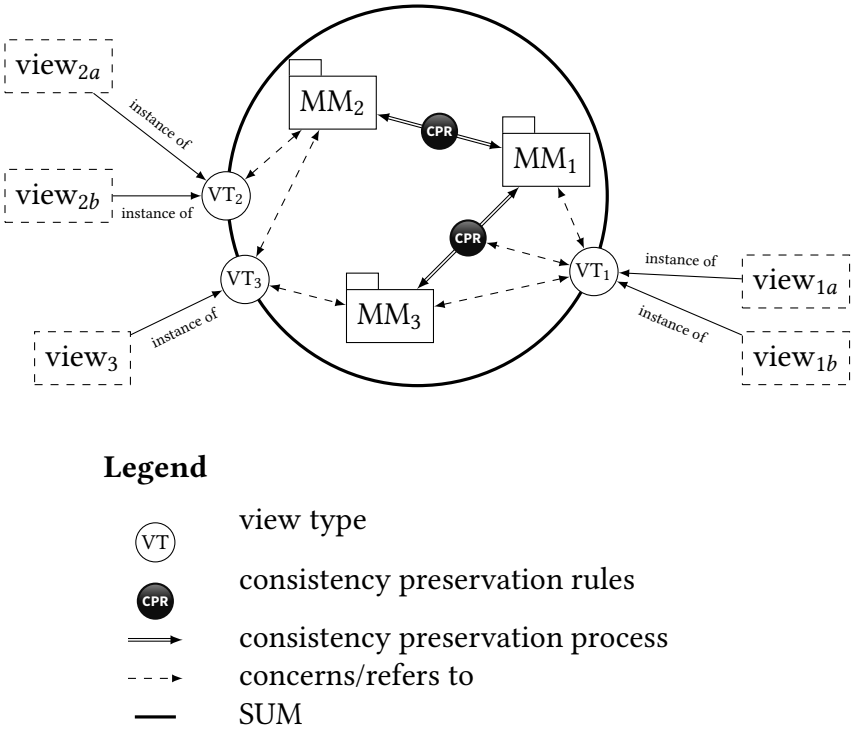


Figure 2.3.: Graphical overview of the most important building blocks and the functionality of Vitruvius [50]

The already introduced views are defined on view type level and the views are basically an instance of a view type. The views enable access to the model instances within the

VSUM. As a consequence, the only way to modify the model instances is provided via views [50]. *ModelJoin* can be used within a view type to bundle data which is distributed over different model types [14, 13, 50]. Figure 2.3 visualizes the conception of Vitruvius and shows the main building blocks.

Vitruvius follows a change-driven approach, i.e. the models are monitored and in case of changes the consistency maintenance process is triggered [40]. In this way, dependent and redundant information in other models is updated, mainly based on the consistency maintenance rules already mentioned. The consistency maintenance process can be roughly divided into four parts: first, the appropriate transformations are selected, then they are executed, afterwards the correspondences between the elements of the different models are updated and finally the adapted models are saved again [47].

The correspondence model is responsible for managing the relationships between the elements of different models within the VSUM. The structure is relatively simple, it consists of only two classes: the superordinate *Correspondences* class, which can contain an arbitrary number of correspondences. The correspondence is established either via Temporarily Unique Identifiers (TUIDs) or via Universally Unique Identifiers (UUIDs) [50]. In other words, a mapping between two elements is characterized by a pair of TUIDs or UUIDs. A TUID is basically a string consisting of the combination of the file and an identifier within the file for the respective element. Consequently, a TUID changes when the file path changes. In contrast to this stands the UUID, which remains unique even if the file path changes. However, this can only be realized if the elements in the respective model have an identifier [50]. An example of a model type where UUIDs can be used is the Palladio Component Model (PCM), see Section 2.2.

The roles for Vitruvius are split into two user groups: the methodologists and the developers [40]. The methodologists define the transformations and the conditions for correspondences between the elements, i.e. they design the basic framework for maintaining consistency. The developers interact with the models in the VSUM via the provided views and thus trigger the specified transformations which ensure that consistency constraints are fulfilled [40].

2.4. Application Monitoring

In the context of thesis, we refer to monitoring as the process of observing a running application and collecting data about run-time properties. A classic use case is the diagnosis of performance problems, such as bottlenecks and excessive loads. During the monitoring process, various metrics are collected and evaluated. These often include CPU usage and network load. An important goal of monitoring is to supervise service level agreements (SLAs) and to ensure their fulfillment [38].

A well known method for establishing a monitoring is the manual insertion of code, which collects metrics at run-time. Another widely used method is the automated instrumentation of the application at startup. Instrumentation is the process of adjusting the code of the application to be monitored in order to collect the required metrics. An example is the insertion of time measurements at the beginning and end of a method to record its execution time. The second method is often used with Java, because it is possible

to change the bytecode of the application during run-time. This means that the application to be monitored does not even need to know about the monitoring process, instead it can be used without additional effort. The monitoring of the application is then realized via the so-called "Bytecode Instrumentation" [8].

In this thesis we focus on Kieker [32] and use it for monitoring purposes. Kieker is a widely used monitoring framework for "monitoring and analyzing the run-time behavior of concurrent or distributed software systems" [33]. In addition to monitoring features, analysis possibilities are also supported, e.g. it is possible "to extract and visualize architectural models" [32]. Special attention is thereby paid to the expandability, so that monitoring and analysis can be extended without much effort by plugins. Kieker supports several programming languages and methods for instrumentation of the program code [34]. As a result, the monitoring can be used across borders and on a large-scale. Another important advantage of Kieker is the low overhead caused by collecting the metrics at run-time [32]. This ensures that the monitoring does not significantly affect the performance characteristics of the application under observation. The underlying data structures for the monitoring data are specified using the Instrumentation Record Language (IRL) [37]. It enables monitoring record types to be defined in a style similar to that of classes. The Instrumentation Aspect Language (IAL) is responsible for determining where measurements are performed within the application [37]. It also determines which attributes should be recorded.

2.5. Coevolution of Source Code Behaviour and Architectural Elements

Langhammer proposed an approach that facilitates the co-evolution of source code and architectural models [50]. The main goal is to keep architecture models consistent with the source code of the implementation. Therefore, the approach of Langhammer uses Vitruvius and defined consistency preservation rules. These rules provide automated consistency maintenance between the repository model and the corresponding source code of the component-based architecture. Besides that, it extends SoMoX to derive Service-Effect Specifications (SEFFs) from source code [50, 48]. EJBMox is a further development of SoMoX which eliminates several limitations [51]. Since the Service Effect Specifications (SEFFs) are an abstraction of the source code, there is a limitation that only the SEFF can be built from the source code and not the other way around [50]. The approach also supports the derivation of a usage model. This procedure is based on analyzing the test cases and then extracting user behavior [50]. The approach also implicitly establishes a mapping of elements in the architectural model to the corresponding parts in the source code. This is realized via the already mentioned Correspondence Model of Vitruvius.

The first step in the process of extracting a repository model from the source code is parsing the source code. This requires a metamodel for Java, in this case JaMoPP [28] was used. The major advantage of JaMoPP is that it is based on the Eclipse Modeling Framework (EMF) and can therefore be seamlessly integrated into the VSUM of Vitruvius.

An overview of the models and the consistency preservation rules in the resulting VSUM is shown in Figure 2.4.

There are two different ways to maintaining consistency between the repository model and the source code. One of them is change-driven and derives changes in the repository directly from changes in the source code. The deduction of the changes is based on Vitruvius and reaction rules for preserving the consistency [50]. The other approach extracts a repository model from existing source code. This is based on a "Linking Integration Strategy" [50].

Langhammer's approach also integrates the possibility of using performance measurements to generate resource demands for the actions within the services. This was realized by means of the monitoring tool inspectIT [1]. However, a crucial drawback is that the application has to be set up and deployed to be able to perform this strategy.

[1] https://inspectit.rocks/

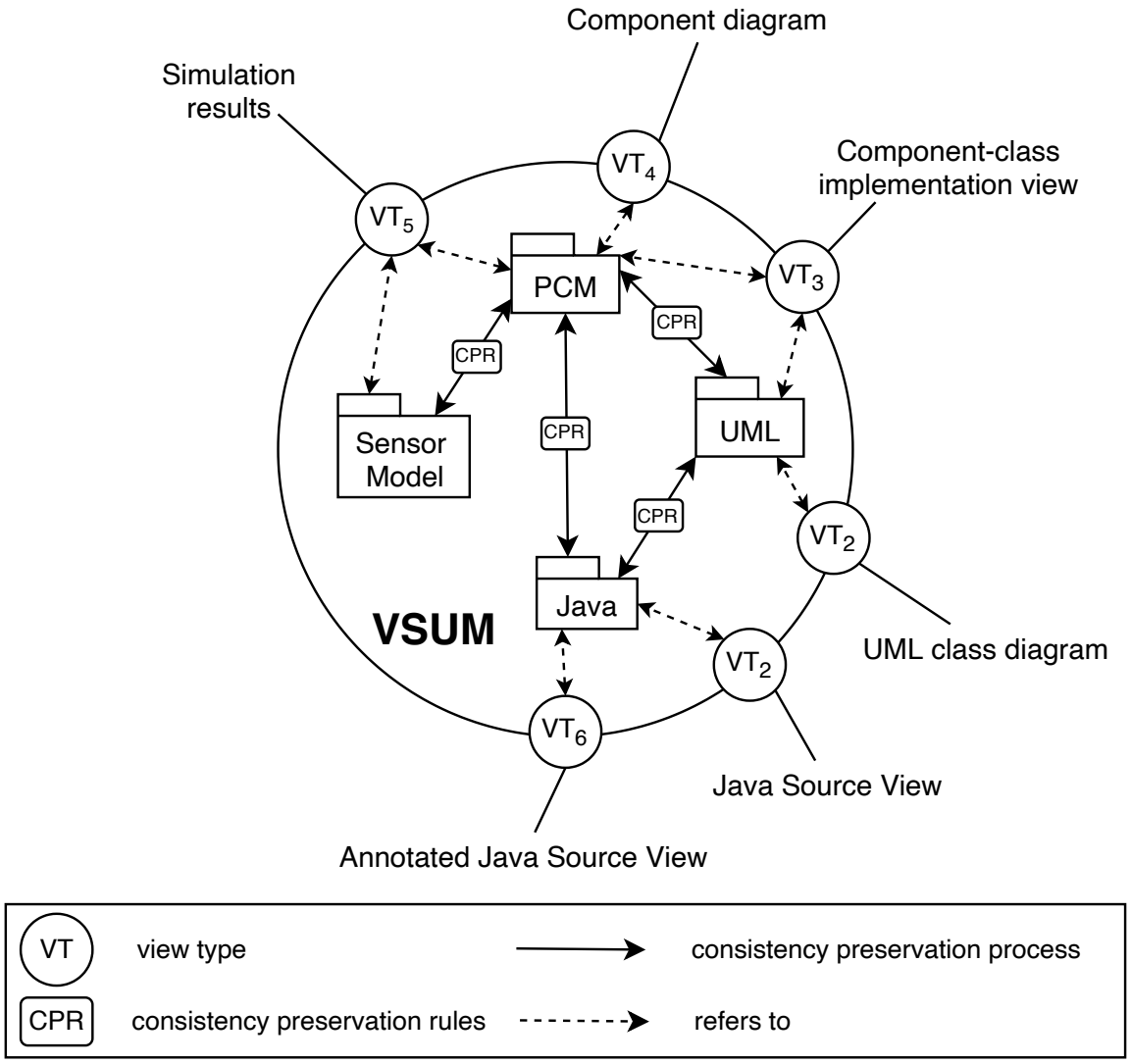


Figure 2.4.: Overview of models, consistency rules and views in Vitruvius' VSUM in the context of Langhammer's co-evolution approach [50]

2.6. iObserve

iObserve addresses the system adaptation and evolution of cloud-based systems to enable continuous quality analysis [29]. The main idea is to use run-time observations to detect changes during the operation phase and derive necessary adjustments to the quality predictions. The Palladio Component Model (PCM) is used as the basis for the quality predictions and Kieker is used for monitoring purposes. Briefly summarized, iObserve collects monitoring data at run-time with the help of Kieker and applies necessary changes to the architecture model (PCM instance). The conceptual implementation of iObserve adopts and extends the MAPE (Monitor, Analyze, Plan, Execute) control loop model [29, 10]. Adaptation and evolution are interwoven and shared models are used throughout the application life-cycle to close the gap between run-time and design-time. Figure 2.5

visualizes the interweaving of adaptation and evolution and also shows the application life-cycle in the context of iObserve.

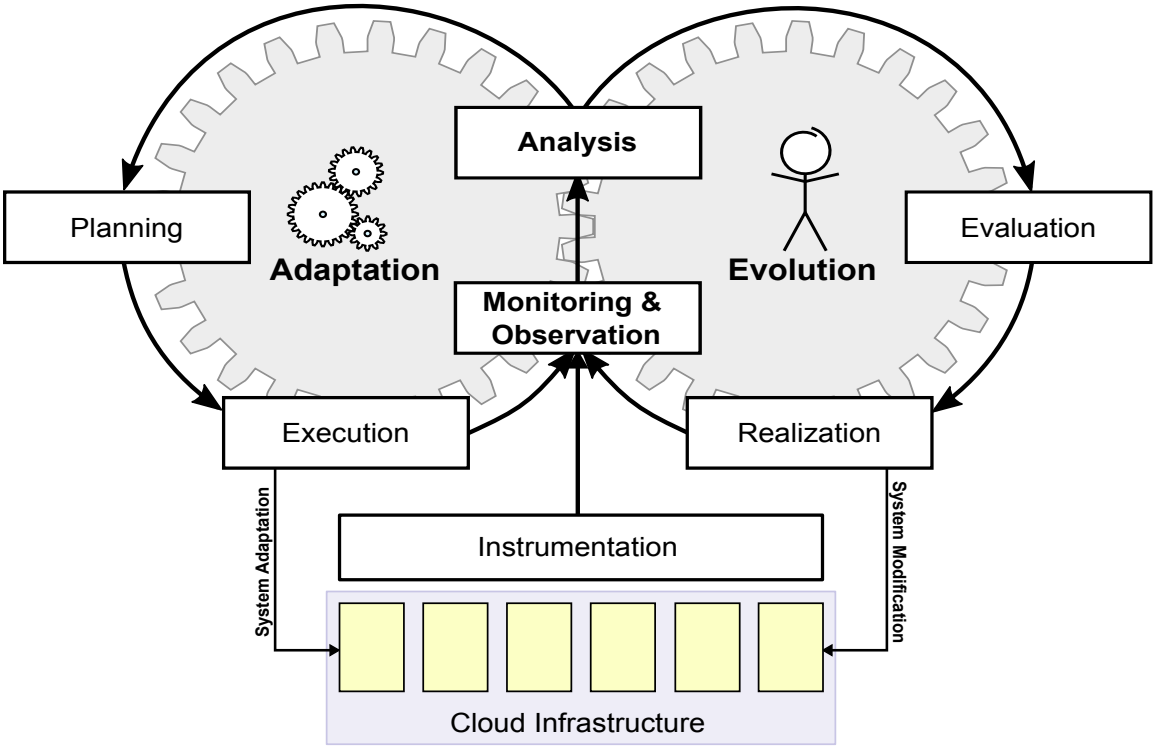


Figure 2.5.: Overview of the application life-cycle in the context of iObserve and illustration of the interweaving of adaptation and evolution [30]

The following six change scenarios concerning the deployment and use of the application are supported by iObserve [30]:

1. Workload characterization changes: Changes in user behavior and/or intensity of use
2. Migration: Transfer of a component from one machine to another
3. Replication: Duplication of a component to achieve load balancing across the different component instances
4. Dereplication: Opposite operation to replication, undeployment of a component that was previously replicated
5. Allocation: Start-up of a new machine
6. Deallocation: Shutdown of an execution container

The mapping between elements in the architecture model and corresponding elements in the source code is implemented in iObserve via the Runtime Architecture Correspondence Model (RAC) [30, 62]. It was developed specifically for this purpose in the context

of iObserve. The mapping is a central part of the concept, otherwise it is not possible to relate the monitoring data to elements of an existing architecture model. In the iObserve megamodel, the RAC forms the transition between the development and operation within the application lifecycle. Figure 2.7 shows the entire megamodel graphically. Four dimensions are distinguished, on the one hand, implementation level and model level, on the other hand, development and operation [30].

The main component of the architecture of the iObserve implementation is a transformation pipeline based on the tea-and-join principle [15]. As input it uses the monitoring data collected by the Kieker monitoring framework. As can be seen in the megamodel of iObserve, the structure and amount of monitoring data is mainly influenced by the Record Type Model and the Instrumentation Model. The Instrumentation Model defines at which points within the implementation measurements should be taken. The monitoring data is processed in such a way that ultimately necessary changes to the architecture model (PCM) can be carried out. The required changes result from the change scenarios mentioned above. In this context, iObserve focuses on deployment and workload changes.

The first step in the pipeline is to pre-process the monitoring data, i.e. to aggregate, group it and discard unnecessary data. This high-level transformation is called $T_{Preprocess}$ and consists of several sub-transformations. The result of this transformation are the "Aggregated & Refined Events", consisting of Deployment Events, Undeployment Events and the Entry Call Sequence Model, which contains the invocations of the externally provided system interfaces. The most important and extensive transformation within the pipeline is $T_{RuntimeUpdate}$. It uses the data generated in the pre-processing steps and derives the necessary changes to the architecture model. Just like the $T_{Preprocess}$ transformation, it also consists of various sub-transformations that break the functionality down to smaller parts. Figure 2.6 visualizes the inner structure of the transformation.

The Entry Call Sequence Model is used to build an adequate usage model by means of the $T_{Workload}$ transformation. The calls of system interfaces by users are first subdivided

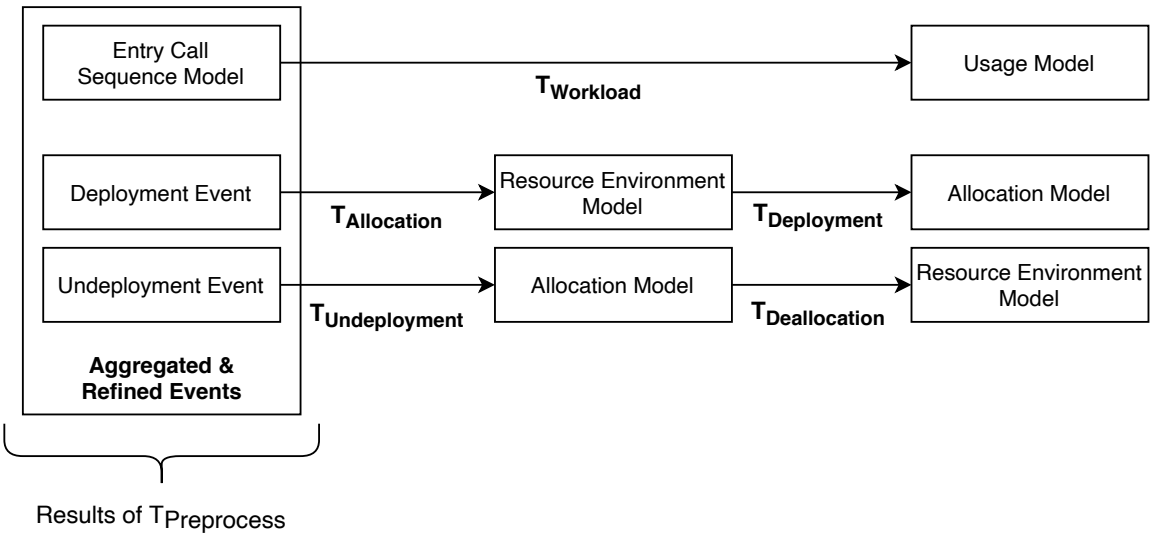


Figure 2.6.: Summary of the structure of the $T_{RuntimeUpdate}$ transformation [30]

into groups, then branches are detected and finally loops are identified. The results are then used to construct a usage model with usage scenarios. A usage scenario represents a group of users that exhibit similar behavior.

The processing of deployment and undeployment events is relatively simple. For each deployment event the $T_{Allocation}$ transformation first checks if the corresponding resource container exists in the resource environment model and if not it is created. Afterwards the allocation model is adjusted and the new allocation context is created. This is done by the $T_{Deployment}$ transformation. First the corresponding allocation context is deleted, then the container in the Resource Environment Model is deleted if it is no longer present in the monitoring data.

In the current version, iObserve does not support the calibration of the resource demands within the repository model, whereas this is supported by the CIPM approach (see following section).

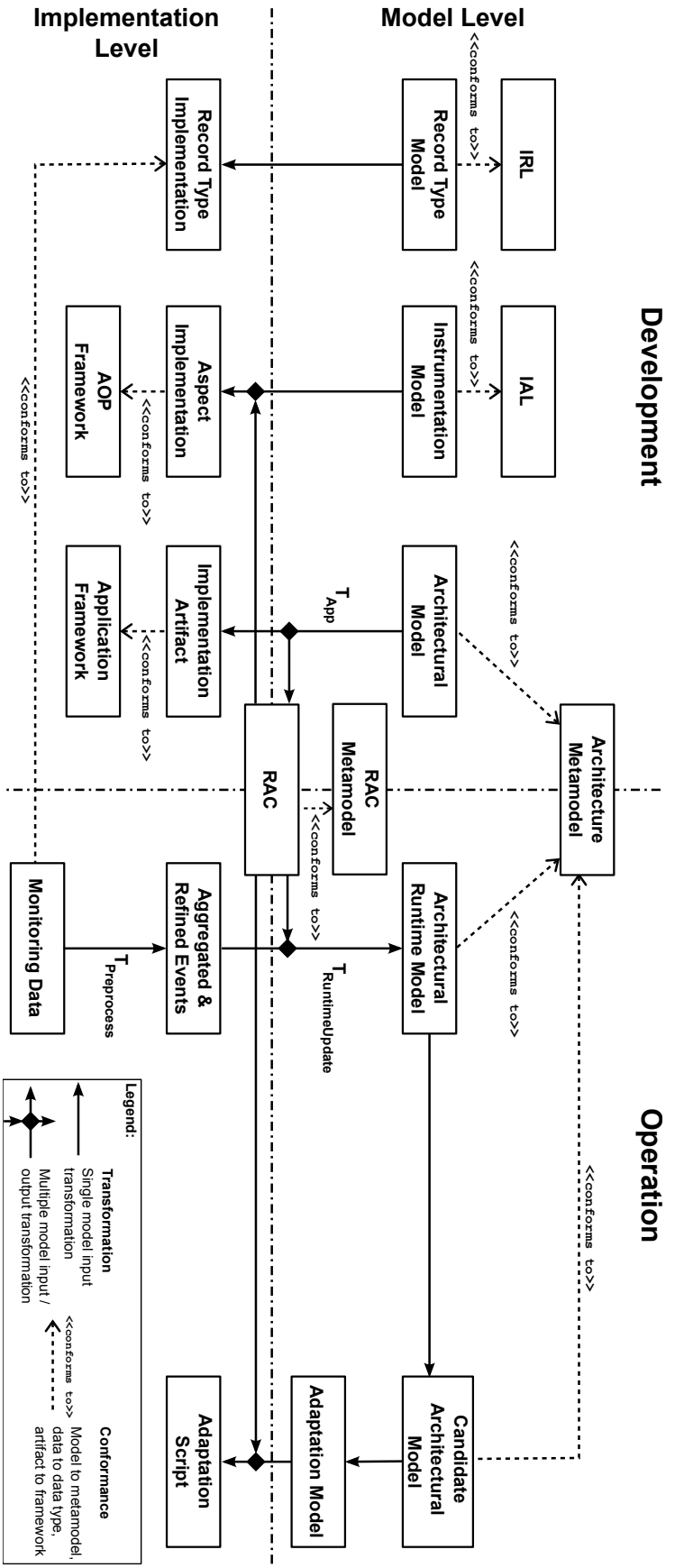


Figure 2.7.: Comprehensive summary of the iObserve megamodel [30]

2.7. CIPM Approach

CIPM (Continous Integration of Performance Models) [55] is an approach that aims to adapt the architectural model after each source code commit. CIPM is based on the incremental extraction of software architecture models from Langhammer [50]. However, further functionalities have been added, for example the parameters in the model are updated with the help of run-time information. It uses an adaptive monitoring to collect data while the application is running. Thereby not only information about service calls is collected (so called *coarse-grained monitoring*), but also about the behavior within the service calls (so called *fine-grained monitoring*) is tracked [55]. It is called adaptive monitoring, because we do not instrument the whole code finely granular, just the parts that have changed. This can significantly reduce the overhead caused by the monitoring. The architecture model is recalibrated using the data collected by monitoring. The calibration also consideres parametric dependencies, which improves the generalization of the calibrated model. In this context, the generalization describes how accurate the simulation results are for an usage scenario that has not yet been monitored. Figure 2.8 summarizes the most important activities and metamodels related to CIPM.

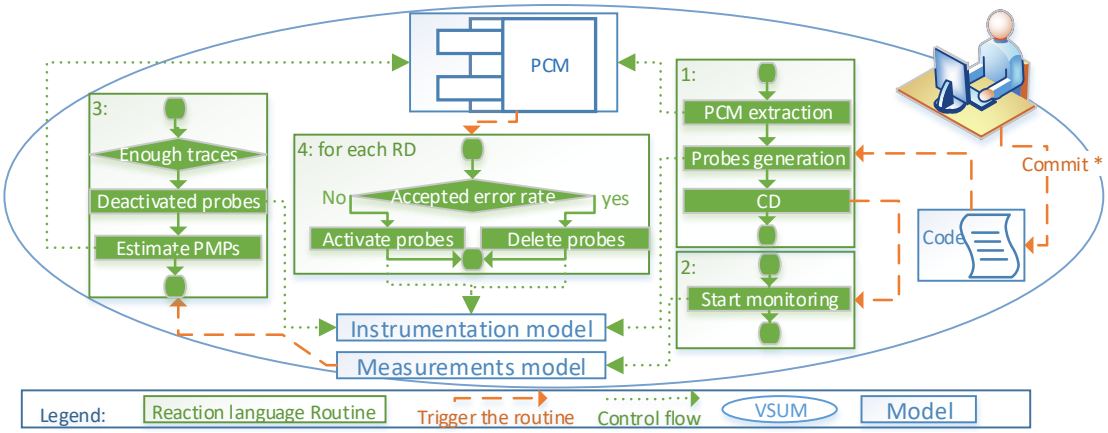


Figure 2.8.: Activities and Metamodels involed in CIPM [55]

In the following sections, the most important parts of the CIPM approach are presented and explained in more detail.

2.7.1. Monitoring

By means of monitoring, measurements are carried out during the run-time of the application, which should examine important properties. In particular, this includes measurements that cover the performance related characteristics. The current implementation of CIPM uses Kieker as monitoring framework. The so-called *Instrumentation Model* controls the extent of the monitoring and decides which services are observed fine-grained and which coarse-grained. In the following paragraphs the Instrumentation Model is described, fine granular monitoring is compared to coarse granular monitoring and the data structures on which the monitoring is based are introduced.

Fine-Grained vs Coarse-Grained Monitoring In general, CIPM distinguishes between two different types of monitoring: fine-grained and coarse-grained.

With coarse granular monitoring, only the service calls are observed at run-time. This means that no information about the internal behavior within a service is collected. This type of monitoring produces a significantly reduced overhead compared to fine-granular monitoring, but the data is less meaningful. A suitable use case for coarse granular monitoring is the validation. If the goal is to detect whether a service is well represented in the architectural model, the simulation results can be compared with the results of the coarse-grained monitoring.

With fine-grained monitoring, on the other hand, all service calls are tracked along with the corresponding internal actions, loop actions, and branch actions. Logically, this results in an increased performance overhead at run-time. But this makes it possible to perform precise adjustments in the model to align it with the run-time behavior (so called "Calibration" process).

It does not make sense to monitor all services permanently fine-grained or coarse-grained. Rather, it is desirable to trigger a fine-granular or a coarse-granular monitoring of certain services, depending on the accuracy of the model. This is enabled by means of the so-called Instrumentation Model. It determines individually for each service which parts should be monitored. Each time after the execution of the "Ops-time calibration" the Instrumentation Model gets updated. Using the monitoring data as reference, the accuracy of the modeled services is estimated. In case of high inaccuracies, the Instrumentation Model is modified in such a way that the corresponding services are monitored fine-grained in the next iteration. The structure of the Instrumentation Model in the context of the CIPM approach is outlined in the next paragraph.

Instrumentation Model The structure of the Instrumentation Model is very simple and consists of only two classes. Figure 2.9 illustrates the structure of the underlying metamodel. The *AppProbes* class serves as container for an arbitrary number of *Probe* instances, which

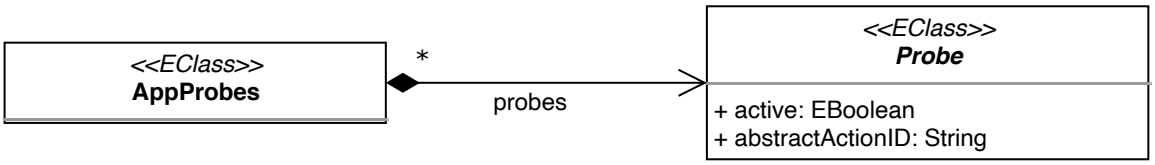


Figure 2.9.: Structure of the Instrumentation Model which is embedded in the CIPM approach [17]

define for the elements of the services whether they are monitored or not. The activation and deactivation is enabled via the "active" attribute. The elements of the services are addressed via their ID as defined in the Palladio Component Model (PCM). The mapping of the service elements to the implementation (code) is realized via Langhammer's co-evolution approach [50].

Monitoring Record Types The record types are the centerpiece of the monitoring and describe in which format the observation data is collected and stored. In order to define

the record types, the Instrumentation Record Language (IRL) of Kieker has been used [33]. Figure 2.10 shows the data structures on which the monitoring is based. In general,

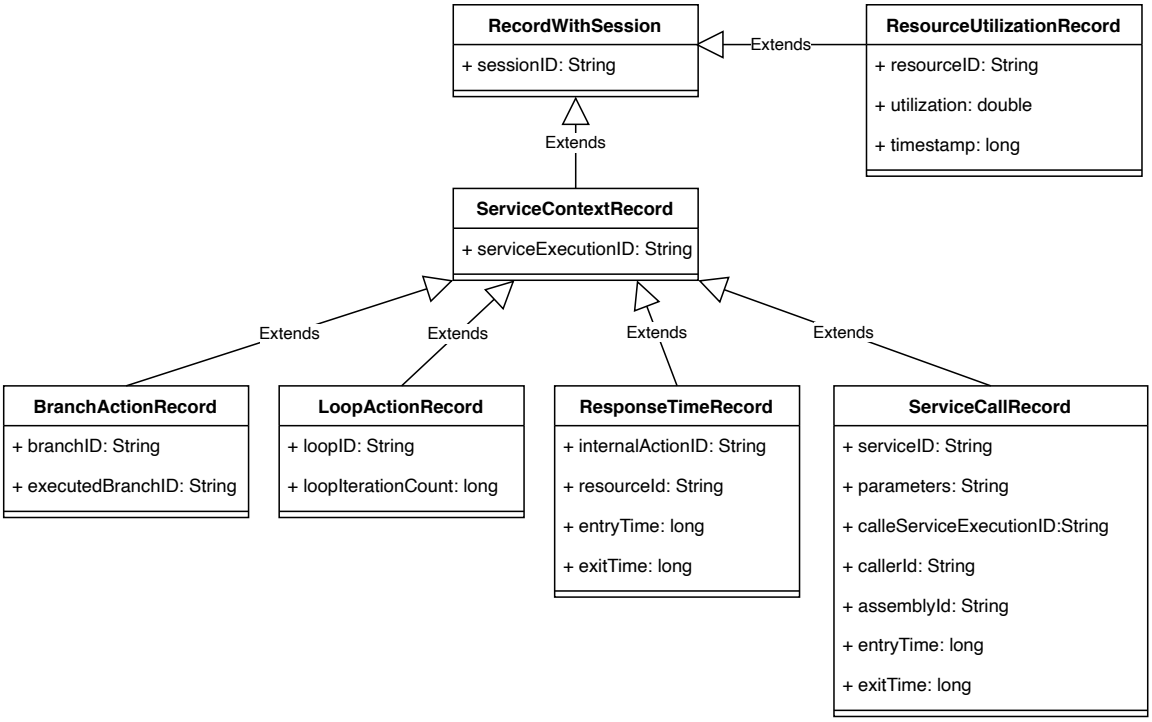


Figure 2.10.: Class diagram visualizing the monitoring record types of the CIPM approach [35, 56]

there are two superordinate classes of record types: the records that can be assigned to a user session (*RecordWithSession*) and the records that are part of a service execution (*ServiceContextRecord*). The utilization of hardware resources is tracked with the help of the *ResourceUtilizationRecord* record type. The remaining record types are responsible for tracking behavior in the context of a service call. *BranchRecords* are used to monitor branch executions, *LoopRecords* to log loop iterations, *ResponseTimeRecords* to capture internal hardware usage and *ServiceCallRecords* to trace the call of a service itself.

2.7.2. Incremental Calibration

CIPM distinguishes two types of incremental calibration of architecture models: *dev-time calibration* and *ops-time calibration*. The dev-time calibration includes the analysis of source code changes and the derivation of resulting changes to the architecture model. The ops-time calibration, on the other hand, uses monitoring data to adjust the deployment and the usage model.

If new parts are added to the source code, the dev-time calibration ensures that they are appropriately instrumented. The goal is to estimate the resource demands using the monitoring data. In this context a special focus is put on the detection of parametric dependencies. In other words, the resource demands are calibrated in such a way that they are expressed in dependence on certain parameters. This increases the prediction

power of the models significantly, since it also allows accurate statements to be made about scenarios that have not been observed so far.

The ops-time calibration is largely based on iObserve (see Section 2.6). For this purpose, the used parts of iObserve were adapted to operate on the monitoring data structures of the CIPM approach. The run-time Architecture Correspondence Model (RAC) was not used because the mapping is implicitly embedded in the monitoring records of the CIPM approach. The mapping is established by the co-evolution approach and subsequently woven into the source code within the instrumentation process [56].

2.7.3. Self-Validation

During self validation, the current architectural model is simulated and the results are compared to reference data. Resulting deviations can then be used to adjust the calibration process in order to eliminate them. The self-validation process is always performed after a calibration. The dev-time calibration uses test data as reference, whereas the ops-time calibration uses monitoring data. Based on the results of the self-validation process, the Instrumentation Model is adjusted as well. For services with a large discrepancy between the simulation data and the reference data, fine-grained monitoring is activated.

3. Objectives and Boundaries

This chapter clarifies the scope of the thesis and points out the main goals. First, the goals are introduced in Section 3.1. Next, in Section 3.2 the basic terms that are necessary for understanding the approach are explained. Section 3.3 outlines the scope of the approach and for which kind of usage scenarios it is suitable. Finally, Section 3.4 presents the limitations of the approach and the assumptions that were made.

3.1. Objectives

The goal is to design and develop a platform that simplifies keeping source code and software architecture models consistent throughout the lifecycle of a software. This is supposed to be realized by automated procedures as well as approaches which support the developers in keeping the source code consistent with the architectural models. The necessity of already existing architecture models should not be required, which is especially interesting for software without existing initial architecture models. Furthermore, special attention is paid to ensure that the evolution and adaptation of software is supported.

Since the architectural models are located at a higher level of abstraction some information is inevitably lost. Therefore, the consistency preservation is often not bidirectional. Due to this fact, in most parts it is only possible to propagate changes of the source code to the architecture model and not vice versa.

In the following section, we introduce the core features and use-cases that are covered by the approach. Thereby we go into more detail about the scope of the consistency maintenance and about the change types that are supported. These pre-defined requirements then determine the scope and certain characteristics of the approach (see Chapter 4 below).

3.2. Terminology

In the following, we will often distinguish between design-time and run-time. Design-time represents the time frame until the deployment of the application under consideration. Run-time, on the other hand, represents the period after the deployment of the application. In this context, deployment refers to the process during which the application is put into a production or test environment. This differentiation is crucial with regard to the information which is available about the composition and the behaviour of the application. In general, there is much more information about the application at run-time; for example, the user behavior is usually only identifiable when the application is executed in a production environment. In addition, re-configurations and other events that occur after the deployment are unknown at design-time. With the help of application monitoring, we are able to keep track of these events. Figure 3.1 illustrates this phenomenon.

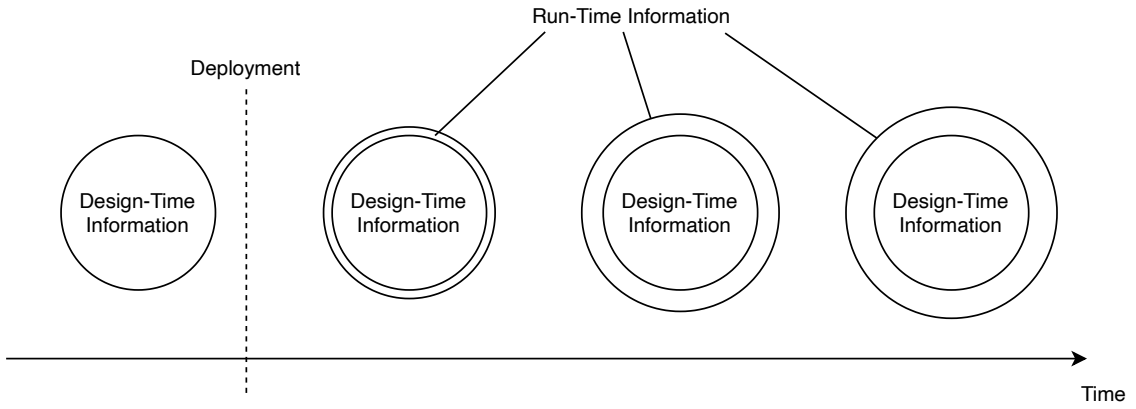


Figure 3.1.: Comparison between the information available at run-time and at design-time as time progresses

The distinction plays an important role in the procedures for the automatic extraction of architectural models. We have to take into account what information and data is already present, depending on the point in time.

This results in techniques that have the same goal, but can be completely different depending on whether they are executed at run-time or at design-time. Therefore, we will make a fundamental differentiation between design-time and run-time.

3.3. Scope and Usage Scenarios

The developed approach supports the consistency preservation of all parts of the Palladio Component Model (PCM). However, depending on the model type, not all element types and properties are kept consistent, i.e. there are some limitations. These restrictions are listed and explained in section 3.4.1.

The following table shows for each model part of the PCM whether consistency maintenance is supported with the corresponding application/source code. We distinguish between run-time and design-time, and also indicate on which preliminary works the consistency preservation is based. Furthermore, it is mentioned whether the consistency preservation of the respective parts has been conceptually extended in our approach.

The generation of usage models at design-time has not been considered in our approach, however there are existing techniques which use test cases to generate them [50]. Furthermore, Resource Environment Models and Allocation Models are not considered at design-time. In many cases it is not possible to derive these models automatically from the data that is available. The information is simply missing, since it cannot be found in the source code. Rather, the deployer of the system determines the resource environment and allocations individually. Nevertheless, there is also an ongoing project in this area. It attempts to derive related models for the resource environment and allocations from documentation and additional information (such as Docker files) [67]. The derivation of the resource environment, allocation and usage at run-time are conceptually identical to the iObserve approach, although they were migrated to our monitoring data structure.

Design-time	Supported	Foundations	Extended
Repository	✓	Coevolution approach[50]	✗
System	✓	✗	✓
Resource Environment	✗	✗	✗
Allocation	✗	✗	✗
Usage	✗	Coevolution approach[50]	✗
Run-time	Supported	Foundations	Extended
Repository	✓	CIPM[56, 35]	✓
System	✓	✗	✓
Resource Environment	✓	iObserve[29]	✗
Allocation	✓	iObserve[29]	✗
Usage	✓	iObserve[29]	✗

Table 3.1.: Summary on the considered Palladio Component Model parts for consistency preservation - distinguished between design-time and run-time

Besides the relatively rough information about the scope in Table 3.1, there are some specific use cases that should be supported by our approach. These are particularly important with regard to the evaluation, which will then measure and ensure the quality of our approach. The most important use cases are:

[U1] *Allocation/ De-Allocation*: Appearance/disappearance of resource containers on which parts of the software system can be executed.

[U2] *Migration*: Moving a component that is allocated on a specific resource container to another resource container. This procedure consists of two parts, first the component is removed on the original container and then deployed on the new container.

[U3] *Replication/De-Replication*: Replication is related to migration, the difference between the two is that replication does not remove the component from the original container. In other words, after the replication process we now have two components of the same type, but on different containers. De-replication is simply the rollback of a replication.

[U4] *Changes of the system composition*: Changes to the system composition at run-time/design-time, including, for example, the change of a used component. This point will be addressed in more detail in the remainder of this thesis when describing the approach to ensure the consistency of the system model.

[U5] *Usage behavior*: Users change their behaviour and thus influence the system. For example, users buy more products on average.

[U6] *Workload*: If the system is used by more or less users than before, the workload changes. This can have significant impact on the system and its performance characteristics.

In the remainder of the thesis, these use cases will be referenced frequently, as they are the foundation for many requirements of the approach. Furthermore, the procedure and structure of the evaluation is designed according to these scenarios.

3.4. Assumptions and Limitations

In this section we go into more detail about limitations and introduce assumptions that have been made. The difference here is that the limitations mainly result from the fact that some parts were not considered due to complexity reasons. These could be eliminated with future research. The assumptions, however, were made for reasons of abstraction and are unlikely to be eliminated by future work.

3.4.1. Limitations

The approach has the following limitations:

[L1] To be able to use the consistency maintenance at run-time, it is necessary that the application can be monitored. If there is no source code or no access to the application, some parts of the consistency maintenance cannot be used.

[L2] Performance attributes of links (*Linking Resources*) within the Resource Environment Model are not calibrated. In this context, calibration describes the alignment of the stochastic expressions within the model elements based on the monitoring data. These include, for example, the latency and throughput of a connection, which are not automatically adjusted using the monitoring data. The same is partly true for the characteristics of the resource containers, here only certain properties are calibrated, such as the number of CPU cores.

[L3] In the automated derivation of the system model at run-time and in the derivation process of the system model at design-time, composite components are not considered. These are a buliding block of the PCM which enable the reuse of certain compositions of components. This limitation can be eliminated, but this requires a very high implementation effort. Furthermore, the same systems can be modelled with and without composite components.

3.4.2. Assumptions

The following assumptions were made in the context of the thesis:

[A1] No new components are added at run-time, instead they are always developed at design-time and are not introduced into the production environment until they are deployed.

[A2] The interfaces provided by the system as a whole do not change at run-time.

[A3] A specific component type can only be deployed once on a particular container. This means, for example, that a component of type "database" can only occur once on a host. However, it is allowed when there are two different component types of databases (e.g. MySQL and SQLite) deployed on one host. The problem that arises when we do not make this assumption is that we cannot reliably distinguish between the component instances. This problem occurs because source code and architectural model are located at different levels of abstraction. Therefore, we usually cannot create one-to-one mappings between elements. For the automated derivation of a System Model we need exactly this one-to-one mapping between monitoring data and the component instances. A pair of component type and host must therefore be unique, otherwise it is not possible to automatically derive the System Model unambiguously. Theoretically, it would be possible to weave information about the component instance into the monitoring, but this requires domain knowledge and differs from usage scenario to usage scenario. This assumption sounds very restrictive at first, but the excluded cases rarely occur in practice. There were also no conflicts in the two case studies which we used for evaluation.

4. Approach

First, an overview of the approach and the most important contributions is given in Section 4.1, then a running example is introduced in Section 4.2. It is used to illustrate parts of the approach with a realistic and clear example. Section 4.3 shows the changes to the Virtual Single Underlying Model (VSUM) of Vitruvius in the context of the thesis. Subsequently, Section 4.4 presents the extensions regarding the monitoring. These include changes to the Instrumentation Metamodel, the extension of the monitoring records and the conception of the automated instrumentation process. Afterwards, Section 4.5 presents the structure and purpose of the so-called Service Call Graphs (SCGs). These are used at different places in our approach to represent call relationships between services. Section 4.6 and Section 4.7 introduce the concepts for maintaining consistency at design and run-time. Finally, Section 4.8 presents the transformation pipeline, which updates the architecture models at run-time using the monitoring data.

4.1. Overview

In this section, we provide a rough overview of the conceptual design of the approach and explain how it can be used to enable consistency between source code and architectural models.

Figure 4.1 shows a very simplified view of the approach as a whole. In order to derive the initial parts of the architectural model, it is only necessary that source code already exists. The source code can be used to extract a repository model. This process was inherited from previous work in this context [50]. The output of the derivation process is a repository model and a mapping between entities in the source code and corresponding elements in the architecture model. This mapping is essential for further processes and especially for the instrumentation (2) of the application. In order to obtain information about the running application, it is necessary to monitor it. To achieve this we use Kieker [33], which is well established and often used for this purpose. Conceptually, the instrumentation process is based on the preliminary work of Dahmane [17], but has been extended within this thesis. The monitoring is based on the idea that we weave information from the mapping into the source code during the instrumentation process. This allows us to easily link the monitoring data to the elements in the architecture model. The methodology of the monitoring and the instrumentation process are explained in more detail in Section 4.4.

Furthermore, the approach supports the derivation of a system model using the mapping together with the source code as input (3). This process does not work fully automated, it is necessary that a developer (e.g. the System Architect; see Section 2.2) manages conflicts that cannot be resolved automatically at design-time. As a core building block we use a

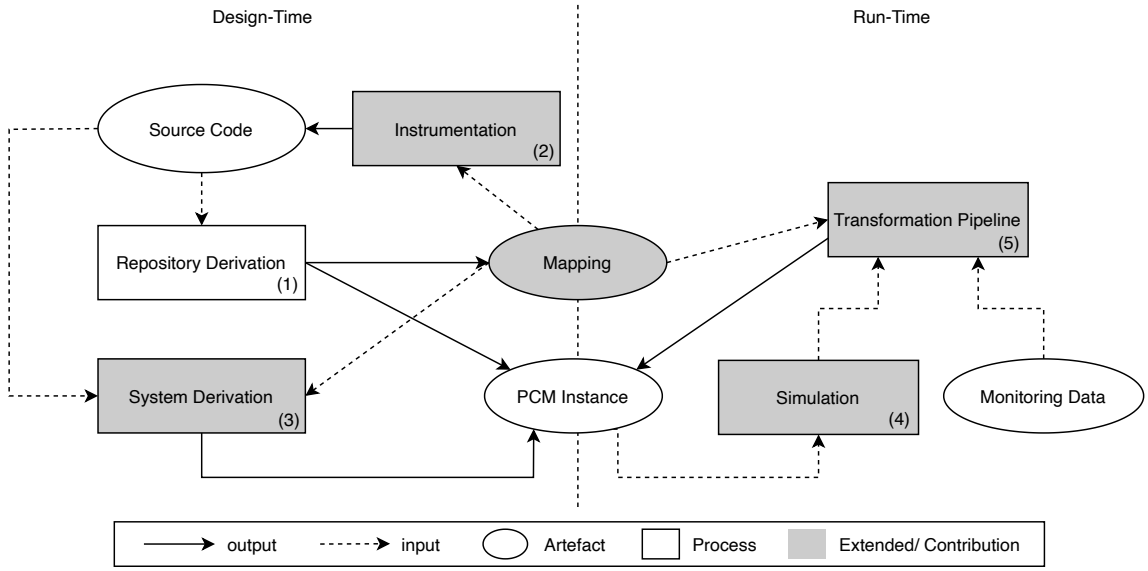


Figure 4.1.: Summary of the most important artifacts and processes at design-time and run-time on a high level of abstraction

static pointer analysis, which determines the composition of the system with the help of the mapping. In existing approaches for consistency preservation, the System Model was not taken into account, which is why this process had to be designed and implemented from scratch. In Section 4.6.2 the approach is explained in more detail.

At run-time we use the monitoring data as input for a transformation pipeline which updates all parts of the architecture model accordingly. The special characteristic here is that the current architectural model is simulated in advance and the results of the simulation are fed into the transformation pipeline as input. Within the pipeline, the simulation results are compared with the monitoring data to estimate the accuracy of the current model. This concept is based on the CIPM approach, which calls this step "self validation" [55]. Consequently, it possible for the pipeline to adjust its behavior according to the accuracy or inaccuracy of the current architectural model. The transformation pipeline and the simulations of the models are discussed in more depth in Section 4.7.

4.2. Running Example

We have implemented and modelled a clearly arranged running example to illustrate parts of our approach and to be able to give informative examples. Our running example is an application that can be used to generate and retrieve prime numbers. Structurally it is designed rather simple, there is a server (*PrimeServer*), that generates or retrieves the prime numbers. The server uses a service (*PrimeGenerator*) to generate prime numbers within a certain interval. There are two different implementations for the server, one of them uses a database as cache and the other one generates the prime numbers anew with every call. Furthermore, there are different implementations (strategies) for the database, as well as for the generators of the prime numbers which can be used. Figure 4.2 shows

the Repository Model of the corresponding PCM instance. We also created models for all other model types in the PCM, but did not include them here to keep this section clearly arranged. Later, when describing some parts of our approach, we will refer to this Repository Model to improve the comprehensibility.

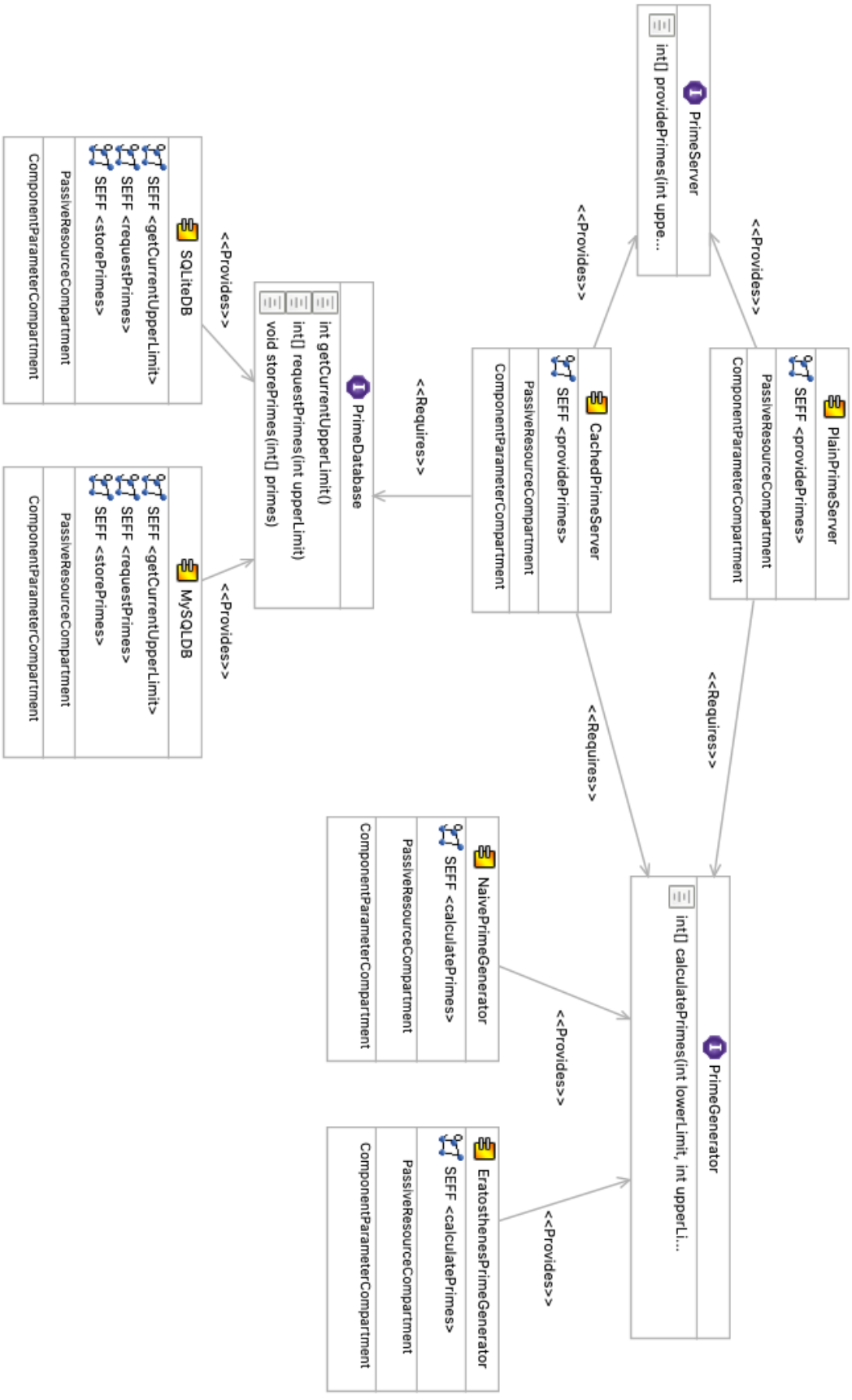


Figure 4.2.: PCM Repository diagram of the prime generator running example

4.3. VSUM Extensions

The correspondence metamodel of Vitruvius which was already mentioned in the foundations (cf. Chapter 2) is used at certain points in our approach to associate elements of different metamodels. Wherever we needed a mapping between elements of different metamodels, we used this correspondence mechanism. Within the transformation pipeline we intentionally opted against the mapping of iObserve. As mentioned in the Foundations, iObserve uses the Architecture Correspondence Meta-Model (RAC) [29]. To keep the mapping uniform, we replaced the RAC and used Vitruvius.

The starting point for the structure of the VSUM was Langhammer's approach [50]. The VSUM has been extended in two aspects for our approach:

1. The VSUM was extended to include the Instrumentation Model (IM). Some consistency rules have been extended in order to ensure that when services are added/removed in the PCM, associated elements are created/removed in the IM. The IM is based on the work of Dahmane [17], but has been completely redesigned for our approach. In previous projects, the IM has not yet been integrated into the VSUM, this has now been realized by our approach.

2. Extension of the VSUM to include the Runtime Environment Model (REM). The REM represents the actual runtime environment. It is mainly used to create a mapping between the resource containers in the PCM resource environment model and computers in the physical runtime environment. The computers within the environment are characterized by host names and other hardware properties. From these properties, we calculate an ID which is set in relationship to a container in the Resource Environment Model. The REM is introduced in Section 4.7. Furthermore, consistency rules were defined with the help of Vitruvius, which ensure that changes in the runtime environment are propagated to the PCM instance.

The final structure of the VSUM as it is used in our approach is visualized in Figure 4.3. We have omitted metamodels that are not relevant for our approach to increase simplicity (e.g. UML models). The consistency rules and the structure of the REM are explained later in Section 4.7, when it comes to consistency preservation at runtime.

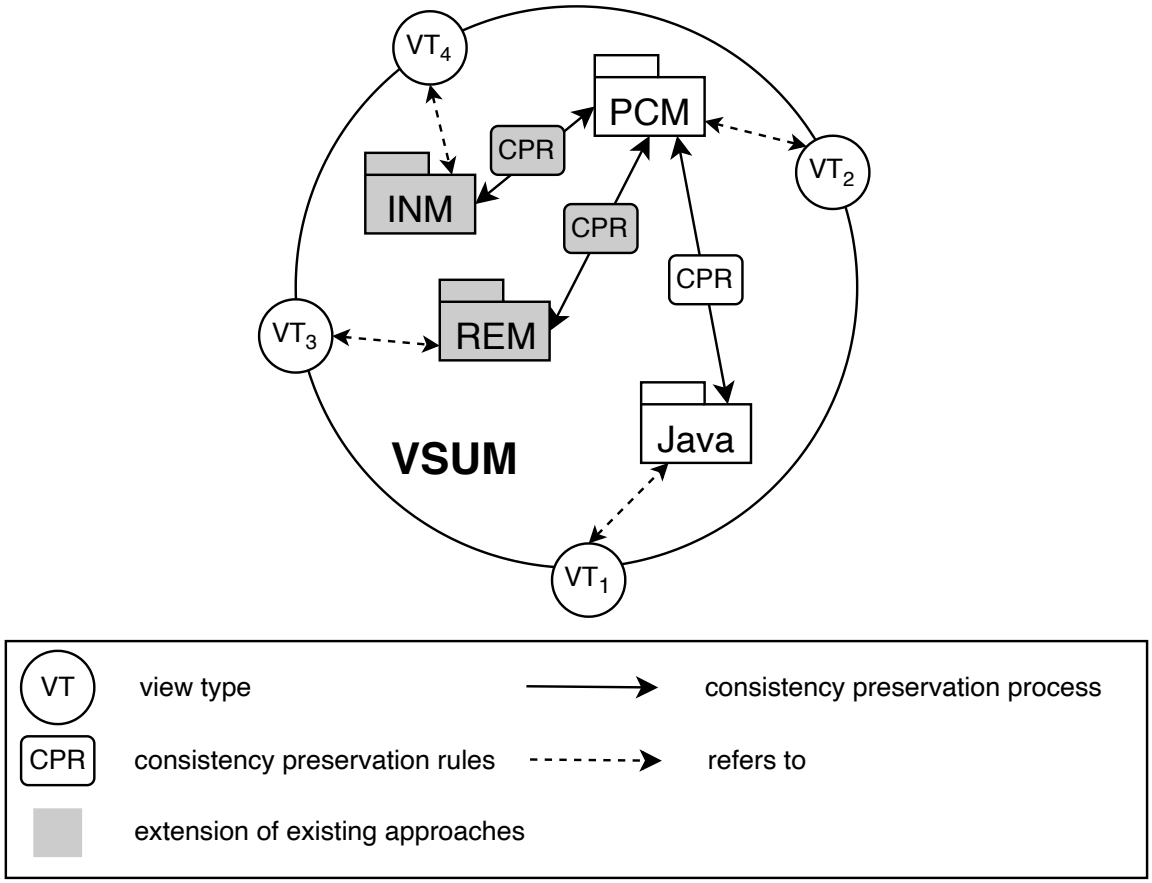


Figure 4.3.: Presentation of the resulting VSUM for our approach based on the structure of the co-evolution approach from Langhammer [50]

4.4. Monitoring

Section 4.4.1 explains the structure of the redesigned and extended Instrumentation Metamodel, which plays an important role in the monitoring. Afterwards, Section 4.4.2 then shows the modified monitoring record types, which are the underlying data structures for the monitoring data. Subsequently, Section 4.4.3 outlines the instrumentation process which is used to modify the source code and to establish the monitoring. Finally, Section 4.4.4 explains how we enable monitoring across the boundaries of different computers.

4.4.1. Instrumentation Metamodel Extension

The concept of the Instrumentation Model (IM) is based on CIPM and has been implemented in a previous work (see Foundations) [17]. The structure of the metamodel was redesigned and extended for our approach. In addition, our enhancements enable the IM to be regularly updated through self-validation, thus affecting monitoring at run-time (see Section 4.7.4). The resulting metamodel is very simple, consisting of only a few classes and one enumeration. It is based on the Eclipse Modeling Framework (EMF) [70] and

contains references to the core of the Palladio Component Model (PCM). The structure of the Instrumentation Metamodel is visualized in Figure 4.4 using a class diagram.

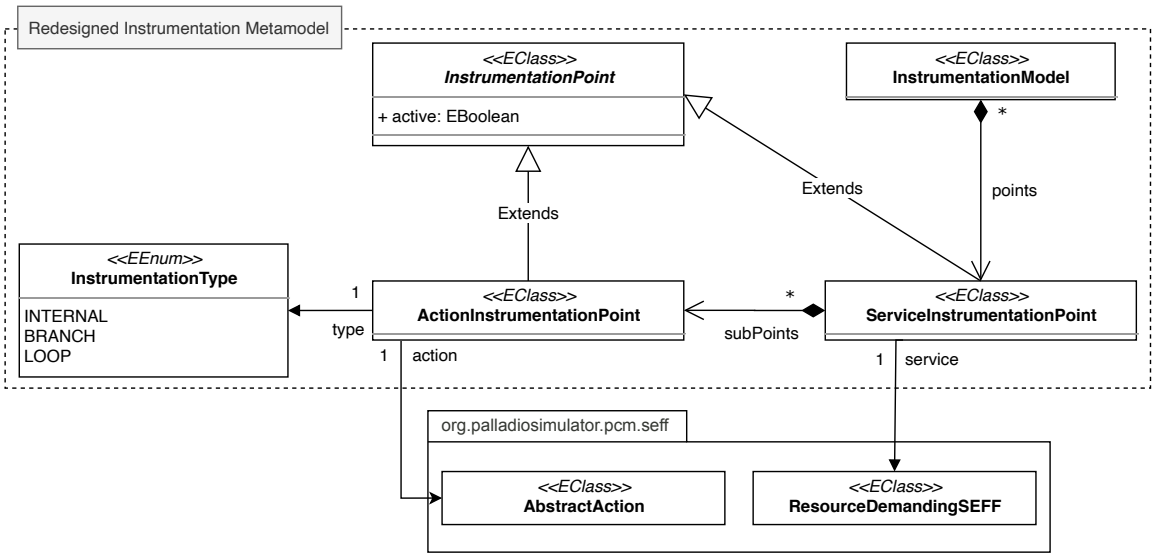


Figure 4.4.: UML class diagram for the Instrumentation Metamodel which has been built using the Eclipse Modeling Framework (EMF)

The *Instrumentation Model* class contains an arbitrary number of *Service Instrumentation Points* and is the root container of all elements. These *Service Instrumentation Points* control the monitoring for a complete service. This allows, for example, that the monitoring of a specific service can be completely deactivated or activated. The *Service Instrumentation Points* additionally contain a number of so-called *Action Instrumentation Points*. These control the monitoring at a finer level and make it possible to monitor branches, loops and internal actions within a service. The type of this instrumentation point is explicitly expressed through a reference to the *InstrumentationType* enumeration. With the help of the Object Constraint Language (OCL) it is ensured that there are no two different instrumentation points for the same element. This means that there can be a maximum of just one *ServiceInstrumentationPoint* per service and only one *ActionInstrumentationPoint* for each sub-action of a service.

During the initial generation of the instrumentation model for a corresponding Palladio Component Model instance, we create instrumentation points for all services and their underlying actions. However, only the points that realize the monitoring on service level are activated (coarse-grained monitoring, see CIPM [55]).

4.4.2. Monitoring Record Types Extension

The extension and modifications of the monitoring record types have been determined on the basis of the requirements for the monitoring data. A special characteristic of our monitoring is that we weave the information about corresponding model elements of the PCM directly into the records (see Section 2.7.1 about the CIPM monitoring). The final monitoring record types are shown in Figure 4.5 as UML class-diagram.

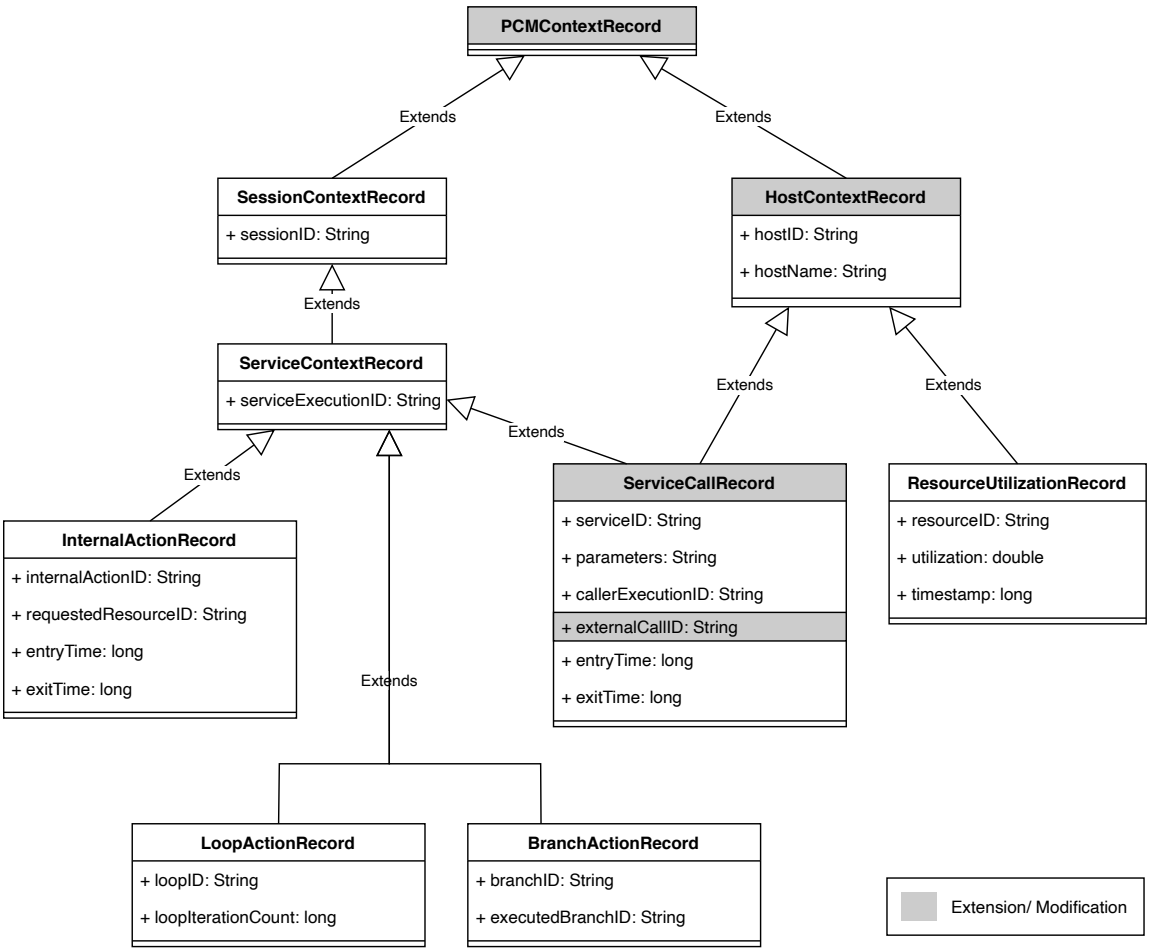


Figure 4.5.: UML class diagram which shows all monitoring record types including their inheritance characteristics and attributes based on the monitoring of the CIPM approach [35]

The newly introduced class *PCMContextRecord* serves as superclass for all record types. This allows us to easily filter out the records that are not relevant for our approach. The concrete types can be classified in two groups: the *SessionContextRecords* and the *HostContextRecords*. However, a record type does not necessarily has to be assigned to only one group, it can also belong to both. The monitoring record type *HostContextRecord* has been added and is used to associate the monitoring data with a host. It is designed analogously to the *ServiceContextRecord*. It indicates that there is an associated computer to which the record belongs. We not only store the hostname of the computer, we also save an ID of the host which is constructed based on the hardware and should be unique within the observed environment over a long-term period. The exact strategy for generating the host ID is not fixed, in the implementation we used a hash value based on the media access control address (MAC address) of the computer. A *ServiceContextRecord* indicates that a record is produced during the execution of a service. We generate a new ID for each execution of a service (*serviceExecutionID*) which must be unique within a certain time window (approx. 1-5 hours). The ID is required to extract traces from the monitoring data.

More details are provided in Section 4.7. Furthermore, we can determine the corresponding host for all subtypes of the "ServiceContextRecord" by using the mapping to service calls.

The *ServiceCallRecord* is used to monitor the execution of a specific service on a particular host. We save the parameters of the service execution JSON [61] encoded into the *parameters* attribute and if the service was triggered by another service, this is reflected by the *callerExecutionID*. This ID allows us to build traces of service calls. Furthermore, the attribute "externalCallID" was added, which indicates whether a service call was triggered by an external call or not. This is required for the automatic derivation of the system model at runtime (see Section 4.8). Multiple inheritance is not a concern here, as Kieker's IRL supports it and uses interface to build the appropriate Java classes. The *LoopActionRecord* and the *BranchActionRecord* have been adopted one-to-one from the preliminary work. The last concrete record type is the *InternalActionRecord* which measures the execution time for a specific Internal Action in the PCM model. In comparison to the former monitoring, it was renamed in order to keep the naming uniform.

4.4.3. Instrumentation Process

The goal of the instrumentation process is to ensure that the monitoring records are automatically recorded at runtime. The generated records must also be sent to a backend where they are processed. This is accomplished by the instrumentation process. In contrast to many other approaches used for instrumentation, our approach directly aligns the source code and not the bytecode of the application (see CIPM [56]). The disadvantage of this method is that we have to perform the instrumentation before building the application, which makes it less flexible. However, for two reasons it is mandatory that we directly operate on the source code and not on the bytecode. The first reason is that loops are irreversibly removed during the generation of Java bytecode and can therefore no longer be detected without further ado. The second problem arises from the mapping of elements in the PCM instance to elements of the source code. The mapping, which is explained in more detail in Section 4.3, cannot be adapted to the bytecode without considerable effort.

The existing instrumentation process uses Vitruvius to adjust the source code via consistency rules [17]. The problem is that the JaMoPP metamodel [28], which was used to represent the source code, only supports Java version 7 and lower. To eliminate this restriction, we use the JavaParser library[1] in our approach to modify the source code. In addition, the instrumentation process was adapted to support the structure of the redesigned Instrumentation Model (IM).

As input, the instrumentation process requires an IM, the Correspondence Model and of course the source code. We also assume that the associated repository model (of the PCM) is consistent with the current source code. This can be realized using Langhammer's approach [50]. The instrumentation process depends on this consistency, otherwise either problems occur directly or the generated monitoring data is not correct in the end. The instrumentation procedure is very straight forward, first we iterate over all services which are present in our IM. Next, we use the correspondence model to find the corresponding parts in the source code and instrument them. Then we check whether subordinate actions

[1] https://javaparser.org/

should also be instrumented for the particular service, and if so, these are instrumented in the same manner. Depending on the type of the action (loop, branch or internal action), the instrumentation procedure is different. It does not matter whether the instrumentation points in the IM are marked as active or not, the instrumentation will be performed in any case. This allows us to easily activate or deactivate monitoring points at run-time. However, it should be noted that the IM determines which parts should be instrumented at all. In other words, services that are not included in the IM will not be instrumented. The modification of the source code is quite simple, for each action to be monitored we add statements in the code before and / or after the action to track its execution. Listing 4.1 illustrates how an instrumented service call looks in Java source code.

```
1  @Override
2  public List<Integer> generatePrimes(int amount) {
3      ThreadMonitoringController threadMonitoringController =
           ThreadMonitoringController.getInstance();
4      try  {
5          ServiceParameters serviceParametersMonitoring = new ServiceParameters();
6          serviceParametersMonitoring.addValue("amount.VALUE", amount);
7          threadMonitoringController.enterService("_2RDcwKMhEemdKJpkeqfUZw", this,
               serviceParametersMonitoring);
8
9          // ORIGINAL SOURCE CODE OF THE METHOD
10     } finally {
11         threadMonitoringController.exitService();
12     }
13 }
```

Listing 4.1: Example of an instrumented service call in the Java source code

In order to clarify the instrumentation process it is described in Algorithm 1 using pseudocode.

Algorithm 1 Source Code Instrumentation

Input: Instrumentation Model (IM), Correspondence Model (CM), Source Code (SC)
Output: Instrumented Source Code

1: **for all** $SIM \in IM.serviceInstrumentationPoints$ **do**
2: $sourceCodeElements \leftarrow getSourceCodeElements(SIM.service, CM)$
3: $instrumentServiceCall(sourceCodeElements)$
4: **for all** $AIP \in SIM.actionInstrumentationPoints$ **do**
5: $actionSourceCodeElements \leftarrow getSourceCodeElements(AIP.action, CM)$
6: $instrumentAbstractAction(sourceCodeElements, AIP.type)$
7: **end for**
8: **end for**

4.4.4. Request Tracing

An important feature of our approach is to support distributed systems which run on different computers and communicate over the network. Since the monitoring on one

computer actually knows nothing about the monitoring on other computers in the network, the data is spread out. This leads to the problem that the monitoring data can no longer be correlated with each other. Especially for service calls that trigger a service call on another computer, this link between the two service calls would be irrevocably lost.

The answer to this problem is the so-called "request tracing". It is based on the idea that information about the current monitoring status is transmitted when communicating via the network. For example, the ID of the currently called service is appended to a network request. Because the monitoring data is processed at a single end point, it is possible to connect related data there. In the implementation, we only supported HTTP requests as a working prototype. However, this could easily be extended to other communication methods. We used a Java agent that modifies all HTTP requests so that the current *serviceExecutionID* and the *externalCallID* (see Section 4.4.2) are appended using HTTP header attributes. These IDs are used at the receiver to determine and recover the current state of the monitoring. This allows us to build service calls traces across the boundaries of different computers in the network. The request tracing is a prerequisite for some transformations within the transformation pipeline, which is explained in more detail in Section 4.7. In order to make request tracing more understandable, it is visualized in Figure 4.6.

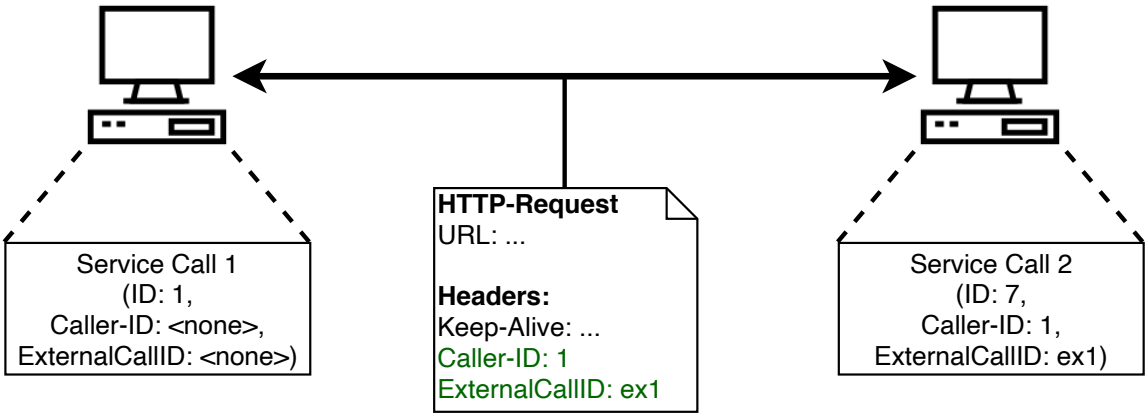


Figure 4.6.: Basic demonstration of request tracing by means of an HTTP request

4.5. Service-Call-Graph (SCG)

An important structure which is used at design-time but also at run-time is the Service-Call-Graph (SCG). Conceptually, we want to describe "calls-to" relationships between services. We also consider the resource container on which the respective services are executed. Graphically, this can be displayed as a directed graph where a service/resource container pair is a node and an edge indicates that a service on a particular container calls a service on a certain container. Figure 4.7 shows an exemplary SCG for the Running Example introduced in Section 4.2.

With the help of this structure, the analysis of the system composition can be significantly simplified, which is why we have implemented it as a metamodel using the

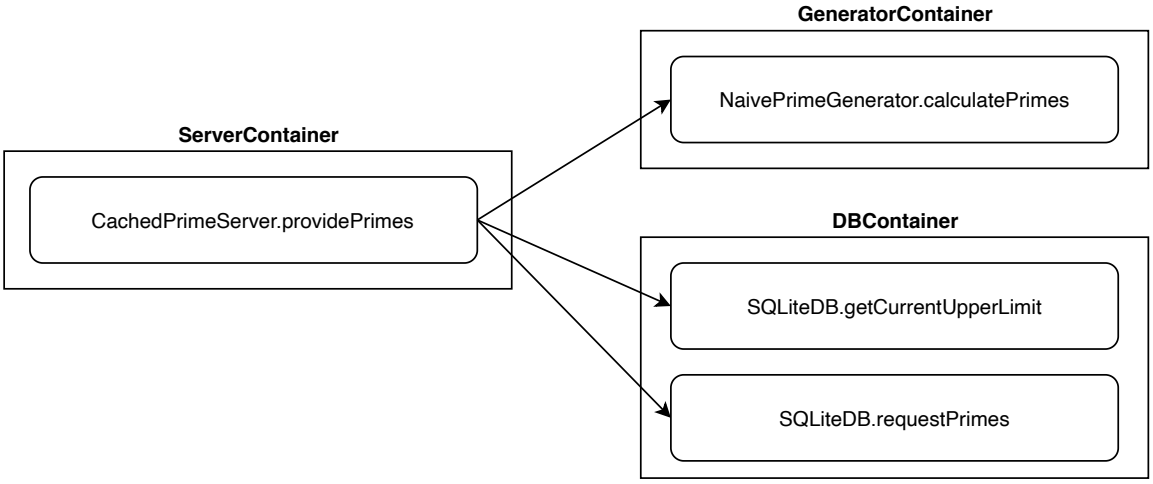


Figure 4.7.: Sample representation of a basic Service-Call-Graph for the Running Example

Eclipse Modeling Framework (EMF). Figure 4.8 shows the design of the metamodel. Due to technical reasons, the implementation became a bit more complicated in order to provide an efficient access to the edges of a certain node. But we hide these technical details here to keep the concept understandable.

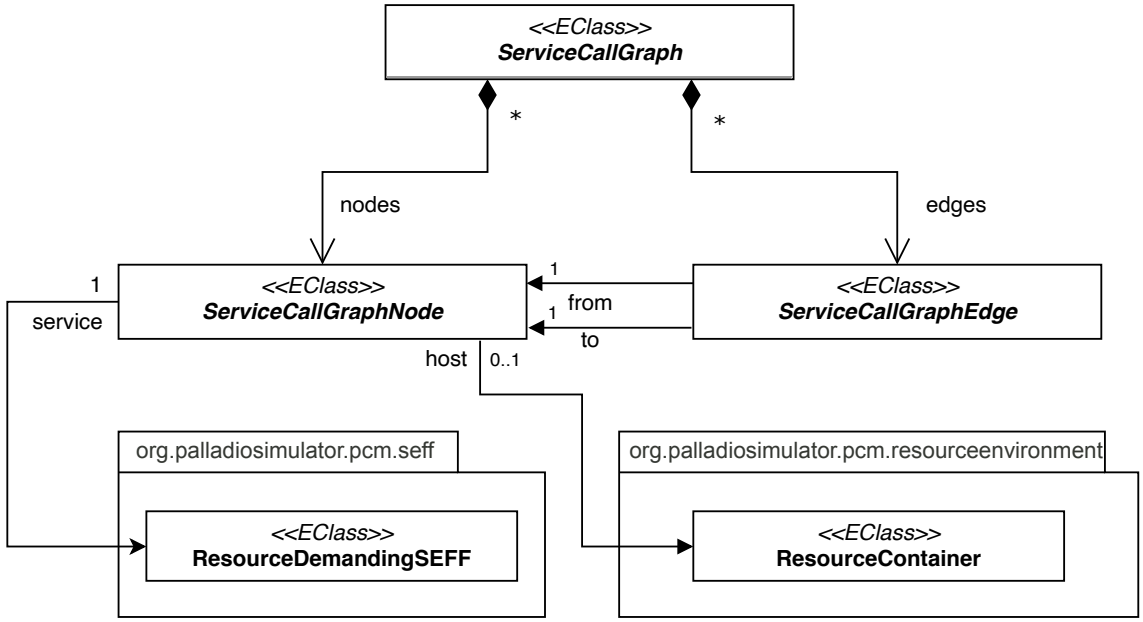


Figure 4.8.: Structure of the Service-Call-Graph Metamodel based on the Eclipse Modeling Framework (EMF)

The structure is straightforward, the *ServiceCallGraph* class serves as root container and represents the whole graph. It consists of an arbitrary number of *ServiceCallGraphNode* and *ServiceCallGraphEdge* elements, which represent the nodes/edges of the graph. A *ServiceCallGraphNode* must have a reference to a service in the Repository Model and optionally a reference to a resource container in the PCM Resource Environment Model.

The reference to a resource container is not mandatory, as we may not yet have information about the resource environment (for example at design-time). A *ServiceCallGraphEdge* refers to two *ServiceCallGraphNode* elements that represent the start and end of the edge.

4.6. Design-Time

First, Section 4.6.1 gives an overview of design-time activities and strategies for maintaining consistency, followed by Section 4.6.2, which presents the concept for extracting a System Model at design-time.

4.6.1. Overview

Our approach considers different strategies to ensure consistency between source code and architecture model at design-time. Figure 4.9 shows an overview of the parts of our approach that contribute to maintaining consistency.

A key component of this is Langhammer's co-evolution approach [50], which has been integrated. This makes it possible to keep the Repository Model, together with the Correspondence Model, synchronized with the source code. Monitors are used to observe the architecture model and the source code. The instrumentation process, which was already prototypically implemented in a previous work [17], was extended. The extensions of the instrumentation process have already been introduced and explained in Section 4.4.3.

Furthermore, an entirely new process for extracting a system model has been added. It can be triggered manually by a developer or an architect. The process is not completely automated, i.e. a "user" is needed, who knows the application under investigation. This procedure is introduced and discussed in detail in the following Section 4.6.2.

4.6.2. System Composition Derivation

The extraction of the system composition is based on the previously introduced Service-Call-Graph (SCG) structure (cf. Section 4.5). It is not a fully automated process, a developer who knows the structure of the system must be available to resolve conflicts. Why and to which extent this is necessary is explained in more detail below. Additionally, it is necessary that a consistent Repository Model is available together with a correspondence model which maps the elements from the Repository Model to the source code (and vice versa, i.e. bidirectionally). For example, these two models can be extracted automatically using the approach of Langhammer [50]. The derivation of the system composition can be divided into two tasks: the extraction of an SCG from the source code and the construction of the System Model from the SCG.

SCG Extraction As a starting point we use a points-to analysis based on the Soot framework [73]. The goal is to build a SCG that reflects which services call each other. To do this, we examine all services of the Repository Model and resolve the corresponding parts in the code. For all subordinate external calls we perform a points-to analysis on

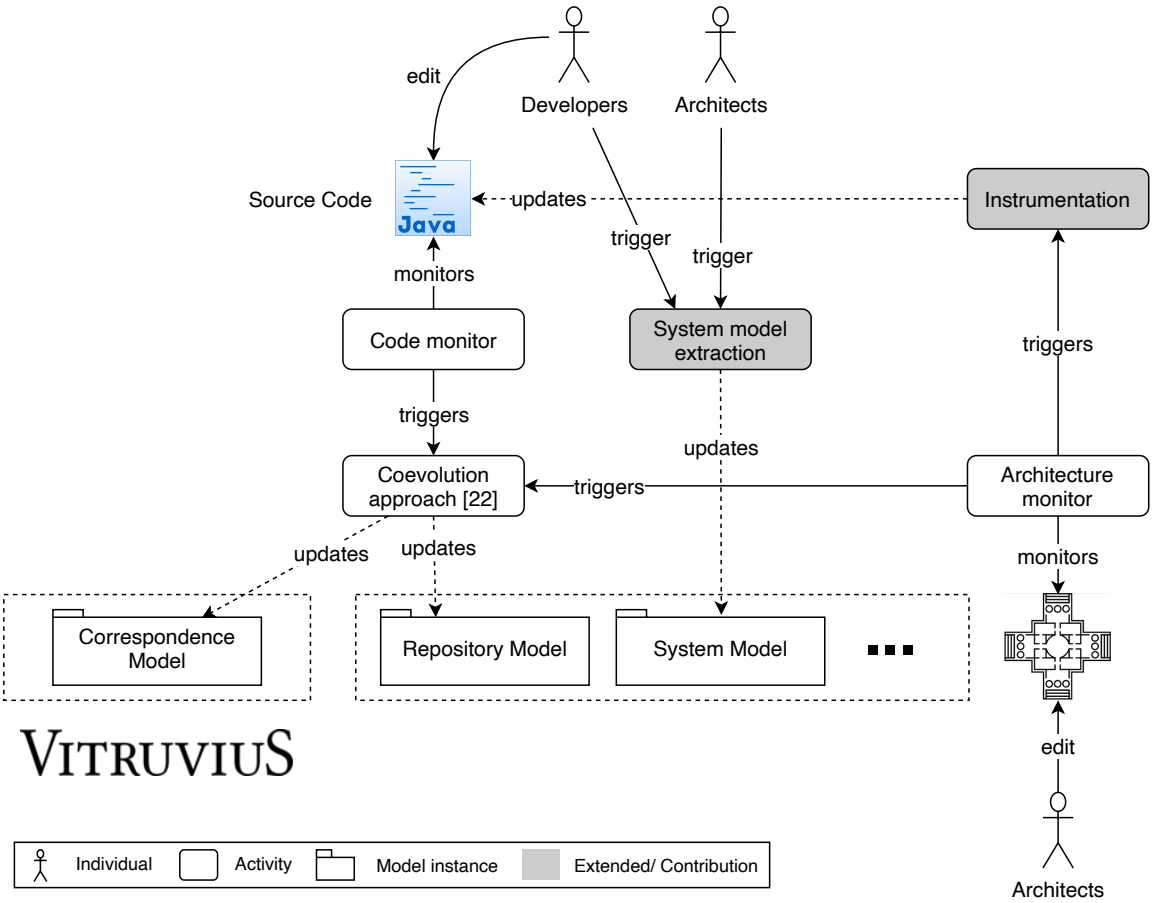


Figure 4.9.: Activities and strategies that are used to maintain consistency between source-code and architectural models at design-time; based on the co-evolution approach of Langhammer [50]

the code. The result is a list of methods that might be called at runtime. We map these methods back to services in the Repository Model using the Correspondence Model. We can then use this information to construct a SCG. A very important characteristic of such analysis is that we tend to overestimate in a points-to analysis and in static code analysis in general. In fact, we often do not know exactly at design time which methods are called at runtime. This is an important difference between design-time and run-time as introduced in Section 3.2. This is one reason why we need the support of a developer to deduce the system composition. We also leave out information about the hosts in the generated SCG, since this information is usually not available at design-time. When deriving the system composition, this also leads to situations where a developer has to intervene. To illustrate and understand the points-to analysis we use a small code snippet from the running example which is shown in Listing 4.2.

```
1 public class CachedPrimeServer implements PrimeServer {
2   private PrimeGenerator generatorService;
3   private PrimeDB database;
4
```

```
5   @Override
6   public int[] providePrimes(int upperLimit) {
7     if (database.getCurrentUpperLimit() < upperLimit) {
8       int[] primes = generatorService.calculatePrimes(database.getCurrentUpperLimit
            (), upperLimit);
9       database.storePrimes(primes);
10    }
11    return database.requestPrimes(upperLimit);
12  }
13 }
```

Listing 4.2: The *CachedPrimeServer* component from the Running Example implemented in Java

To build the SCG for this example the following steps are executed:

1. We iterate over all services of the *CachedPrimeServer* component. It only contains the service *providePrimes* in our example. To do so, we use the repository model which is the prerequisite for the system composition extraction.

2. We create a node for the service without an associated host in the SCG.

3. We use the correspondence model to find the associated parts of the service in the source code. This is the part that spans from line 6 to 12 in Listing 4.2.

4. Then we search this part of the code for external calls, i.e. for method calls which are defined outside the current class. As result we obtain the four calls "*database.getCurrentUpperLimit*" (lines 7 and 8), " *generatorService.calculatePrimes*" (line 8), "database.storePrimes" (line 9) and *database.requestPrimes* (line 11).

5. As these are calls to an interface, we perform a points-to analysis for the instance variables *database* and *generatorService*. This gives us a list of types that the instance variables can have at runtime. The list is overestimated, i.e. it is not guaranteed that all these types will actually occur at runtime. For each type in the list, we resolve the method that would be called and check whether a corresponding service can be found in the correspondence model.

6. Finally, we create a node in SCG for each of the services found in the previous step (if it does not already exist). Additionally, we create an edge from the *providePrimes* service of the *CachedPrimerServer* component to the newly created nodes.

The resulting SCG is shown in Figure 4.10. It should be mentioned, that no information about the associated hosts is embedded here (c.f. Section 4.5). The extraction of a SCG from the source code using a Repository Model and a Correspondence Model is fully automated. The assistance of a person who knows the system structure is only required in the subsequent steps.

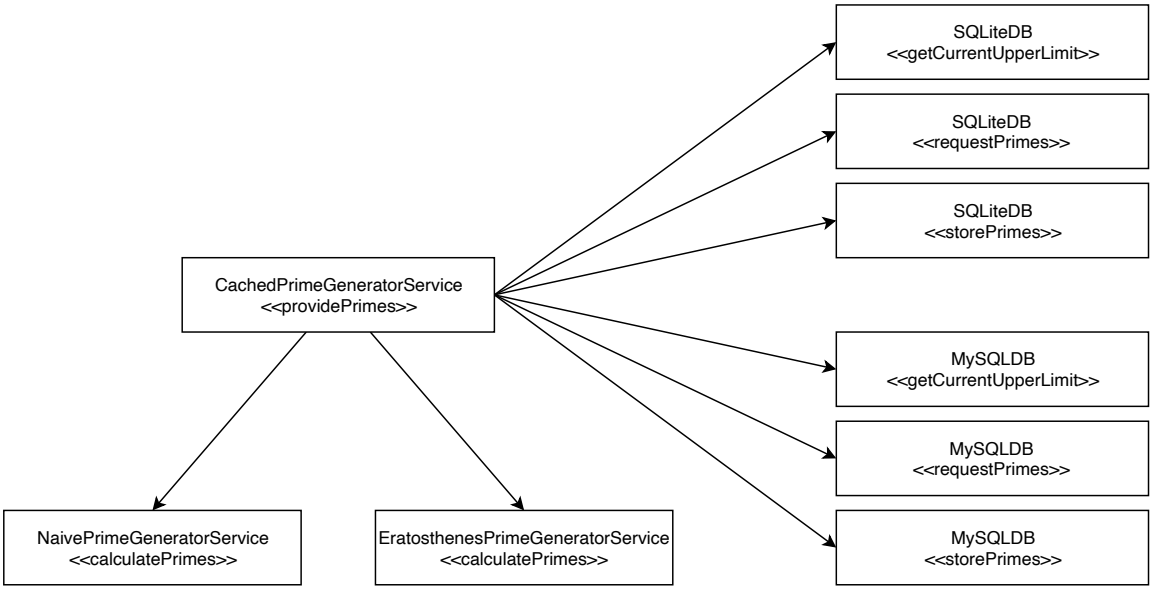


Figure 4.10.: Sample Service-Call-Graph (SCG) for the service *providePrimes* of component *CachedPrimeGeneratorService* (see running example in Section 4.2), extracted by using code analysis

Construction of the System Model In the second stage, the extracted SCG is used to build a System Model. As mentioned before, we need the help of a developer who knows the structure of the system at least to a large extent. Primarily, this developer has to resolve conflicts which result from the lack of information at design-time. In the following we will refer to the developer as "user".

First, the user has to specify which interfaces should be provided by the system in the end (the so-called provided roles). For each of the specified interfaces, the Repository Model is used to determine all components that offer this interface. If there are several components that offer this interface, the user decides which one is actually used. Then, we initialize an assembly context in the System Model for each of the selected components and create the delegations for the provided interfaces.

In the next step, the required interfaces of the selected components are examined (the so-called required roles). To satisfy these required roles the previously built SCG is used. All called services of the component are analyzed and assigned to the corresponding required roles. The services will then be grouped together to the components to which they belong. If there are various components available for a required role, there is a so-called **Connection Conflict**. The user has to resolve this conflict by specifying the component to be used. Afterwards, it is checked whether there is already an assembly context of this component in the System Model. If this is the case, a so-called **Assembly Conflict** arises. The user has to decide whether a new assembly context should be created or which of the existing ones should be used. Subsequently, the provided role and the required role are connected with an *AssemblyConnector*. This procedure is repeated until all required roles are satisfied.

At this point the construction of the Repository Model is completed. Because the description of the derivation process is very theoretical, we have illustrated it using pseudocode in Algorithm 2. For the sake of simplicity, we assumed that a provided role and a required role belong directly to an assembly context, which is normally not the case in the PCM. To be correct, the provided and required roles would have to be determined using the component that belongs to an assembly context. But in our case this would just make the pseudo code more complicated and was therefore omitted.

An important part of the system model extraction is the source code analysis. The number of conflicts strongly depends on the quality and accuracy of it. Even with the best possible code analysis, not all conflicts can be solved automatically. Later on in the evaluation, this assumption is confirmed (see Section 5). It follows from the nature of the difference between design-time and run-time that this process cannot work without potential conflicts (see Section 3.2).

Algorithm 2 System Model Extraction at design-time using a Service-Call-Graph

Input: Repository Model (REPO), Correspondence Model (CM), Service-Call-Graph (SCG)
Output: PCM System Model (result)

```
1:  result = createEmptyPCMSystemModel()
2:  openProvidedRoles ← emptyList()
3:  openRequiredRoles ← emptyList()
4:  outerProvidedInterfaces ← askUserForSystemProvidedInterfaces()
5:
6:  for all opr ∈ outerProvidedInterfaces do
7:      outerProvidedRole ← createSystemProvidedRole(opr)
8:      innerProvidedRole ← supplyProvidedRole(outerProvidedRole, null)
9:      createProvidedDelegation(outerProvidedRole, innerProvidedRole)
10: end for
11:
12: while openRequiredRoles.size > 0 do
13:     currentRequiredRole ← openRequiredRoles.pop()
14:     r ← currentRequiredRole
15:     currentProvidedRole ← supplyProvidedRole(r, r.component)
16:     createAssemblyConnector(currentRequiredRole, currentProvidedRole)
17: end while
18:
19: function SUPPLYPROVIDEDROLE(target, source)
20:     selectedComp ← supplyComponent(target, source)
21:     existingProvRoles ← emptyList()
22:
23:     for all role ∈ openProvidedRoles do
24:         if role.component = selectedComp & role.interface = interface then
25:             existingProvidedRoles.add(role)
26:         end if
27:     end for
28:
29:     if existingProvRoles.size > 0 then                ▷ Assembly Conflict
30:         resultRole ← askUserForExistingRoles(existingProvRoles)
31:         if resultRole <> null then
32:             return resultRole
33:         else                                  ▷ User decided to create a new one
34:             return supplyNewAssemblyContext(selectedComp, interface)
35:         end if
36:     else
37:         return supplyNewAssemblyContext(selectedComp, interface)
38:     end if
39: end function
```

```
40: function supplyNewAssemblyContext(selectedComp, interface)
41:     newAssembly ← createAssemblyContext(selectedComp)
42:     for all providedRole ∈ newAssembly.providedRoles do
43:         if providedRole.interface = interface  &  result = null then
44:             result ← providedRole
45:         end if
46:         openProvidedRoles.add(providedRole)
47:     end for
48:     for all requiredRole ∈ newAssembly.requiredRoles do
49:         openRequiredRoles.add(requiredRole)
50:     end for
51:     return result
52: end function
53:
54: function supplyComponent(role, source)
55:     possComps ← getComponentsWithProvidedInterface(role.interface)
56:     filteredComps ← filterComponents(possComps, SCG, role, source)
57:                 ▷ Filter the conforming components by using the Service-Call-Graph
58:
59:     if possComps.size > 1 then                          ▷ Connection Conflict
60:         return askUserForComponent(possComps)
61:     else if possComps.size = 1 then
62:         return possComps[0]
63:     else
64:         return
65:     end if
66: end function
```

4.7. Run-Time

Section 4.7.1 first gives an overview of the design of the consistency maintenance at run-time. Afterwards, Section 4.7.2 explains how the monitoring data is collected and grouped, Section 4.7.3 describes how the simulations of the PCM architecture models are performed. Next, Section 4.7.4 illustrates how the self-validations were woven into the approach as a central building block. Section 4.7.5 introduces the Runtime Environment Model (REM), which is used to represent the runtime environment, consisting of hosts and links between them.

4.7.1. Overview

The core at run-time is a transformation pipeline that processes the monitoring data (similar to iObserve [29]). An important feature of our approach is that the simulation and validation of the derived models are an integral part. The results of the validations are used as input for the transformations. This enables them to adapt and for example to perform a strategy change. This characteristic is called Validation Feedback Loop (VFL) hereinafter and is described in more detail in Section 4.7.4.

Figure 4.11 provides an insight into the architecture of our approach which we use at run-time.

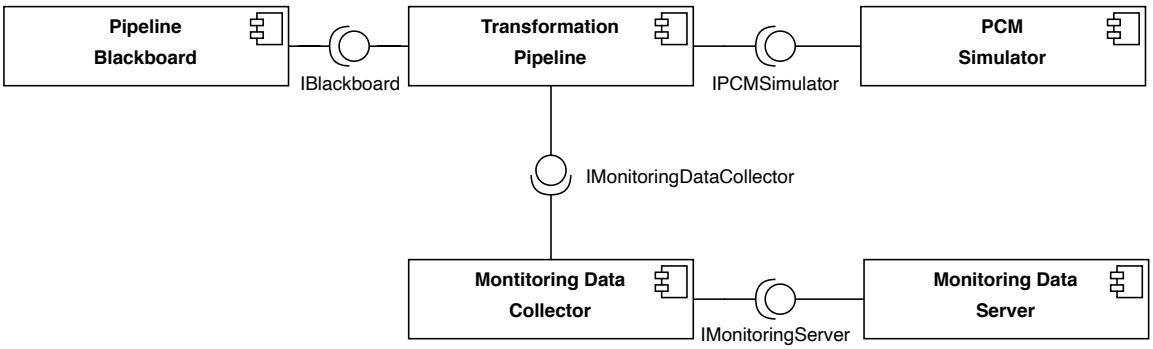


Figure 4.11.: Overview of all components that are involved at runtime

As already mentioned, the focus is on the transformation pipeline, which processes the monitoring data and derives updates of the models. In addition, the transformation pipeline uses a blackboard, which can be written and read by all transformations. The following data is stored in the blackboard:

- Current PCM instance
- Resource Environment Model (REM)
- Simulation and validation data
- Current state of the transformation pipeline

The REM reflects the run-time environment with all participating hardware. The structure of this model is described in Section 4.7.5. At different points within the transformation pipeline the current models are evaluated and compared to the monitoring data (further information is provided in sections 4.7.4 and 4.8). The results are written to the blackboard so that they are available for the transformations. To perform the validation, the transformation pipeline must be able to simulate the current PCM models. For this purpose an isolated simulator is used, which is accessed via a RESTful interface [21]. Section 4.7.3 deals with the simulator in more detail. The monitoring data is received and bundled by a server. Kieker is used as monitoring framework and the records are sent from the monitored application to the server via TCP. The Monitoring Data Collector then preprocesses these and partitions them using a sliding window mechanism. How exactly this mechanism works is described in Section 4.7.2. The partitioned monitoring data is then used as input for the transformation pipeline.

4.7.2. Monitoring Data Collector

The Monitoring Data Collector is a very important component for the performance of the approach as a whole. If the transformation pipeline is executed very frequently or triggered with too much monitoring data, this could lead to a performance decrease of the pipeline. More monitoring data also means more information in most cases, nevertheless it does not make sense to keep using very old monitoring data as input for the transformation pipeline.

To acknowledge the importance of this component, we used the strategy pattern. As a result, the strategy used to group the monitoring data is easily exchangeable. As standard procedure we implemented a sliding window mechanism. It uses a sliding window whose size is a customizable time interval (*WindowSize*). The mechanism is demonstrated in Figure 4.12 in a simplified form. The transformation pipeline is not executed permanently, instead it is always executed after a certain time (*TriggerSize*) with the monitoring data which is currently in the sliding window. It is important to ensure that the following is true: $TriggerSize < WindowSize$. Otherwise, some monitoring data will not be considered at any time. On the other hand, if this assumption is valid, all monitoring data is considered several times. For example, if $WindowSize = 2 * TriggerSize$, then all monitoring data is analyzed exactly twice. This is an important point to note, because it means that all transformations in the pipeline must be robust against multiple occurrences of the same monitoring data. Furthermore, at no time is it guaranteed that the current monitoring data is complete. For instance, it can often happen that a trace is not yet completely reflected in the monitoring data, especially in distributed systems that are monitored. Therefore, this must be taken into account when designing and implementing the transformations.

Furthermore, we split up the data within the sliding-window, before using it as input for the pipeline. The data is divided into two groups: first the *training data* and second the *validation data*. The purpose of this is to avoid falsifying the self-validations within the pipeline. If we execute all transformations on the complete data and then execute a validation on the same data, this is not informative. For example, it would be possible to adjust the model so that exactly the values from the data are reproduced. In this case, the self-validation assumes a high model quality, which is not intended. The ratio between

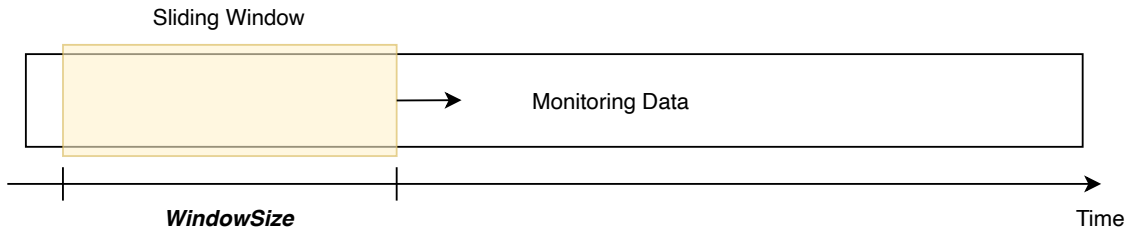


Figure 4.12.: Illustration of how the sliding window mechanism for partitioning the monitoring data works

training and validation data can be customized according to the specific use case. During the splitting process, care is taken to ensure that no sessions are fragmented, otherwise this could lead to incorrect results (e.g. Usage Model transformation).

At this point it is important to note that some transformations are always executed on all data (training data + validation data). This includes, for example, those transformations that are responsible for extracting the system composition and the resource environment. Here it does not make sense to distinguish between training and validation data, since all data is needed to derive consistent models.

4.7.3. PCM Simulator

The simulations of the PCM models are performed completely isolated from the approach itself. This ensures that simulation strategies remain easily exchangeable and expandable in the future. The transformation pipeline accesses the simulation component via a RESTful interface [21]. Using this interface the models are transferred and the simulation results are retrieved. We use a headless implementation for the simulations because it is mostly independent of Eclipse and therefore very lightweight[2]. The simulation component can also be virtualized (e.g. using Docker). Furthermore, the parameters of the simulations are fully configurable (simulation engine, simulation time, number of measurements, etc.).

4.7.4. Validation Feedback Loop

An important innovation of the approach is the so-called Validation Feedback Loop (VFL). The idea is that the behaviour of the transformations does not have to be strictly static. Based on the accuracy of the current models it should be possible for the transformations to adapt. For most approaches there are scenarios where the accuracy of the models is constantly very poor. Since these inaccuracies are not known without a proper validation procedure, the models remain permanently inaccurate, i.e. inconsistent. Moreover, this approach opens the door to machine learning techniques which could be used, for example, to learn from inaccuracies in the past. In addition, the transformations could adapt individually to a certain application and “learn” its behavior.

The VFL covers all processes involved in the simulation and evaluation of the models. In this section, we will only explain the process of the model validation and not at which points

[2]https://github.com/dmonsch/PCM-Headless

in the transformation pipeline it is executed. This will be discussed in the following Section 4.8. To determine the accuracy of a model, it is always necessary to have comparable values as a reference. We assume that the monitoring data reflects the reality and therefore use them as a reference. Figure 4.13 displays the interaction of the VFL and the transformation pipeline graphically. The simulation results are grouped into so-called measuring points, i.e. the points at which measurements were taken. For example, a typical measuring point is the CPU load of a computer. In addition, the type of the associated metric is recorded for each measurement (e.g. response time in seconds). To be able to compare the simulation results with the monitoring data, we have to map the monitoring data to the corresponding measuring points in an upfront step.

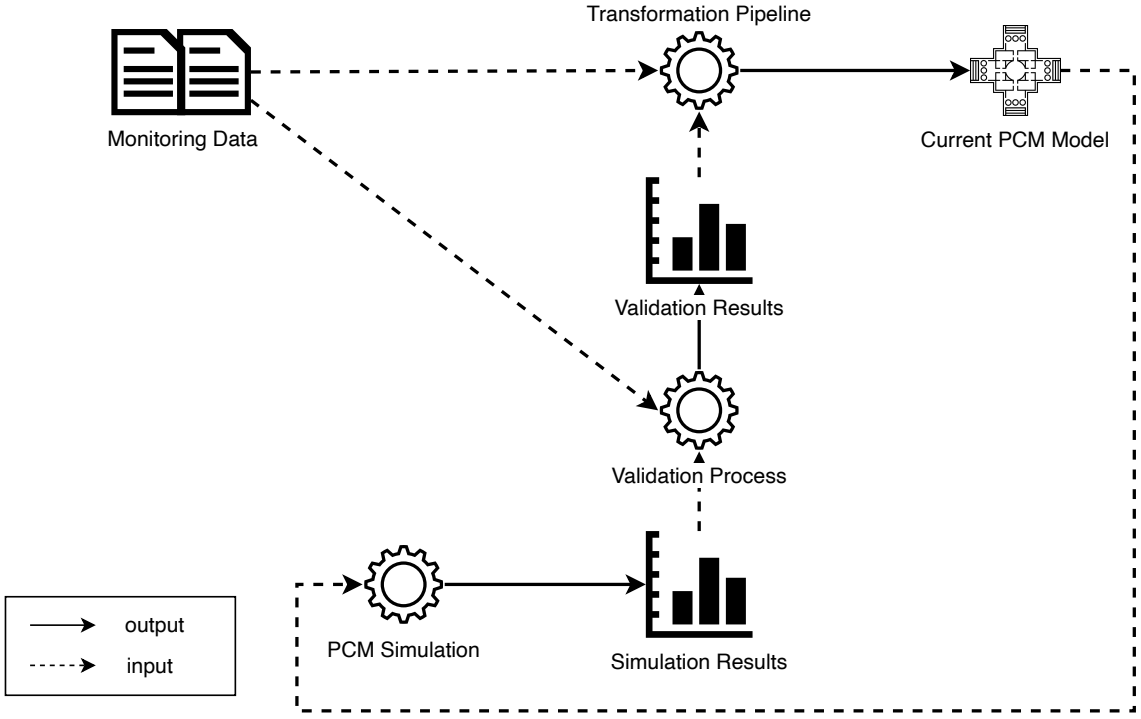


Figure 4.13.: Illustration of how the Validation Feedback Loop (VFL) works in combination with the transformation pipeline

After the monitoring data has been mapped to the corresponding measuring points of the simulation results, we have two distributions for each measuring point. These are compared and different metrics are calculated to determine how close the simulation results are to the actual measured values. We have used the following metrics to compare the two distributions:

- Wasserstein distance [66]
- Kolmogorov–Smirnov test (KS test) [41]
- difference of conventional statistical measures:
  - average

- minimum and maximum
- standard deviation and variance
- quartiles (Q1, Q2, Q3)

The implementation was done in such a way that it is easily possible to add further metrics. The KS test measures the discrepancy between two distributions using the maximum distance between the Cumulative Distribution Functions (CDFs). Usually it is used to check whether an empirical distribution conforms to a theoretical assumption. Therefore it is not the perfect metric for our use case, but gives a good indication of how much simulation and monitoring differ. The Wasserstein distance on the other hand is very well suited for our use case, but gives only an absolute value that is difficult to interpret.

Because we also compare the accuracy of two models within the transformation pipeline, we need to combine these metrics into one metric. This is very important for the approach as a whole, because it can be used, for example, to make statements about whether the accuracy of the models is increasing or decreasing. This metric has to express which model does represent the monitored behaviour better. We used a simple method to calculate this metric. We compare the metric values pairwise and assign one point for each metric to the more accurate model. So we only need to be able to decide for each metric which value is the better one. Finally, the model that scores more points is considered more accurate. The priority of the individual metrics can be configured as required, which is especially helpful because we want to compare the values of different measuring point types. For example, when comparing CPU utilizations, the meaningfulness of the metrics is significantly different than when comparing response times. A more mathematical representation of this comparison is shown below:

$$CompareMetrics(M1, M2) = \sum_{i=1}^{|M1|} w_i * sign(compare(M1_i, M2_i))$$

w	Priorities of the individual metrics
$M1$	set of metrics
$M2$	set of metrics with the same size as M1
$compare$	calculates the distance, taking into account the properties of the metric

4.7.5. Runtime Environment Model (REM)

The Runtime Environment Model (REM) is mainly used for two reasons: first, to create the mapping between the resource containers in the PCM and the computers in the physical run-time environment. Second, to keep the properties of the mapped elements consistent (for example, the number of the CPU cores). The REM is generated at run-time from monitoring data and then synchronized with the PCM resource environment model using Vitruvius.

The corresponding metamodel of the REM is outlined in Figure 4.14.

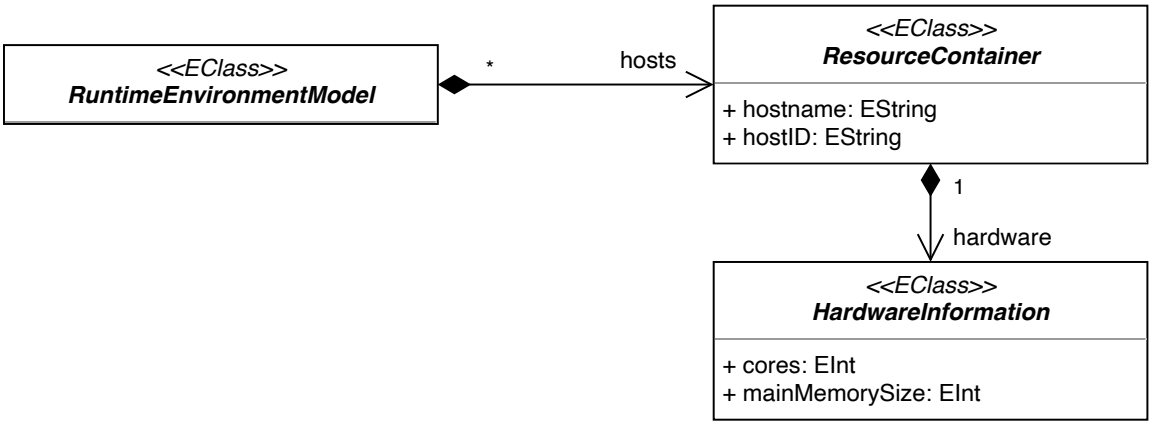


Figure 4.14.: Structure of the Runtime Environment metamodel based on the Eclipse Modeling Framework (EMF)

Like the Resource Environment Model of the PCM, the REM also consists of resource containers (*ResourceContainer*). These have a host name and also a "host ID". In contrast to the host name, the host ID should remain unchanged over a longer period of time. In the implementation we used a hash of the MAC address of the respective machine. However, the actual strategy used for the generation is not relevant in this context. The idea is that the mapping between the resource containers in the PCM and the machines in the REM is established just once via the host name. Subsequently, the mapping is maintained based on the host ID. We therefore assume that at the time of deployment, the names of the resource containers in the PCM match the host names of the associated computers in the physical environment. But this assumption can be slightly softened, as it is only necessary for those containers that have not yet been mapped. In addition, hardware information is assigned to each container in the REM (*HardwareInformation*). This information is currently incomplete and is intended for future work. The purpose of this is to keep the properties of the containers consistent (e.g. number of CPU cores, main memory size etc.). At the moment we only considered the number of cores and the size of the main memory, as a proof of concept. In the future, this could be extended to enhance the automated consistency maintenance.

4.8. Transformation Pipeline

The transformation pipeline is the main component that is used at run-time. It contains the complete logic for updating the architectural model parts. We used a tee and join pipeline architecture [15] and complemented it with a blackboard that can be written and read by all transformations.

4.8.1. Overview

Figure 4.15 shows the entire transformation pipeline in a very simplified version.

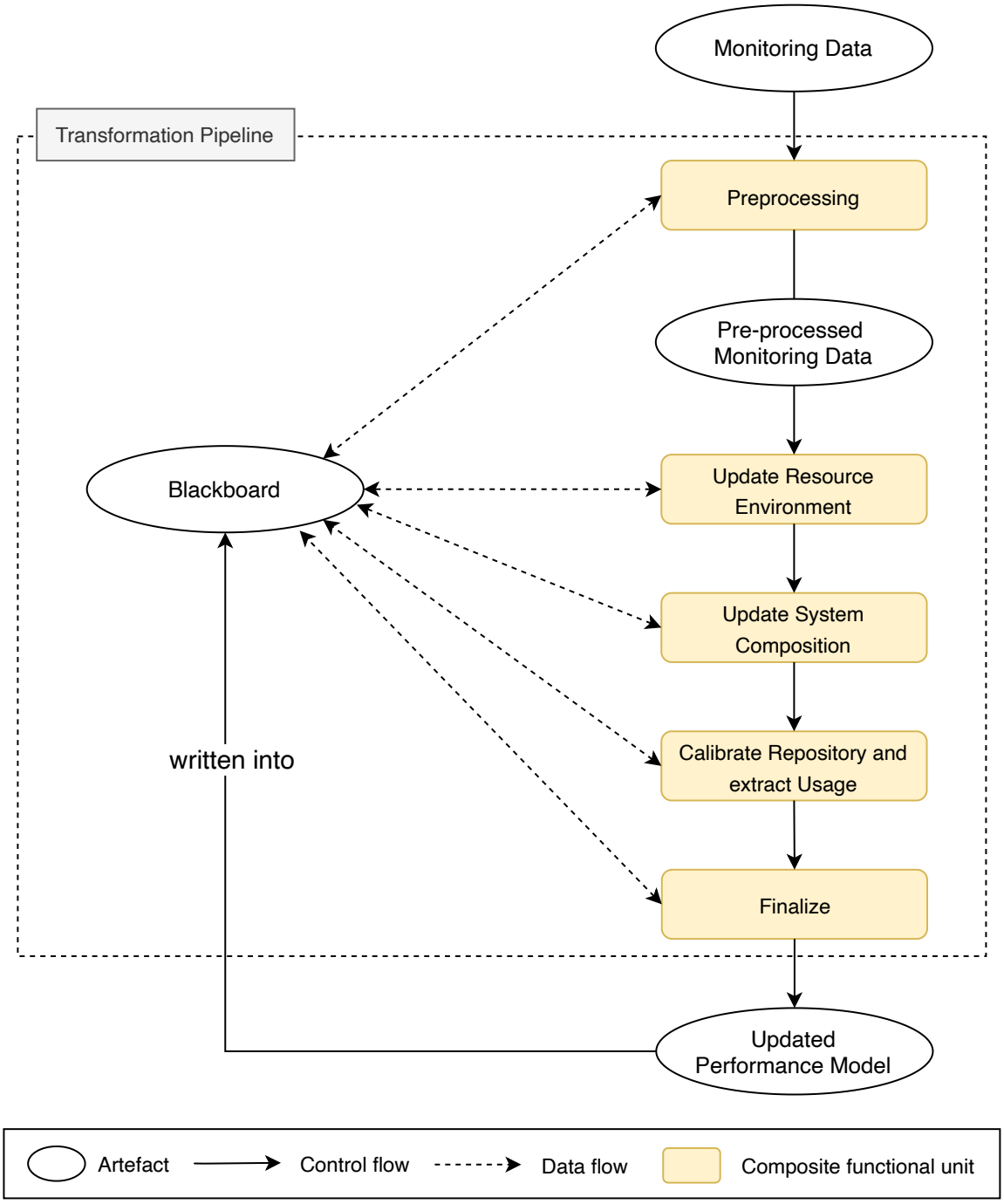


Figure 4.15.: Overview of the transformation pipeline structure which is triggered at runtime

Within the pipeline, the architectural model is simulated and validated several times. For this reason, we have completely encapsulated the simulations from the pipeline and use a headless simulation approach to keep the effort as low as possible (see Section 4.7.3).

The pipeline is structured as follows:

1. In a pre-processing step (*Preprocessing*), the monitoring data is filtered and converted into suitable data structures. Furthermore, the initial PCM instance is simulated and validated. The exact structure and functionality of this functional unit is explained in more detail in Section 4.8.2.
2. After the monitoring data has been pre-processed, the current run-time environment is analyzed. This step is based on Vitruvius and the already introduced Runtime Environment Model (REM). The procedure is discussed in Section 4.8.3.
3. Next, the system composition and the associated deployment is examined. Compared to iObserve, this is a major innovation, iObserve only considers the deployment, not the system composition. The exact design and implementation of this procedure is introduced in Section 4.8.4.
4. In the next step, the stochastic expressions in the Repository Model are calibrated and the user behavior is evaluated. This functional unit is much more complex than it appears at first glance. Section 4.8.5 describes the internal structure.
5. In the final step (*Finalize*), the updated model is put together and validated one last time. In addition, the Instrumentation Model is adjusted based on the validation results. This step is explained in more detail in Section 4.8.6.

4.8.2. Preprocessing

In the pre-processing step, the monitoring data is filtered and transformed into suitable data structures. In addition, the current model is simulated and validated. The exact structure is shown graphically in Figure 4.16. The results of the preprocessing step are the

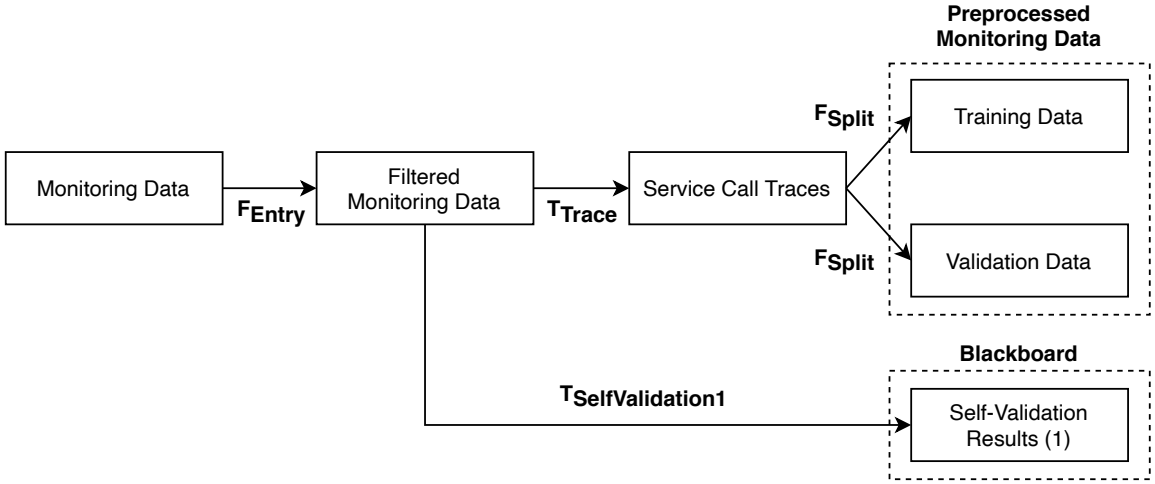


Figure 4.16.: In-depth view of the transformations that are executed during the preprocess step

partitioned monitoring data (training and validation data) and the outcome of the first self-validation.

In the following, all involved transformations are briefly introduced and explained.

F_{Entry} In this pre-processing step, we simply filter out all monitoring data that is not needed and/or used in the pipeline. In addition, parts of the blackboard are reset and initialized for the subsequent transformations.

$T_{SelfValidation1}$ This is the first of four validation steps. The current PCM is simulated and compared with the monitoring data, as described in Section 4.7.4. The validation results are written to the blackboard and can be read by the subsequent transformations.

T_{Trace} Within this transformation, the service calls are combined into traces. These can also span several hosts using the request tracing technique (see Section 4.4.4). All subordinate information (such as the execution time of an internal action) is then attached to the service calls in the traces. Due to the sliding window mechanism, it is possible that we notice incomplete traces or traces where the root call is no longer present in the monitoring data. Traces where the root call is missing are not considered and discarded. Furthermore, the subsequent transformations are designed to be robust against incomplete traces. The procedure for building the traces is quite simple, it iterates over all monitoring records and the service calls are assigned to their parent via the *callerExecutionID*. If additional data about the execution of a service (internal actions, loops, branches) is available, this is simply assigned to the corresponding service call.

An exemplary service call trace could look like this:

- Service Call Record
  - Internal Action Record
  - Internal Action Record
  - Service Call Record
    - Internal Action Record
    - Branch Action Record
    - Loop Action Record
    - Internal Action Record
    - ...

F_{Split} This filter transformation splits the data into training and validation data (see Section 4.7.2). The ratio between training data to validation data is customizable.

4.8.3. Resource Environment Update

This functionality consists of a single transformation ($T_{ResourceEnvironment}$). It modifies the Runtime Environment Model (REM) based on the monitoring data. With the help of Vitruvius, the defined consistency rules automatically maintain the consistency of the respective Resource Environment Model.

As input, the transformation requires all monitoring data (training and validation data). First, we write all hosts and their associated information that appear in the service call

traces into the Runtime Environment Model (REM). If a host does not yet exist in it, it is created. A host in the REM is addressed by its host ID, which is designed to be unique over a long period of time. The consistency rules which are executed after each change are used to update the corresponding Resource Environment Model. This process is realized with Vitruvius. We recognize connections between several hosts by searching within the traces for host changes. If there is a change, we can conclude that there must be a connection between two containers.

A sample service call trace that contains a host change looks similar to this example:

- Service Call Record (Hostname: **Logic Server**)
  - ...
  - Service Call Record (Hostname: **DB Server**)

Figure 4.17 summarizes once again the procedure for updating the resource environment.

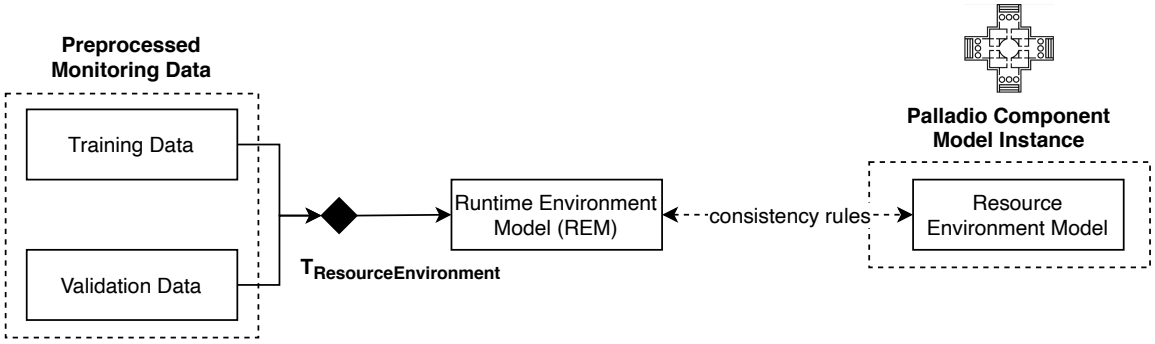


Figure 4.17.: Graphical overview of the process for updating the Resource Environment Model

In the following transformations we assume that the Resource Environment Model is consistent with the actual run-time environment.

4.8.4. System Compositon Update

This functional unit is responsible for updating the System Model and the Allocation Model. However, this can only be accomplished if the assumptions **A2** and **A3** from Section 3.4.2 are satisfied. Figure 4.18 shows all involved transformations.

$T_{ServiceCallGraphs}$ The procedure for deriving the system composition at run-time is similar to the procedure which is used at design-time. For each service call trace, we build a Service-Call-Graph (SCG) which reflects the calls-to relationships between the services. In contrast to the SCG we are building at design-time, we also attach the information about the host to the nodes in the SCG. We also update the Allocation Model within this transformation by analyzing the information about the corresponding hosts.

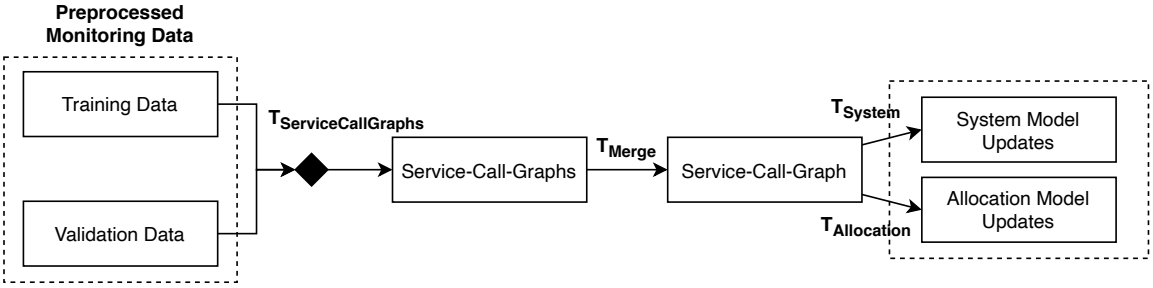


Figure 4.18.: Detailed view of the transformations within the functional unit that is responsible for deriving the updates of the System Model and the Allocation Model

T_{Merge} The next step is to combine all SCGs into one. For this purpose, we create an SCG that contains all nodes and all edges from all SCGs to be merged. In addition, the time of their appearance is added to all nodes and edges as metadata. This allows us to resolve conflicts in later stages automatically by preferring the more recent information.

$T_{Allocation}$ For each node in the SCG, the corresponding assembly context in the current System Model is resolved. This works as follows: a node in SCG is a pair of a resource container and a service. First, we determine the component to which the service belongs. Then we check all assembly contexts that are allocated on the container to see if they match this component type. According to our assumption (**A3**), this can be a maximum of one assembly context. If there is no associated assembly context, we create a new one with the respective component type and also create a corresponding allocation context in the Allocation Model. The newly created assembly contexts will additionally be flagged as "new".

T_{System} Next, the actual derivation of changes in the system composition begins:

1. We start at the entry nodes of the merged SCG and examine all outgoing edges. The "entry nodes" are all nodes with input degree 0. We assume that there are always nodes without incoming edges. If this is not the case there is a loop within the SCG, but we decided to exclude these scenarios because they are very unlikely to occur.

2. It is checked whether the source node is a newly created assembly context. The same is done for the target node.

3. If the associated assembly context is marked as new for both nodes, we identify the associated provided role and the associated required role and connect them with an assembly connector.

4. If only one or none of the two associated assembly contexts is a newly created one, we determine the associated provided role and the required role as before and check whether an assembly connector already exists for one of them. If so, we delete the previous one and finally create a new assembly connector that connects the required role and the provided role.

5. We continue this procedure recursively for all target nodes until all outgoing edges of the SCG are processed.

6. We check for each provided role of the system which open provided roles fit to it. Then we select one of these provided roles, preferring those that belong to a "new" assembly context. Here we use the metadata that we annotated during the merging of the SCGs. In this way it is possible to always select the role that was used most recently. Then we create or adjust the delegation from the outer provided role of the system to the one of the selected assembly context.

7. Finally, we start with each provided role of the system and trace the paths through the System Model. We do this by following the connectors of the system. This allows us to locate assembly contexts that are not reachable and these are marked as deprecated. We chose a soft approach, meaning that not all assembly contexts are deleted immediately. It is possible to define conditions when an assembly context should be deleted. For example, it is possible to configure that an assembly context is only deleted if it is marked as unreachable for three consecutive pipeline executions. When an assembly context is deleted, the associated allocation context in the Allocation Model is also removed.

In order to make the procedure more clear, we will demonstrate it using a sample scenario from the Running Example (c.f. Section 4.2). Figure 4.19 shows an initial system model of the running example.

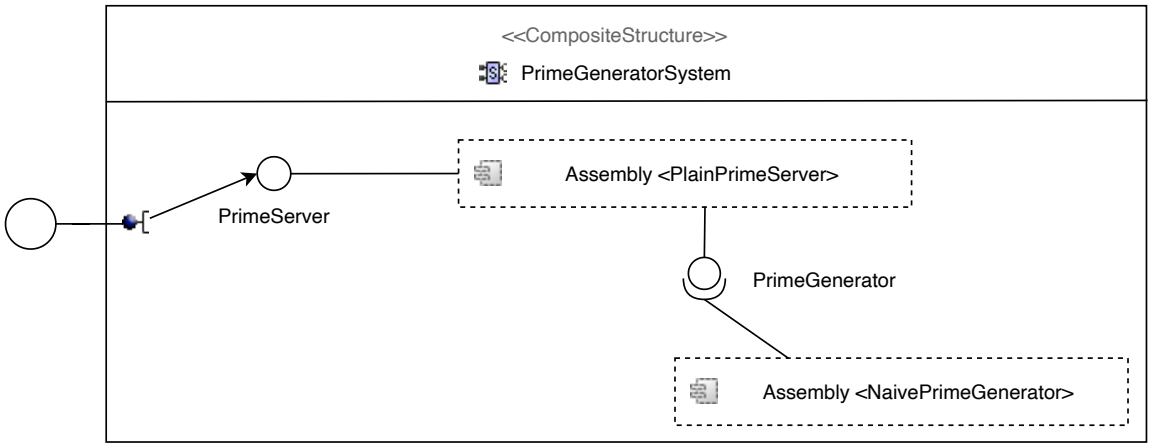


Figure 4.19.: Initial System Model taken from the running example

Also, we assume that both assembly contexts are deployed on the same container named “Server”. Assuming that we extracted the following service call trace from the monitoring data:

- Service Call (Component: PlainPrimeServer, Signature: providePrimes, Host: Server)
  - Serivce Call (Component: EratostehensPrimeGenerator, Signature: calculatePrimes, Host: Server)

This service call trace indicates that the service for generating the prime numbers has changed from the “NaivePrimeGeneratorService” component to the “EratostehensPrimeGeneratorService” component. Figure 4.20 shows the corresponding SCG.

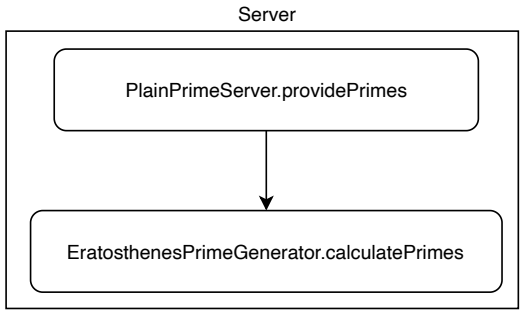


Figure 4.20.: Exemplary Service-Call-Graph (SCG) of the running example

Following the procedure described above, the first step is to find the assembly context for each node. For the root call we find the corresponding assembly context, for the node "EratosthenesPrimeGenerator.calculatePrimes" we have to create a new one. Then we start at the node “PlainPrimeServer.providePrimes” and process all outgoing edges. In our example, there is only one edge, which points to the “EratosthenesPrimeGenerator.providePrimes” node. Because only the corresponding assembly context of the target node has been newly created, we search the old connector (for the required role of the assembly context that belongs to the source node) and remove it. In the next step, a new connector is created, which connects the required role of the source node with the provided role of the target node. Finally, the assembly context “Assembly <NaivePrimeGeneratorService>” is marked as deprecated because it is no longer in use. The final updated system model is shown in Figure 4.21.

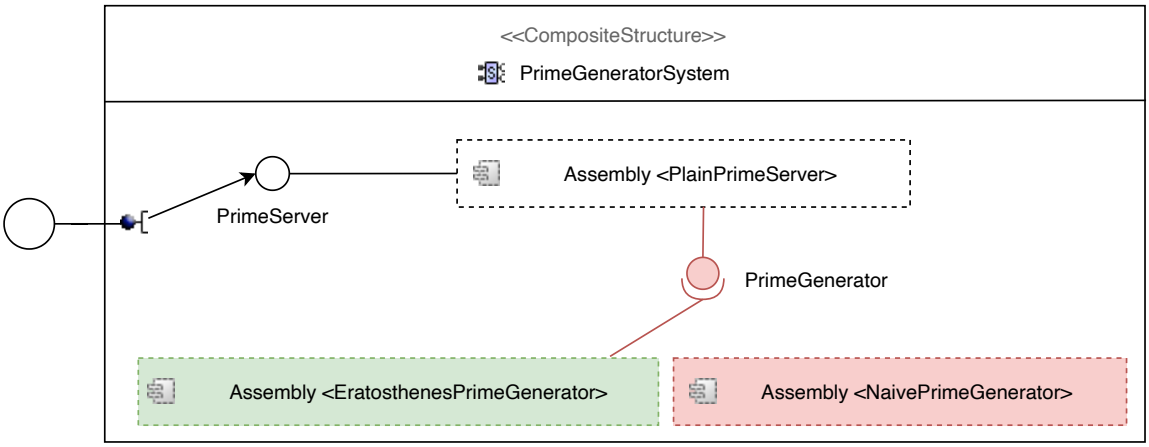


Figure 4.21.: Updated System Model after the exemplary execution of the $T_{SystemComposition}$ transformation

Algorithm 3 in the appendix shows the functionality of the transformations T_{System} and $T_{Allocation}$ in a more structured way with pseudocode.

4.8.5. Repository and Usage Model Update

Within this functional unit the Repository Model is calibrated and the user behavior is extracted.

Unlike iObserve, all transformations can access the current validation data. This leads to different characteristics that must be taken into account in the sequence and structure of the transformations. For example, the parallelism of transformations that use validation data can cause oscillations. To illustrate this with an example: during the validation it is recognized that the deviation between measured and simulation values is high. Therefore, transformation 1 adapts its behavior to overcome these inaccuracies. Transformation 2 is executed in parallel and also adapts its behavior. Under certain circumstances, it is possible that both transformations have counteracted each other and the resulting model is even less accurate. To avoid exactly these oscillation effects we first perform the transformations that consider the system composition and the runtime environment (see previous sections). For these we assume that they are very performant and therefore do not block the execution of the pipeline for a long time. Subsequently, the Repository Model and the Usage Model are updated.

In order to avoid the oscillation effects, we decided to use a more complex structure. Figure 4.22 summarizes the entire inner structure of the procedure for updating the repository and the usage model.

First, the training data is used as input for the calibration of the resource demands in the repository model ($T_{Repository}$) and for the transformation which extracts the user behavior (T_{Usage}). These two tasks are executed simultaneously. The exact behavior of these two transformations will be described in the following, since they are also relatively complex. $T_{SelfValidation2}$ and $T_{SelfValidation3}$ simulate and validate the updated models following the procedure as described in Section 4.7.4. $T_{SelfValidation2}$ validates the model where the Usage Model has already been updated and $T_{SelfValidation3}$ validates the model where only the repository has been calibrated. Next, the validation results are compared by means of $T_{CrossValidation}$ (also with the procedure which is explained in Section 4.7.4). According to the accuracy of the two models the $T_{Crosswise}$ transformation behaves different. If the model resulting from the execution of $T_{Repository}$ is considered to be superior, we execute the T_{Usage} transformation on this model. As input we use the validation results of $T_{SelfValidation3}$. Similarly, if the model resulting from T_{Usage} is more accurate, we execute $T_{Repository}$. As input we use the validation results from $T_{SelfValidation2}$. Ultimately, we get a set of updates for the Usage Model and for the Repository Model.

With the help of this structure we avoid the oscillation effects mentioned above, because the transformations do not operate on the same model simultaneously. In the first step, we can execute both transformations in parallel, which results in an additional overhead of one transformation execution (T_{Usage} or $T_{Repository}$ is executed twice) and two model simulations. Since the approach integrates the validations as an integral part, these simulations not only increase transparency across the pipeline but also provide more accurate input for the transformations. In addition, the validation is designed to minimize the impact on the pipeline performance.

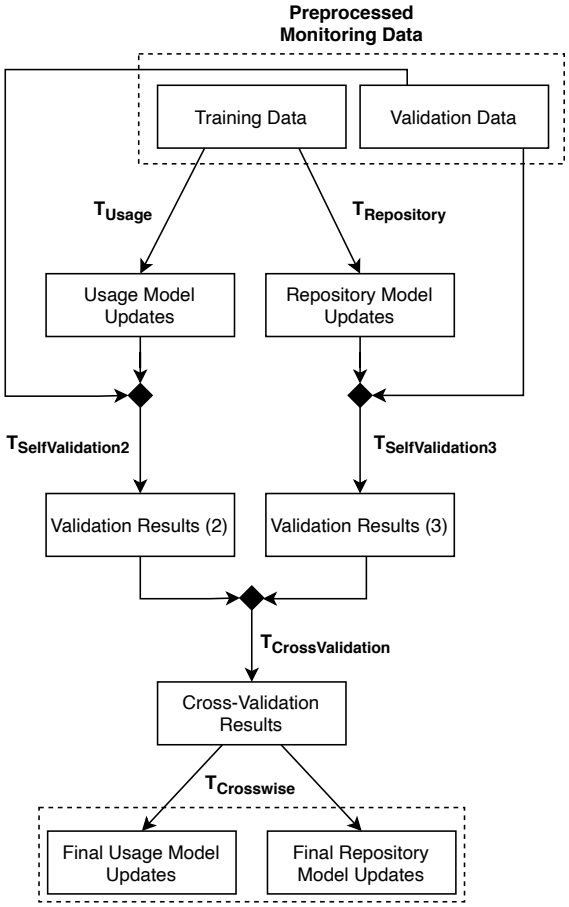


Figure 4.22.: Visual representation of the structure within the procedure for updating the Repository Model and the Usage Model

$T_{Repository}$ This transformation calibrates the stochastic expressions which describe the resource demands. The calibration is based to a large extent on the preliminary work in the context of the CIPM approach [56, 55]. It uses regressions and also considers dependencies on the parameters of the service calls. In order to make this procedure more dynamic, we have implemented several small extensions in our approach. An important characteristic of the customized calibration process is that we include the validation results.

The extended procedure works as follows:

1. Building a graph that reflects which services call each other. For this purpose, only the Repository Model is used. We iterate over all edges and create an edge for each external call within the services. This graph helps us to group the services according to levels. In addition, we annotate each node (service) of this graph to indicate whether it is currently instrumented with fine or coarse granularity.

2. Identify all nodes that are not yet instrumented coarse-grained under the condition that all nodes that are reachable via outgoing edges are already instrumented coarse-grained. We call this set of nodes *adjustment set*. For these nodes (services), the

regression is modified with the goal to improve the accuracy of the Repository Model.

3. For each service from the adjustment set we examine the validation data and compare it with the monitoring data. We evaluate whether the response times in the validation data are higher than in the monitoring data. If this is the case, we scale up the values for the regression by a factor $f_{adjustment}$. Analogously, if the response times in the validation are higher than in the monitoring, we scale the values down. The factor $f_{adjustment}$ can be configured.

4. We carry out the normal procedure for calibrating the resource demands, as it was used in previous work [56, 55].

The idea behind this method is that we want to prevent that the accuracy of the simulations remains permanently low. One could apply much more complicated procedures here, this strategy is more to be seen as a proof of concept. Especially with machine learning methods, such as reinforcement learning, much more advanced methods could be used here in the future (see Section 7 about future work).

T_{Usage} The extraction of user behaviour is based on the preliminary work done in the context of iObserve [29]. However, because the structure of the monitoring data is very different in our approach, the concept had to be adapted. In iObserve, the monitoring data is bundled into sessions and these are clustered into groups with similar user behavior based on their behavior. As iObserve does not weave the information about the corresponding PCM directly into the monitoring, the run-time architecture correspondence model (RAC) is used to bridge the gap between elements in the source code and elements in the architecture model. This step is not necessary in our approach, as the required information is directly available through the customized monitoring. The adaptation of the procedure from iObserve to our monitoring data structures was already implemented in the context of CIPM [56]. In our approach, it was reused, only due to the changes in the monitoring data types there were minor adjustments.

Figure 4.23 provides a visual compressed overview of the procedure.

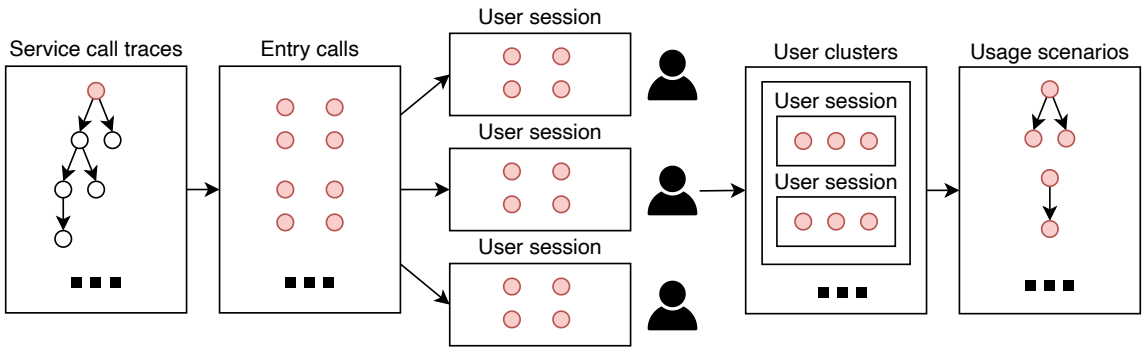


Figure 4.23.: Overview of the process that is used to extract the usage scenarios (based on iObserve [29])

4.8.6. Finalization

In the last step the final PCM model is built, then validated and finally the Instrumentation Model (IM) is updated. Figure 4.24 shows all transformations that are part of this step.

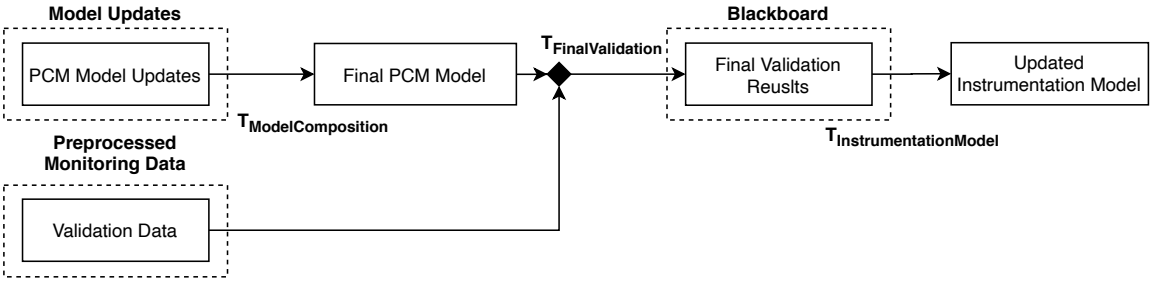


Figure 4.24.: Detailed view on the transformations that are triggered in the last step of the pipeline execution (Finalization)

$T_{ModelComposition}$ This transformation constructs the final model. The updates from the previous steps are merged and applied to the final PCM instance.

$T_{FinalValidation}$ This step is logically equivalent to the previous validations. The monitoring data is compared to the simulation results of the current model and validation results are calculated. Based on these validation results, the IM is updated in the subsequent transformation $T_{InstrumentationModel}$.

$T_{InstrumentationModel}$ This transformation is used to adjust the current IM at the end of the pipeline execution. The validation results of $T_{FinalValidation}$ are used and depending on that, the fine-grained monitoring for a service is either disabled or enabled. The corresponding validation results are evaluated for each service. Based on configurable criteria, the fine-granular monitoring is then activated or deactivated. It is possible to specify composite criteria. The sub-criteria can be combined with a logical “And” or a logical “Or”. In this way, all criteria that are relevant in practice can be realized. If the superior criterion is met, the service will be instrumented coarse-grained. Within the criteria, conditions can be specified using Java syntax. An example of such criteria looks as follows:

```
KSTest < 0.2 OR Wasserstein < 10
```

This criterion is met if the KS test metric is lower than 0.2 or the Wasserstein distance is lower than 10. If a service should be instrumented coarse-granular it means that all *AbstractInstrumentationPoints* are deactivated for the corresponding *ServiceInstrumentationPoint* in the instrumentation model.

The monitoring clients poll the IM at regular intervals and thus adjust the monitoring. The goal is that after a certain period of time the accuracy specifications for all services are fulfilled and therefore all of them are instrumented coarse-grained. This reduces the monitoring overhead to a minimum.

5. Evaluation

In this chapter the evaluation of the thesis is presented. We decided to use a case study based approach because it allows us to make profound statements about the applicability and practicability of our approach in a real world setting.

5.1. Overview

First, the objectives of the evaluation are derived and outlined in Section 5.2. Section 5.3 then presents the Goal Question Metric (GQM) plan for the evaluation. As the name suggests, it introduces the goals, the related scientific questions and metrics to answer these questions. The goals are imported from the previous section, where they were deduced. Section 5.4 presents the metrics that are used in the evaluation. A distinction is made between metrics for comparing models and metrics for comparing distributions. Section 5.5 introduces the evaluation procedure and defines requirements for the case studies to be used. Based on these requirements, the concretely selected case studies are named. Then, in Section 5.6, the selected case studies are described. Thereafter, Sections 5.7, 5.8 and 5.9 present the results for the respective goals. Finally, Section 5.10 summarizes the evaluation results and Section 5.11 discusses the threats to validity.

5.2. Evaluation Objectives

In this section we introduce the goals for the evaluation and discuss the reasoning behind them.

The primary goal is to ensure the accuracy of the derived architecture models (**G1**). This is also the most important quality feature of our approach. In order to use the approach in a meaningful way, it is necessary that the resulting architecture models represent the actual application as well as possible. To measure the accuracy we use example systems. We compare the derived models with reference models and use the monitoring data for benchmarking. We also simulate changes of the systems. For this purpose we are using the scenarios from Section 3.3. With the help of these scenarios we want to simulate the run-time adaptation of the software system under investigation. The support of software adaptation is an integral part of the approach and therefore it is important that the accuracy of the models is preserved, even after the execution of change scenarios. We will not examine the accuracy of the extracted usage models in detail, because the procedure was migrated from iObserve and has therefore already been evaluated [30].

The second objective of the evaluation addresses the performance of the approach (**G2**). We focus on the performance of the components that are used at run-time. These parts of

the approach have to be efficient enough to process the monitoring data in an acceptable time. If this is not the case, no consistent models can be generated because the processing is lagging behind. In this context, we also focus on the overhead of monitoring. It must be guaranteed that the monitoring does not significantly influence the performance of the application under investigation, otherwise the approach cannot be used in practice. We will not examine the performance of the components used at design time (e.g. extraction of the system model) regarding performance characteristics. The execution time is not crucial here, since the behavior of the application under investigation does not change in the short term.

The third and final goal is to investigate the scalability of the transformations within the pipeline (**G3**). The execution time of the transformation pipeline depends heavily on the amount and structure of the monitoring data. As a result, the performance of our approach highly depends on the scalability of the transformations. In G2, we only consider the behavior for example systems; here we examine the run-times of the transformations in more detail. We use best and worst case scenarios to evaluate the scalability. This ultimately allows us to make accurate statements about the behavior of our approach in general.

5.3. Goal Question Metric (GQM) Plan

The evaluation is based on the Goal Question Metric (GQM) approach [3]. First, we define goals, which are based on the previous section. Afterwards, we define questions concerning the goals and finally we introduce metrics to answer these questions. The metrics are discussed in more detail in Section 5.4, in the following list they are only mentioned.

G1 To ensure the applicability of our approach as a whole, it is necessary that the extracted and updated models are accurate. In this context, we consider two different types of accuracy. First, the difference between the actual behaviour of the application (monitoring) and the behaviour of the models within a simulation (analysis). Second, the comparison of the actual models with reference models, that represent the ground truth. The first goal is therefore that the extracted and updated models represent the application under consideration well.

- **Q1.1** How accurate are the system models that are extracted at design-time?
  - **M1.1.1** Number of the extracted elements in the resulting System Model (*AssemblyContexts*, *AssemblyConnectors*, ...), compared to the reference model.
  - **M1.1.2** Content-related conformance of the elements in the extracted system model, compared to the reference model. Conformity is determined by using the Jaccard coefficient (JC).
- **Q1.2** How many conflicts must be resolved by the developer when extracting the system models at design-time and how many conflicts can be resolved automatically based on a static code analysis?
  - **M1.2.1** Ratio of the number of conflicts in relation to the total number of elements in the resulting model (*without* preceding code analysis)

M1.2.2 Ratio of the number of conflicts in relation to the total number of elements in the resulting model (*with* preceding code analysis)

Q1.3 Are the Resource Environment Model, the allocation model and the system model adjusted correctly at run-time when applying software adaption scenarios?

M1.3.1 Content-related conformance of the elements in the updated system model, compared to the reference model. Conformity is determined by using the JC.

M1.3.2 Content-related conformance of the elements in the updated Resource Environment Model, compared to the reference model. Conformity is determined by using the JC.

M1.3.3 Content-related conformance of the elements in the updated allocation model, compared to the reference model. Conformity is determined by using the JC.

Q1.4 How accurate is the combination of all models that are adjusted at run-time? As ground truth, the monitoring data is used.

M1.4.1 Distances of traditional statistical measures between the response times of the services in analysis and monitoring. As traditional statistical measures the average, the standard deviation and the quartiles (Q1, Q2 and Q3) are used.

M1.4.2 Wasserstein distance between the response time distributions of analysis and monitoring.

M1.4.3 Kolmogorov–Smirnov test (KS test) of the response time distributions of analysis and monitoring.

G2 It is necessary that the transformation pipeline is able to quickly eliminate inconsistencies within the models. Therefore, the second objective of the evaluation is to confirm that the transformation pipeline as a whole performs well. It is also essential that the monitoring does not noticeably affect the response times of the observed application.

Q2.1 How long does it take to execute the transformation pipeline for a certain number of monitoring records?

M2.1.1 Execution time of the transformation pipeline as a whole.

M2.1.2 Execution times of the individual transformations within the pipeline.

Q2.2 To what extent does the instrumentation model help to reduce pipeline execution times in the long run (by supervising the fine grained monitoring)?

M2.2.1 The time that elapses until the models are accurate enough to disable the fine grained monitoring.

M2.2.2 The percentage by which the number of monitoring records is reduced (after the fine grained monitoring has been disabled).

M2.2.3 The percentage by which the execution time of the transformation pipeline has decreased due to the reduction of the monitoring data.

Q2.3 How significant is the overhead that is caused by the monitoring?

M2.3.1 The absolute time required to generate and process the monitoring records (per service execution).

M2.3.2 The percentage by which the execution times of the individual services increase.

M2.3.3 The difference between the quantified overhead caused by the fine-grained monitoring and that caused by the coarse-grained monitoring.

G3 The third goal is to examine the scalability of the individual transformations in general. In particular, worst-case scenarios are considered in order to ensure that the scalability of the approach is guaranteed.

Q3.1 How does the transformation which calibrates the Repository Model scale in worst case scenarios?

M3.1.1 Execution times of the calibration for an increasing number of monitoring records.

Q3.2 How do the transformations for updating the Resource Environment Model, the System model and the Allocation model scale in worst case scenarios?

M3.2.1 Execution time of the transformations for an increasing number of monitoring records.

Q3.3 How does the extraction of the usage scenarios react to an increasing number of monitoring records in worst case scenarios?

M3.3.1 Execution time of the Usage Model extraction for an increasing number of monitoring records.

5.4. Evaluation Metrics

In the evaluation we mainly used two types of metrics. These are introduced and explained in more detail within this section. On the one hand these are metrics that quantify the equality of PCM models (see Section 5.4.1) and on the other hand there are metrics that are used to compare two distributions (see Section 5.4.2).

5.4.1. Model Conformity

In general, there are two different approaches to examine the equality of models. The first approach directly analyzes the model elements and compares them with each other. The second approach is based on simulating both models and comparing the results. We decided to use the first approach, because it is possible that two Palladio Component Model (PCM) models (that are completely different) produce the same analysis results for a given user scenario. The reason for this is that a simulation of a PCM model depends on

the interaction of several model parts (e.g. Usage Model and Repository Model). If, for example, the user behavior changes and we adjust the Usage Model in both PCM instances, it is still possible that the analysis results of the models are completely different, even if the simulation results were similar in advance.

For our evaluation we need to be able to compare System Models, Resource Environment Models and Allocation Models. For all model types, the resulting metric is based on the Jaccard index [20] or also called Jaccard coefficient (JC). The JC is defined as follows:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In this context, A and B are sets. If the two sets are exactly identical, the JC is equal to 1 and if no single element matches it is 0. Normally, the JC is used to quantify the equality of sets. However, we can also apply this concept to models by considering the model as a set of model elements. In order to apply the JC it is necessary that we define equality conditions for all element types. If the resulting JC is equal to 1, then the two models under investigation are completely identical. In our implementation we perform a greedy search for the intersection of the model elements. In other words, we iterate over all elements in the first model and look for elements in the second model that are equal. As soon as we found the first match, both elements are added to the intersection and are not considered in the remaining process. Theoretically it is possible that such a greedy search algorithm underestimates the intersections. However, since we always strive for a value of 1 in the evaluation, i.e. that the models are completely identical, this fact is not important.

In the following paragraphs, the details for the comparison of the individual model types are explained.

System Model The idea is that we map all elements in the System Model to elements in the Repository Model. The referenced elements are considered as static and can therefore be used as anchor for the equality definition. For the elements in the Repository Model, the definition of equality is particularly easy, we just need to compare the IDs of the elements. For example, two assembly contexts are equal if the IDs of the underlying components are the same. This concept has been extended to all types of model elements within System Models.

Resource Environment Model and Allocation Model The Resource Environment Model and the Allocation Model are analyzed in a combined procedure. As a consequence, we calculate only one JC for both models together. Similar to the JC calculation for the System Model, we use other parts of the architectural model as anchors. We also assume here that the Repository Model is static.

Next, we define the equality on the resource containers within the Resource Environment Model. Two resource containers are considered equal if the same components are deployed on both of them. This means that the number of deployments on the respective containers are the same and the component types that have been deployed are also the same. In addition, the defined hardware resources must be identical, i.e. amongst other things the number of CPU cores and the processing rate must be equal. Of course, this must not only

apply to the CPU but also to all other hardware resources, because if this is not the case, both containers will behave differently in simulations. With the help of this definition, we can then also specify equality for links between containers: they are equal if the linked containers are equal.

If all containers and links in both models are the same according to these definitions, both Allocation Models and Resource Environment Models are identical.

5.4.2. Distribution Comparison

To compare distributions we mainly use three types of metrics: conventional statistical measures, non parametric tests (KS test) and distance functions (Wasserstein). In the following, we go into more detail about the actually used metrics and explain their informative value in general. These metrics are used to compare the distributions of the monitored response times (reality) with the simulated response times of the models (prediction).

Conventional statistical measures The classic statistics metrics are calculated for both distributions and afterwards the distance is calculated. The following metrics are used:

- average
- quartiles (Q1, Q2, Q3)
- standard deviation

The higher the distance between these metrics, the less similar the two distributions.

Kolmogorov–Smirnov test (KS test) The KS test is a non-parametric test that checks whether two distributions match [41]. This is done by calculating the maximum distance between the Cumulative Distribution Functions (CDFs). The minimum is 0 (if both distributions are perfectly identical) and the closer it is to 1, the more different are the distributions.

Normally, this non-parametric test is used to check whether two random variables originate from the same underlying distribution. Therefore, this metric is not ideal for our use case. In particular, higher values are not very meaningful because shifts of distributions are not considered. This phenomenon can be illustrated using Figure 5.1. Both are normal distributions, the second one is only shifted to the left on the x-axis. Here the KS test outputs a very high value, although the distributions are not significantly different. This fact was taken into account in the evaluation and is also a reason why we will not make any statements based on the KS test solely.

Wasserstein metric The Wasserstein metric is a distance measure for distributions [54]. In simple terms, it describes how much a distribution must be changed in order to be transformed into the other one. An advantage of this metric is that, unlike the KS test, it is not sensitive to shifts of the distributions. A drawback, however, is that the result is an absolute number that cannot be easily interpreted. In the evaluation we often consider several Wasserstein distances to have a baseline for comparison.

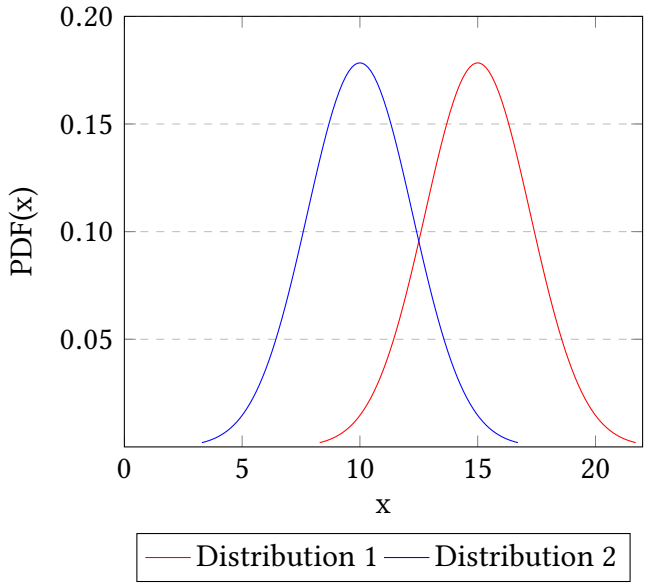


Figure 5.1.: Two separate normal distributions with different mean value (μ), visualized as probability density functions (PDFs) [24].

A notable disadvantage of the Wasserstein distance (and all other metrics that are considered in the evaluation) is that they all tend to be higher when the distributions have a high variance. Especially during the monitoring of applications, many influencing factors can cause the response times of services to be strongly spread temporarily. This fact was compensated in the evaluation by performing all simulations, calculations and measurements several times and calculating average values. Consequently, certain outliers are negligible. Another drawback is that two distributions with different numbers of data points cannot be compared without further ado. It is possible to assign penalty points for each additional data point, but this would not be reasonable in our use case, because the size of the distributions should not influence the distance. If no penalty points are used, the Wasserstein distance is no longer a valid distance metric [52]. For this reason we make sure that the compared distributions always have the same number of data points within the evaluation.

5.5. Evaluation Setup and Procedure

For the evaluation we decided to follow a case study based approach. However, due to the design and structure of the GQM plan, there are also some requirements for the case studies:

[R1] The source code of the application must be available. Otherwise, we are only able to perform the procedure for extracting the system model at design-time to a limited extent. A code analysis is then not possible and therefore **Q1.2** can not be answered.

[R2] The implementation of the instrumentation and the monitoring is based on the widely used Kieker Framework [32]. Therefore, it must be possible that the case study can be monitored with the help of Kieker.

[R3] It is possible to simulate the scenarios from Chapter 3 at run-time. Since the approach should also support software evolution and adaptation scenarios, it is necessary that we can simulate them by means of the case study.

R3 is not a mandatory requirement here if several case studies are used and at least one of them meets this requirement. In addition, it is also advantageous if there are already initial architecture models (PCM) and related work has already dealt with the same case studies. This makes it possible to compare the results with each other.

For these reasons, we decided to use the following case studies: **CoCoME** [65] and **TeaStore** [39]. R3 is only partially fulfilled for CoCoME, since not all change scenarios can be realized. Therefore we used CoCoME to get preliminary results and reports. Afterwards we use the TeaStore to provide detailed conclusions. There were already existing architecture models for CoCoME, for TeaStore most parts had to be modelled from scratch. In the following sections 5.6.1 and 5.6.2 these two case studies are introduced in more detail.

The evaluation procedure is based on the GQM Plan and can be divided into the experiments displayed in Table 5.1. The table gives a rough description of the experiments and also lists the research questions that were addressed. Furthermore, it is mentioned which case studies were used for the experiments.

The procedure used to run **Experiment 3 (E3)** is visualized in Figure 5.2. The foundation for this procedure is a number of defined change scenarios which can be executed. The *Scenario Generator* selects a certain number from these input scenarios (a scenario can also be selected several times). For each of the selected scenarios, a reference model is generated. It reflects how the model has changed compared to the previous one. Since the *Scenario Generator* knows the executed change, it also knows how the model must change. Another output is the *Change Orchestration* component, which applies the selected changes at run-time. The arising monitoring data is then used as input for the pipeline of our approach. Finally, the resulting models are compared with the reference models and possible deviations are detected.

For Experiment 2 (E2) and Experiment 3 (E3) we use the transformation pipeline, which must be configured before it can be used. There are the following five important configuration parameters:

- *Sliding window size*: The size of the sliding window which contains the monitoring data that is used as input for the transformation pipeline (see Section 4.7.2).
- *Sliding window trigger*: The time interval between the executions of the transformation pipeline.
- *Simulation configuration*: Since the simulation of the PCM models is an integral part of the pipeline it is necessary to configure them. In particular, the simulation time and the number of measurements are required.

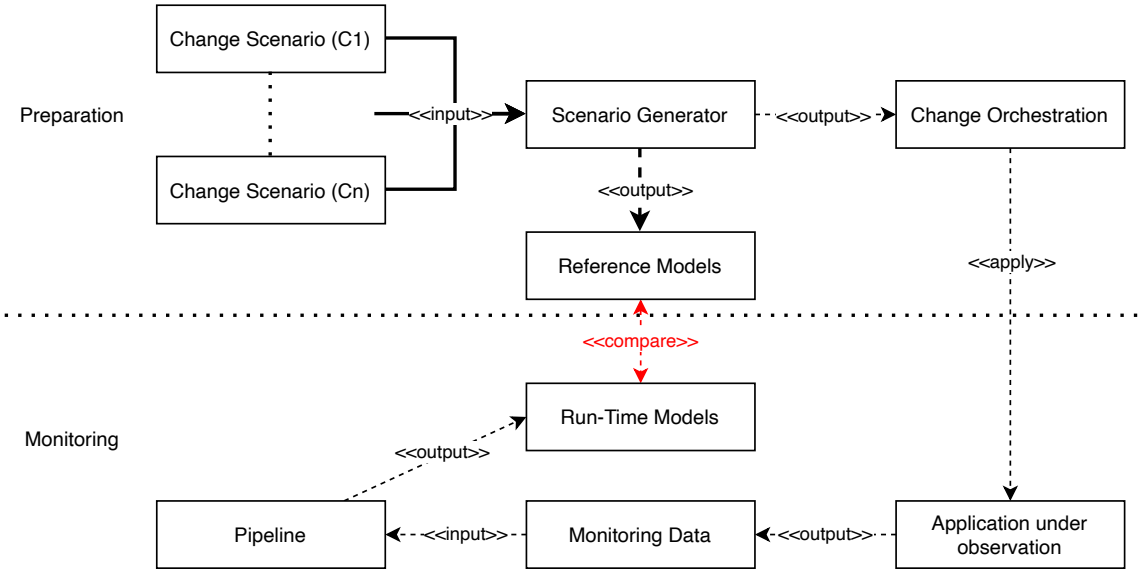


Figure 5.2.: Overview of the procedure that is used to simulate change scenarios of the application

- *Validation split*: Within the transformation pipeline, the monitoring data is divided into two sets: training and validation. The ratio between them can be configured using this parameter. A common value here is 80/20, which means 80% training data and 20% validation data [75].
- *Validation predicates*: The activation and deactivation of fine-grained monitoring is controlled by validation predicates. These are the rules (based on the metrics) that must be fulfilled in order to deactivate fine-grained monitoring for a service (see Section 4.8.6). In the evaluation these predicates were selected manually. For this purpose, several tests were performed and different combinations have been tried out to determine the most suitable settings.

Depending on these parameters, the behaviour and possibly also the results of the evaluation will change. For this reason, we will always provide the parameters that were used. For most of the experiments we tried out various parameter values, but we concentrate on those that turned out to be the best.

ID	Description	Research Questions	Casestudys used
E1	Execution of the System Model extraction at design-time, based on the source code of the respective application.	Q1.1, Q1.2	CoCoME, TeaStore
E2	Monitoring of the application under examination and execution of a load test. Meanwhile, metrics about the monitoring are collected and the monitoring data is used as input for the pipeline. The monitoring data is then compared with the simulation results of the derived models. Finally, metrics are calculated to quantify the deviation between the monitoring data and the simulation results. Since we are deriving newer models over time, the metrics can be plotted over time.	Q1.4, Q2.1, Q2.2, Q2.3	CoCoME, TeaStore
E3	Similar to E2, but change scenarios are simulated to emulate the software adaptation. Here, we use the scenarios described in Section 3.3. Initially, a sequence of changes is generated, which are then applied to the application at run-time. In this way, we want to ensure that our approach also derives accurate models if the observed system changes at run-time. The procedure is summarized visually in Figure 5.2.	Q1.3, Q1.4, Q2.1, Q2.2, Q2.3	TeaStore
E4	Synthetic generation of monitoring data and using it as input for the transformations of the pipeline. The monitoring data is generated in such a way that it causes worst-case execution times. The goal is to examine the scalability of the pipeline.	Q2.1, Q3.1, Q3.2, Q3.3	-

Table 5.1.: Description of the experiments that were conducted to answer the scientific questions of the evaluation

5.6. Evaluation Environments

The following Sections 5.6.1 and 5.6.2 introduce the two case studies that we used within the evaluation. Especially the structure is explained as well as how they are used for the evaluation.

5.6.1. CoCoME

CoCoME (Common Component Modeling Example) is a software system as it can be observed in supermarket chains [65]. It is possible to scan products at cash desks, the purchases can be paid by credit card or cash. In addition, administrative tasks are also covered, e.g. it is also possible to reorder and create products, but also to generate reports [65].

There are several implementations of the case study. The oldest one is the plain Java implementation[1]. Furthermore, there is a cloud-based version[2] and an implementation that was realized via REST interfaces[3]. Within the evaluation we used the cloud-based implementation. Figure 5.3 shows all components of CoCoME. The *Application.Store* component is highlighted in particular; it is the entry point and provides the main functions of the system to the outside.

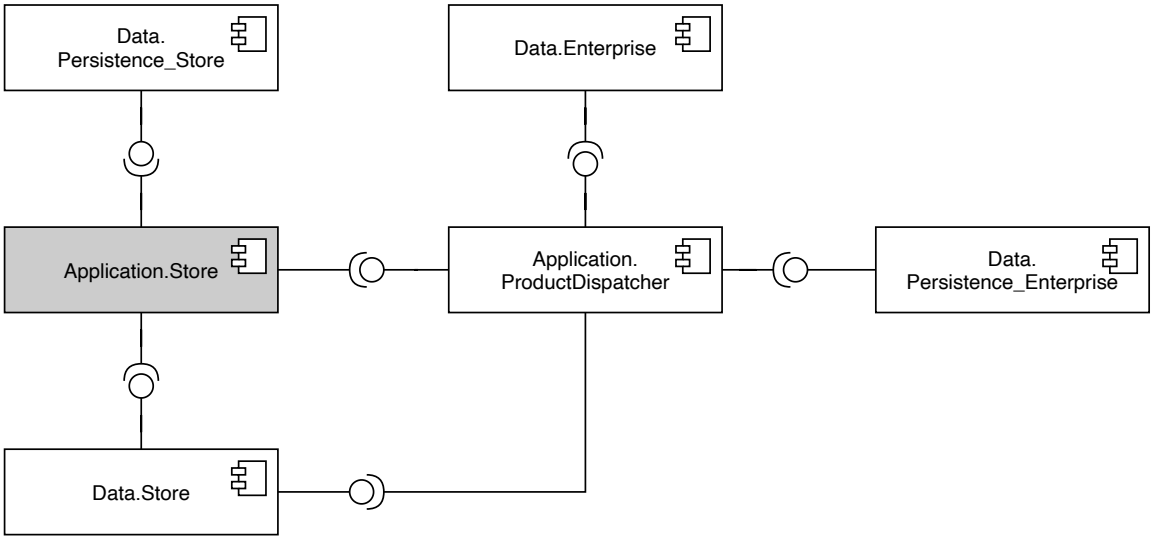


Figure 5.3.: Overview of all components of the cloud based implementation of CoCoME

The associated Palladio Component Model (PCM) had already been used in previous work and therefore it was not necessary to create it in the context of this thesis [29, 30, 56]. The mapping between source code elements and elements in the architectural model had to be established manually, since CoCoME was not developed with the support of Vitruvius. The subsequent instrumentation of the source code was performed automatically, as described in the conception of this thesis.

[1] https://github.com/cocome-community-case-study/cocome-plain-java
[2] https://github.com/cocome-community-case-study/cocome-cloud-jee-platform-migration
[3] https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest

In the evaluation we consider the "bookSale" service of CoCoME. This is the service with the most consecutive service calls and almost all components are involved. Moreover, this service has the highest complexity of all services included in the model. Consequently, if we have reflected this service well in the model, it provides well-founded information about the quality of our approach.

The environment for the run-time measurements and pipeline executions (Experiment **E2**) is designed to avoid mutual influences. In particular, this is important for a meaningful evaluation of the performance characteristics of our approach. For this purpose, the pipeline (core of our approach), the PCM simulations and the application under investigation (CoCoME) are executed on different computers. Additionally, we use Apache JMeter [26] to put load on the system. As a result, we bring the system into a state as it occurs under realistic conditions in the production environment. This significantly increases the meaningfulness of the evaluation. Figure 5.4 summarizes the experiment setup for CoCoME graphically.

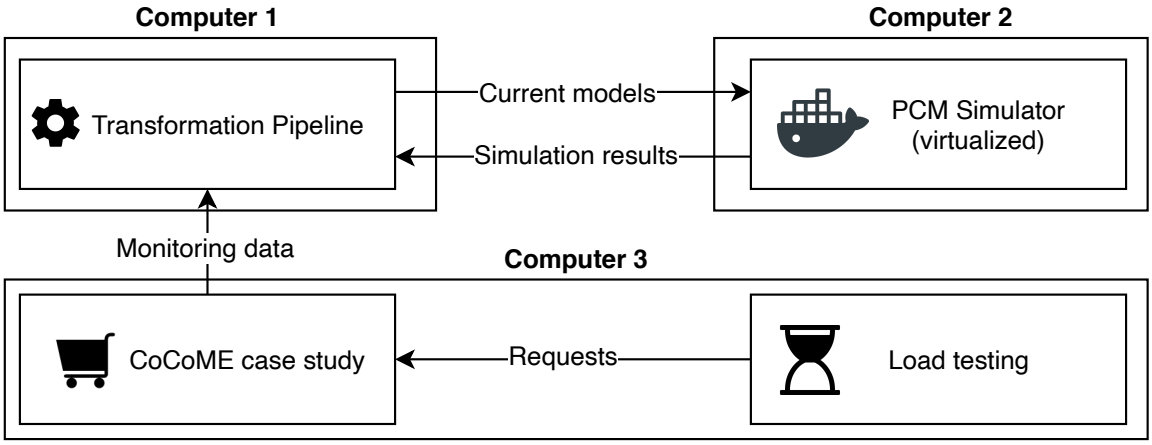


Figure 5.4.: Summary of the experiment setup in the context of the CoCoME case study

5.6.2. TeaStore

The TeaStore is a web-based application which, as the name already implies, implements a shop for all kinds of tea [39]. The application is designed to be suitable for the evaluation of approaches in the field of performance modeling [39]. The TeaStore is based on a distributed microservice architecture. The participating components can be replicated and de-replicated. A load balancer distributes the load to components of the same type. In addition, the TeaStore can be easily extended. These features make the TeaStore very suitable for our evaluation (especially for Experiment **E2** and Experiment **E3**).

Figure 5.5 shows all components of the TeaStore and visualizes their interaction. All components register themselves at the registry, which remembers all instances and makes them available for the individual components. This enables client-side load balancing, which was implemented using Ribbon[4]. Communication between the components is

[4] https://github.com/Netflix/ribbon

based on the widely used representational state transfer (REST) standard. The WebUI uses all other components and assumes that they are operational. There are four different Recommender components (implementations) with identical functionality, which are used to suggest related products to the users. All recommenders offer two different services, “train” and “recommend”. The "train" service is used to learn the recommender with orders from the past. The “recommend” service, on the other hand, is used to suggest products to the user that are related to his current shopping cart.

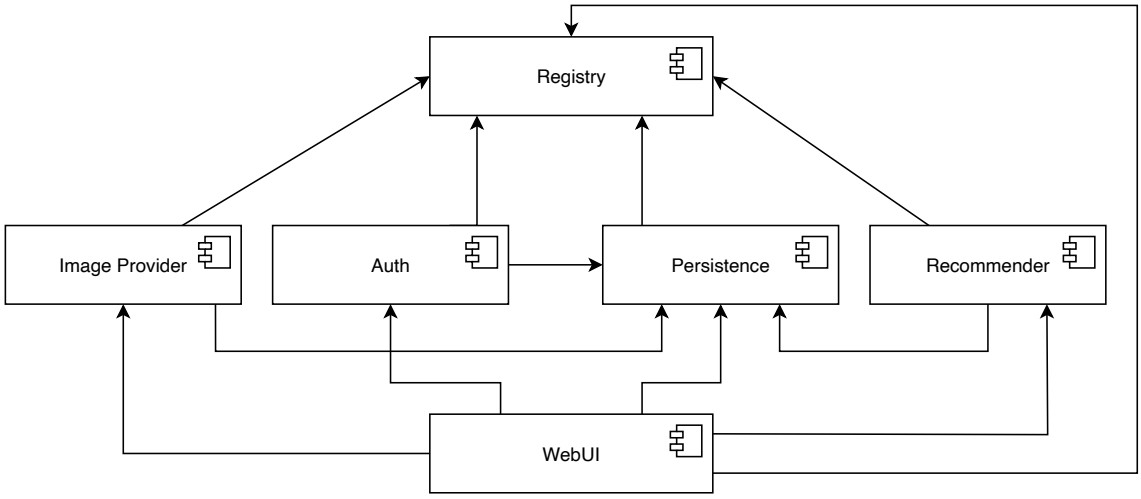


Figure 5.5.: Overview of all TeaStore components and visualization of the interaction between them[39]

So far, no comprehensive PCM models for the TeaStore existed. In previous work, parts of it had already been modelled and examined, but the entire system was not covered. Therefore, the case study was modelled extensively in the context of this thesis. In the evaluation, we consider the service “confirmOrder” which is offered by the WebUI. This service is executed when a user places an order. We examine this service in the evaluation because it has the highest complexity of all included services. Furthermore, almost all components are involved in the execution (except the Image Provider and the Recommender). In order to include the Recommender as well, the TeaStore was slightly modified. After the order has been processed within the “confirmOrder” service, the Recommender is re-trained. As a result, all components, except the Image Provider, are involved in the “confirmOrder” service. Figure 5.6 shows a simplified version of the corresponding Service Effect Specification (SEFF). For reasons of clarity, the modeling of the overheads caused by the REST calls and the detailed representation of the load balancing were omitted. Nevertheless, the figure gives a good indication about the complexity of the service and the extension that has been added (training the recommender after placing the order).

Due to the introduced extension, the run-time of the “train” service directly influences the response time of the “confirmOrder” service. The different implementations of the Recommender have different performance characteristics, but in the majority of them, the response time increases proportionally to the orders stored in the database. This leads to

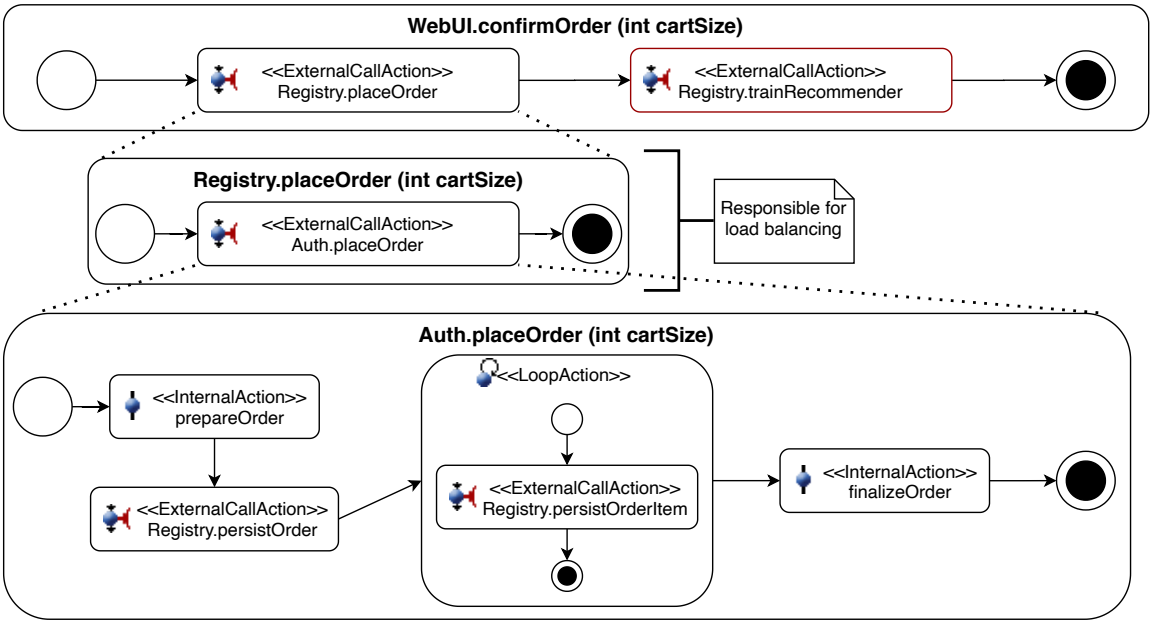


Figure 5.6.: Simplified presentation of the Service Effect Specification (SEFF) of the extended "confirmOrder" service and some selected subsequent service calls (the extensions are highlighted in red)

the fact that the response time of the "train" service increases with the number of persisted orders. Logically, the execution time of the "confirmOrder" service also increases. This adds an additional level of difficulty to the evaluation, as the approach must also recognize the dependency between the number of orders and the increasing response times.

All components except the registry are replicable and can therefore be used to simulate change scenarios concering the deployment. Because we want to run Experiment **E3** with the help of the TeaStore, it is necessary that we can simulate the scenarios from Section 3.3. In order to be able to realize the changes of the run-time environment easily, we use Docker[5] for virtualization. This makes it possible to create and remove containers with certain components in a simple way. The following list shows how the change scenarios from Section 3.3 are executed by means of the TeaStore case study:

[U1] *Allocation/De-Allocation*: Start-up or shut-down of a container via Docker. When shutting down it must be ensured that no component is deployed on the selected container.

[U2] *Migration*: First the component is removed on the source container, then it is deployed on an already existing container. Again, this is achieved by using Docker.

[U3] *Replication/De-Replication*: Either we select a previously replicated component and dereplicate it, i.e. we delete the deployment using Docker. Otherwise, we select one of the five replicable components (Auth, Persistence, Image Provider or Recommender) and deploy it on an existing container. We exclude the WebUI here, otherwise

[5] `https://www.docker.com/`

the load testing would also have to be adjusted so that the load is distributed over all instances. In this context it was ensured that the WebUI does not become a bottleneck, which itself is very unlikely, since the requests are passed directly to the Registry.

[U4] *Changes of the system composition*: Change of the used Recommender implementation at run-time. In addition, replications and de-replications implicitly trigger changes to the system composition, since components are connected to/ or disconnected from the load balancer.

[U5] *Usage behavior*; We use different load test scenarios, which differ for example in how many products a user buys on average. The more products are purchased, the higher the response time of the "confirmOrder" service. Thus a change of the load test scenario has a direct impact on the performance characteristics of the system.

[U6] *Workload*: The number of users can be easily modified in the load test. We use three different numbers of users: 20 (default), 30 and 40. The higher the number of users, the faster the database fills up and thus the run-time of the services under observation increases.

An illustration of the final setup for the experiments based on the TeaStore case study is shown in Figure 5.7.

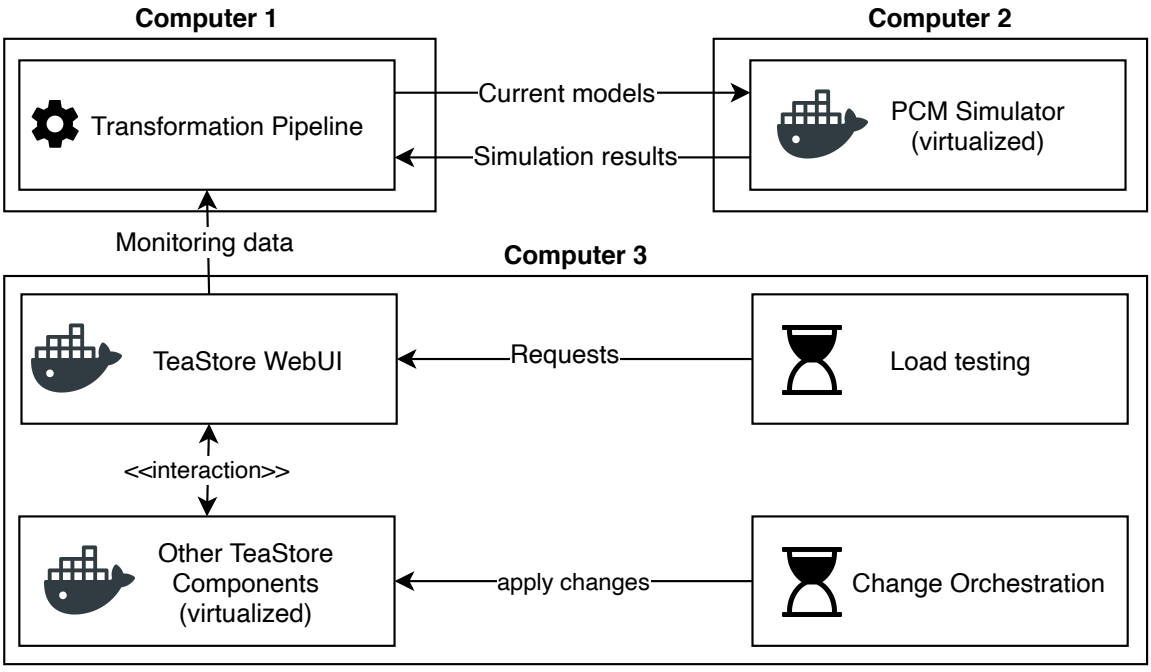


Figure 5.7.: Summary of the setup of the TeaStore case study for carrying out the evaluation experiments

First, we generate a sequence of change scenarios with the procedure shown in Figure 5.2. As input we use the specified change scenarios for the TeaStore. These changes are

then applied to the system at fixed time intervals; this task is performed by the *Change Orchestration*. The actual components of the TeaStore are all virtualized using Docker, allowing us to easily apply changes to the run-time environment. The remaining setup is similar to the setup for CoCoME (see Figure 5.4).

5.7. Model Accuracy

In order to evaluate the quality of the models (G1) we use the experiments introduced in Section 5.5. In the following sections the performed experiments and their results are explained in more detail.

5.7.1. Experiment 1 (E1)

Experiment 1 concerns the accuracy of the System Model extraction at design-time. The corresponding concept was introduced in detail in Section 4.6.2. For this purpose, we need the source code and the mapping between source code elements and elements in the architecture model (PCM) as input. The resulting models are compared with reference models and we use the Jaccard coefficient (JC) to quantify the equality (see Section 5.4.1). As case studies we use both CoCoME and TeaStore.

The designed procedure for the derivation of the system model works, as mentioned before, semi-automatically. Conflicts that cannot be handled automatically must be resolved manually by a developer. In order to keep the number of conflicts as low as possible, the first step is to synthesize a Service-Call-Graph (SCG) from the source code using a code analysis. This SCG is used to automatically resolve as many conflicts as possible. To estimate the effect of the code analysis on the number of conflicts, we ran the experiment once with and once without a preceding code analysis. Figure 5.8 summarizes the procedure and the evaluation of Experiment 1 (E1) graphically.

The following paragraphs examine the results of the experiment, broken down according to the respective case studies.

CoCoME Table 5.2 lists the number of elements in each model and indicates whether they match. Only the most important element types are included in the table; for example, the roles provided by the system were ignored. These are specified by the user at the beginning of the procedure and should therefore always conform to the reference model. In the subsequent calculation of the JC, however, all elements were taken into account.

Model	Assembly Contexts	Assembly Connectors	Delegations
Derived Model	6	6	2
Reference Model	6	6	2
Matching	✓	✓	✓

Table 5.2.: Number of model elements grouped by element type for the extracted System Model and the reference model; concerning the CoCoME casestudy

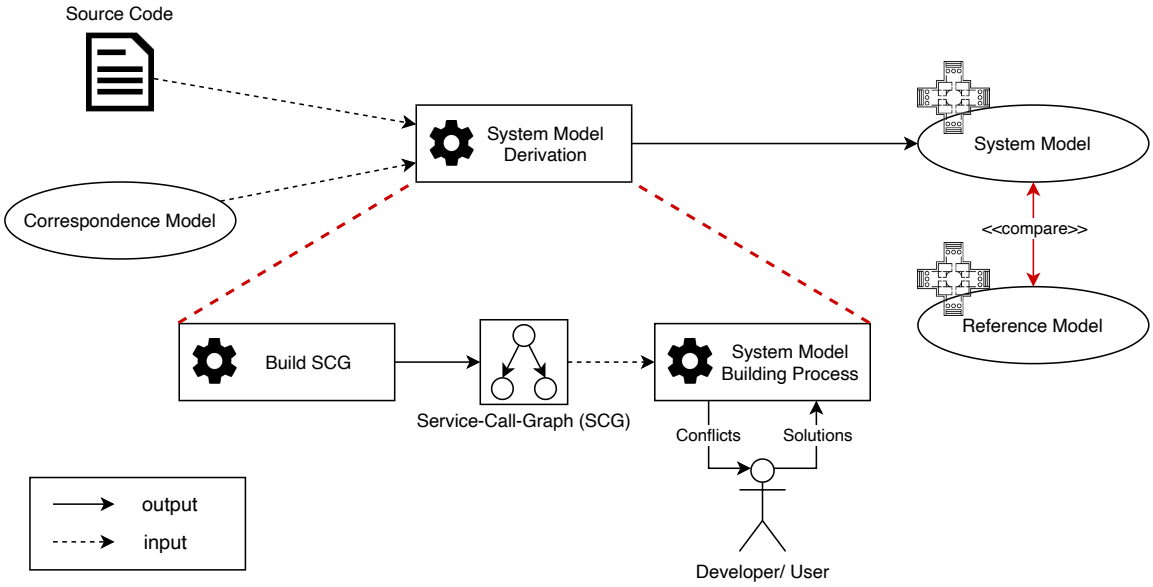


Figure 5.8.: Overview of the procedure and the evaluation of Experiment 1 (E1)

$$JC_{CoCoME}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{16}{16} = \mathbf{1.0}$$

A Set of all elements of the extracted model
B Set of all elements of the reference model

The method for the calculation of the JC was introduced and described in Section 5.4.1. In this case it is calculated as follows: The result, a Jaccard index of 1.0, indicates that both models are **completely identical**. Consequently, we can also conclude that both models behave equally in a simulation.

The extraction without prior code analysis resulted in 5 conflicts that had to be resolved manually. With a previous code analysis the number of conflicts could be reduced to 2. This corresponds to a reduction of 60% of the number of conflicts. Putting the conflicts in relation to the total number of elements in the final model gives a ratio of $\frac{5}{16} = \mathbf{0.3125}$ for the case without code analysis and $\frac{2}{16} = \mathbf{0.125}$ for the case with code analysis. It becomes clear that, depending on the application, the code analysis can help to significantly reduce the number of conflicts. As a result, the manual effort can also be reduced to a minimum. Therefore, the higher the quality of the code analysis, the more conflicts can be handled automatically. An excerpt of the SCG that was extracted by means of the code analysis is visualized in Figure 5.9. With the help of this SCG it was possible to resolve three conflicts automatically.

TeaStore The procedure for the TeaStore is the same as that used for the CoCoME case study. Table 5.3 shows the counts of the most important element types in the extracted model and in the reference model.

Also, as before with CoCoME, we calculated the JC in the same way: We can conclude

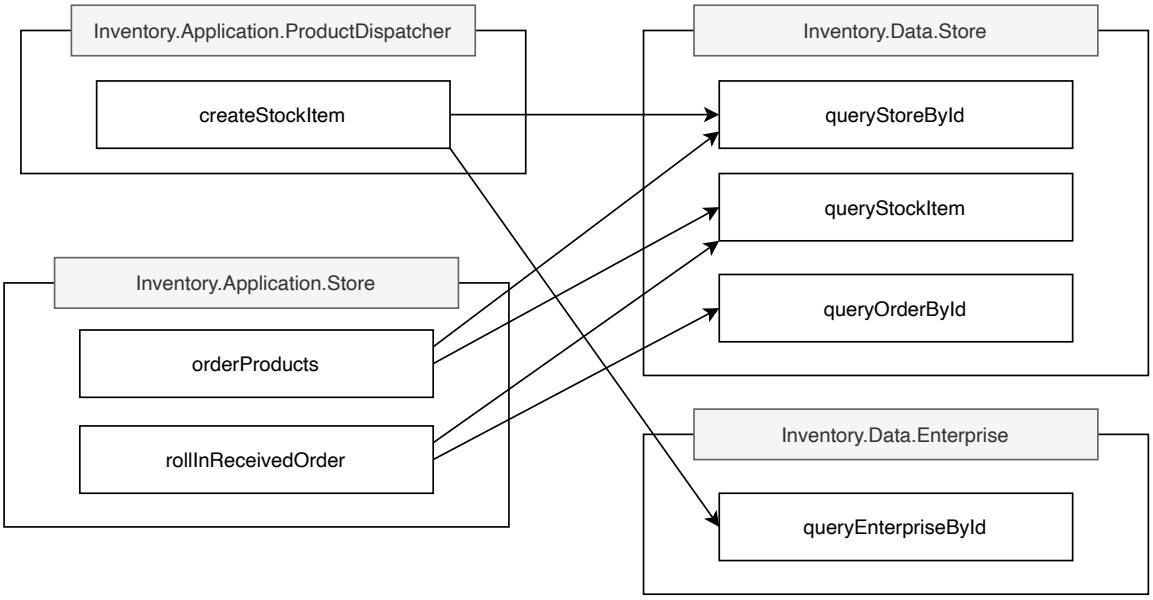


Figure 5.9.: Part of the Service-Call-Graph (SCG) that was obtained from the source code of CoCoME by applying a code analysis

$$JC_{TeaStore}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{18}{18} = \mathbf{1.0}$$

A Set of all elements of the extracted model
B Set of all elements of the reference model

that both models are identical based on the value 1.0 for the Jaccard coefficient.

There were **five conflicts** during the extraction of the system model for TeaStore. It did not matter whether a code analysis was performed before or not. In both cases five conflicts occurred. In other words, the code analysis and the resulting SCG could not contribute to the automatic resolution of some conflicts. The ratio between the number of conflicts and the number of elements in the extracted model corresponds to $\frac{5}{18} = 0.2\overline{7}$.

The reason that the code analysis has no effect is the architecture of the TeaStore. The components of the case study communicate exclusively via REST interfaces. Therefore, the code only contains the HTTP addresses and no direct method calls that can be traced. The mapping between the HTTP addresses and the REST interfaces cannot be resolved solely by a code analysis. Consequently, developer knowledge is required to build a meaningful SCG.

Summary In conclusion, the accuracy of the extracted model was optimal in both considered case studies. Furthermore, it became clear that a preceding code analysis can help to minimize the number of conflicts. However, it also turned out that there are cases in which the code analysis does not provide any benefit (TeaStore). For both case studies the number of conflicts was significantly lower than the number of elements in the final

Model	Assembly Contexts	Assembly Connectors	Delegations
Derived Model	7	7	2
Reference Model	7	7	2
Matching	✓	✓	✓

Table 5.3.: Number of model elements grouped by element type for the extracted System Model and the reference model; concerning the TeaStore casestudy

model. This means that the manual effort for modeling has been reduced significantly by the system extraction procedure. In addition, the conflicts can be prepared in such a way that the developer/ user no longer needs any knowledge of architecture modeling at all. For example, it would be possible to provide a list of possible components directly to the developer and ask which one is used in the present case. Then the developer is not responsible for the concrete modeling by means of a PCM, he only has to answer questions about the system composition.

5.7.2. Experiment 2 (E2)

In the second evaluation experiment we put the case studies under load and monitor them. On the one hand, the monitoring data is used to derive updates in the PCM. On the other hand it is used within the evaluation to estimate the accuracy of the simulation results. If the simulation results are accurate, it can be concluded that the derived models reflect the actual system well. The simulations of the models were carried out after the experiment itself, i.e. not the simulation results from the pipeline were used. The simulations were executed in such a way that the distributions of the response times contain exactly the same number of data points as in the monitoring. This must be ensured, otherwise there might be undesired effects when calculating the Wasserstein distance (see Section 5.4.2). In this experiment, we do not consider change scenarios, which means that the experiment environment does not change at run-time.

The following paragraphs go into more detail about the procedure and the results of the experiment, separately for the two case studies.

CoCoME As already mentioned in the introduction to CoCoME, we focus on the "bookSale" service here. The behavior of this service does not change over time, so we proceed with the experiment as follows:

1. Initial run which uses the transformation pipeline and the monitoring data as input to derive up-to-date PCM models.

2. Independent additional run during which CoCoME is only monitored and no models are derived.

3. Comparison of the monitoring data from the second run with the simulation results of the models from the first run.

Through this procedure we ensure that the transformation pipeline has not yet seen the data used for evaluation. Otherwise it would be possible for the transformation pipeline to build models that simply reflect the input. If one would then use the same data again for the evaluation, one attests the models a high accuracy, which is not intended.

The configuration of the parameters for the transformation pipeline is summarized in Table 5.4. The second part of the validation predicate is intended to ensure that services with very low execution times, which are rarely measured, are only instrumented with coarse granularity. It does not make sense to instrument these services with a fine granularity although the deviations are very small and therefore do not influence the actual services under consideration. Each run of the experiment was carried out for 3 hours. To obtain meaningful results, the experiments were performed several times (in this case 25 times). Afterwards, the data of the individual experiments were combined to be able to make final statements.

Configuration Paramter	**Value**
Sliding window size	15 minutes
Sliding window trigger	5 minutes
Simulation engine	SimuCom[9]
Maximum simulation time	300000 time units
Simulation measurements	20000
Validation split	80/20 (80% training, 20% validation)
Validation predicates	(KS test <= 0.2 **AND** Wasserstein distance < 20) **OR** (Average distance <= 2 **AND** Wasserstein distance < 5)

Table 5.4.: Configuration parameters for the transformation pipeline in the context of the CoCoME case study (see Section 5.5)

Figure 5.10 and Figure 5.11 both show examples of two randomly selected distributions from the experiment. Hereby, the response times of the "bookSale" service were considered. Figure 5.10 shows the density plot for a distribution of the response times in the monitoring and Figure 5.11 shows the density plot for a distribution of the response times in the simulation.

It is easy to see that both distributions look almost identical, which is a first indication for a good quality of the simulated model. Table 5.5 shows the corresponding quartiles and the average of the two distributions.

Table 5.5.: Quartiles for the sample distributions of the monitoring and the simulation

Distribution	Min	Q1	Q2	Q3	Max	Avg
Monitoring	29ms	240ms	322ms	453ms	1521ms	364ms
Simulation	21ms	229ms	339ms	427ms	1529ms	363ms

Again, it can be seen that both distributions are very close to each other. This impression can be confirmed by looking at the Cumulative Distributive Functions (CDFs) shown in

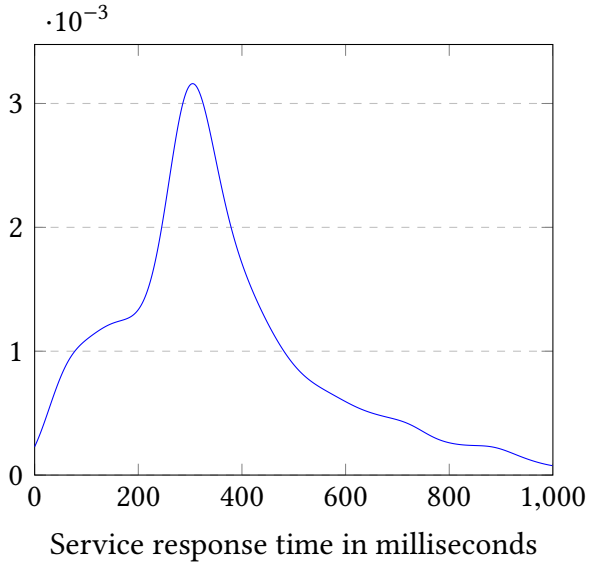


Figure 5.10.: Sample density plot for a distribution of the response times of the "bookSale" service in monitoring

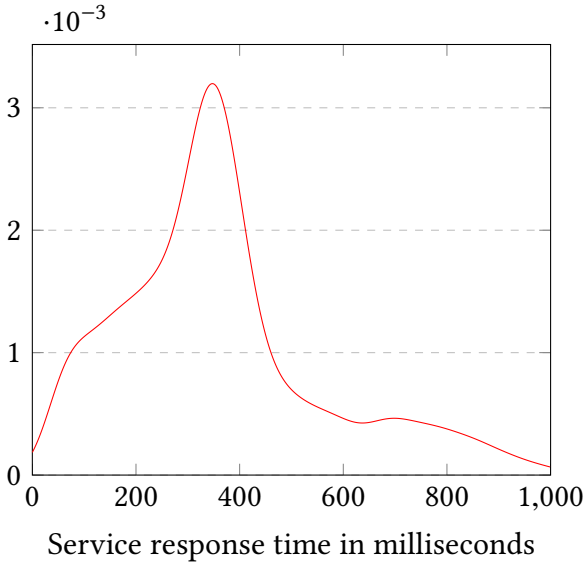


Figure 5.11.: Sample density plot for a distribution of the response times of the "bookSale" service in simulation

Figure 5.12. From this figure it is possible to retrieve the KS test metric, it results to the maximum distance between the two curves. In this case, the KS test is **0.075947** and the Wasserstein distance amounts to **12.384184**.

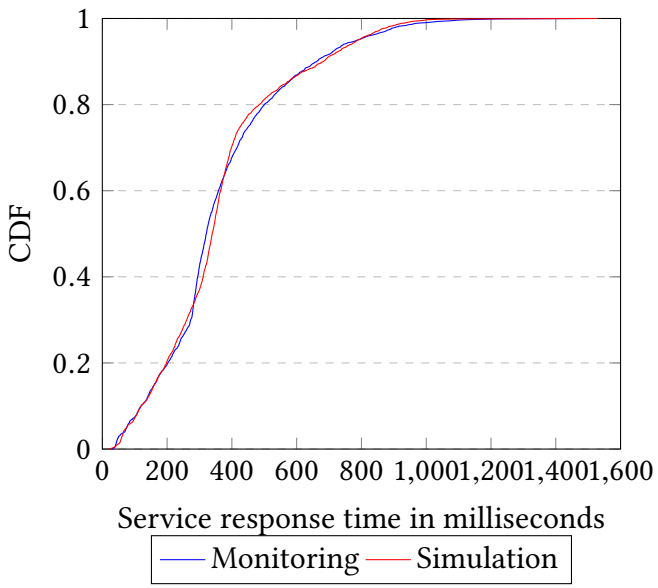


Figure 5.12.: Cumulative Distribution Functions (CDFs) for the "bookSale" service

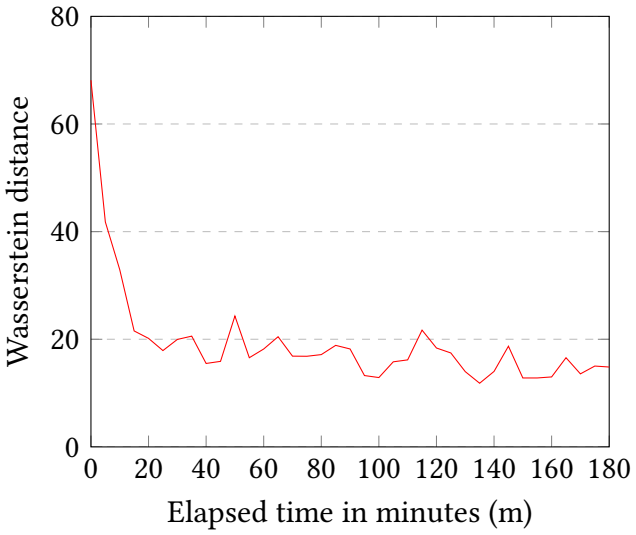
In this experiment we perform these comparisons for all derived models and aggregate the results. The results are plotted over time and this allows us to make statements about whether the models become more accurate over time or not. In order to eliminate outliers, the experiment was carried out 25 times, as already mentioned. When presenting the metrics over time, the medians of all runs were calculated. The used metrics were introduced in Section 5.4.2 and their informative value was explained in more detail.

Figure 5.13 (a) shows the Wasserstein distance between the simulations of the derived models and the monitoring data for the response times of the "bookSale" service. We always look at the PCM models at a certain point in time and compare the simulation results with the monitoring data from the additional, independent monitoring run.

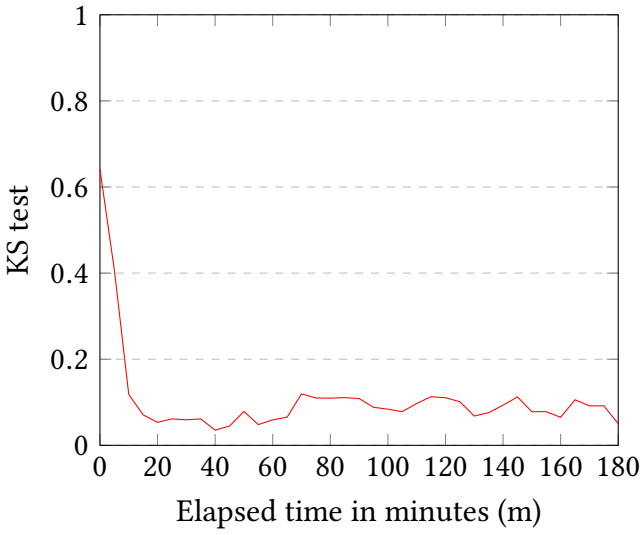
It can be seen that the Wasserstein distance decreases very rapidly at the beginning and then settles below a value of 20 with minor fluctuations. This means that the accuracy of the simulations increases over time and consequently the accuracy of the derived PCM models increases too. This observation conforms to the charts on the KS test (Figure 5.13 (b)) and on the distances between the conventional statistics metrics (Figure 5.13 (c)). In both graphs it can be seen that the metrics decrease over time and then stabilize at a low level. The fluctuations in the graphs are caused by the fact that the simulations are stochastic processes. In other words, it is very unlikely that two simulations produce identical results, even if the used models are equal. Although this was compensated by executing simulations and calculating the metrics several times, the fluctuations could still not be completely eliminated. This means that the accuracy of the models increases over time and then remains at a constant, solid level. The improvement in accuracy over time can be explained by two factors. First, the pipeline receives more and more data over time and thus information about the performance characteristics of the application. Second, the repository transformation learns over time from the monitoring data (see Section 4.8.5).

TeaStore The procedure for the TeaStore case study is similar to the procedure with CoCoME. We examine the "confirmOrder" service and focus on its response time. For the TeaStore, each run of the experiment was carried out for 2 hours. An important difference compared to CoCoME is that the response time of the considered service increases over time (see Section 5.6.2). As a result, we cannot use the same procedure as with CoCoME. Actually, it does not make sense to compare the complete monitoring data from the second run with the simulations of the models from the first run. Otherwise, particularly the models at the beginning would perform very poorly, just because they have not yet seen the increasing response times. Therefore we slightly adjust the procedure. For the evaluation of the models, we only use the monitoring data that has already been gathered at this point in time. As with CoCoME, we run the whole experiment 25 times in order to be able to make well founded statements and to eliminate outliers.

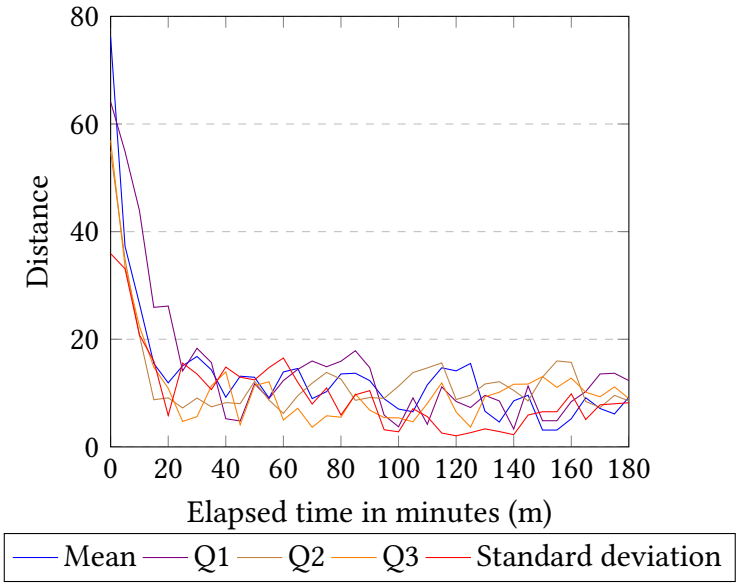
The configuration parameters for the transformation pipeline are summarized in Table 5.6. Here, we used a smaller sliding window for the monitoring data and the pipeline is also executed more frequently (compared to the experiment with CoCoME). In this way, we want to take care of the fact that the behavior of the service is constantly changing. The validation predicates are similar to the ones used for CoCoME, because they have proven to be very solid.



(a) Wasserstein distance over time



(b) KS test over time



(c) Distance between conventional statistical measures over time

Figure 5.13.: Overview of the metrics over time for the CoCoME case study (comparing the distributions which result from the analysis and the monitoring)

Configuration Paramter	Value
Sliding window size	6 minutes
Sliding window trigger	3 minutes
Simulation engine	SimuCom[9]
Maximum simulation time	300000 time units
Simulation measurements	20000
Validation split	80/20 (80% training, 20% validation)
Validation predicates	(KS test <= 0.2 **AND** Wasserstein distance < 40) **OR** (Average distance <= 2 **AND** Wasserstein distance < 5)

Table 5.6.: Configuration parameters for the transformation pipeline in the context of the TeaStore case study (see Section 5.5)

Figure 5.14 visualizes the increasing execution time of the considered service. This increase is implicitly caused by the growing number of orders that are stored in the database.

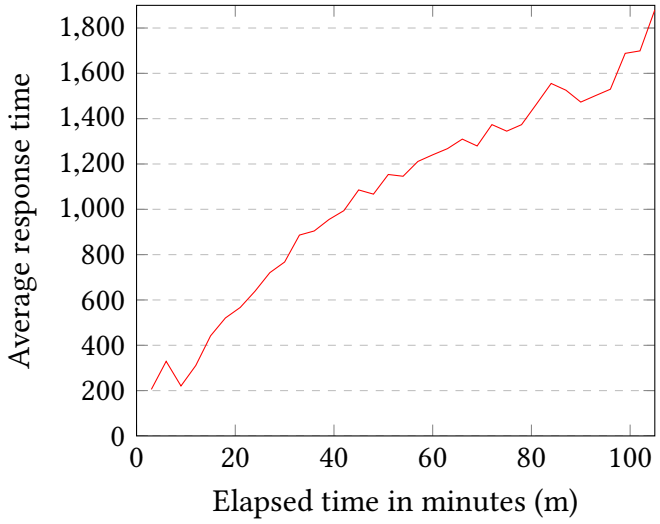
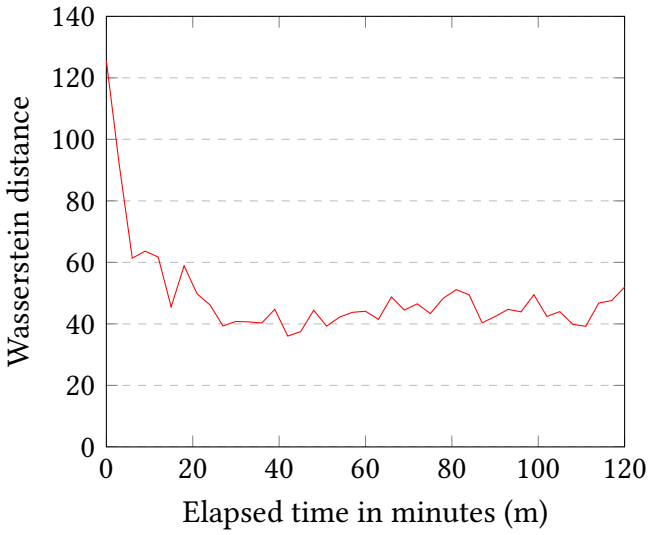


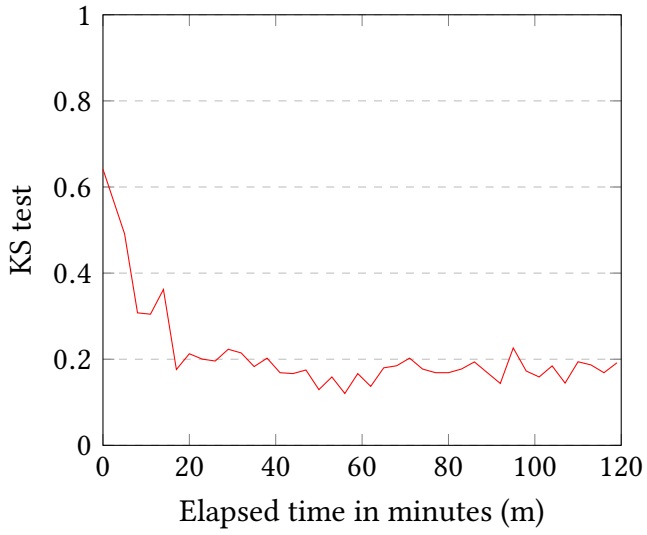
Figure 5.14.: Average response time of TeaStore's "confirmOrder" service over time (averaged over ten experiment executions)

To evaluate the accuracy of the derived models, we use the same approach as for CoCoME. In order to compare the simulation results with the monitoring data, we use the metrics introduced in Section 5.4.2. Figure 5.15 summarizes the Wasserstein distance throughout the execution of the experiment, the KS test and the distances between the conventional statistical metrics of both distributions (monitoring and analysis).

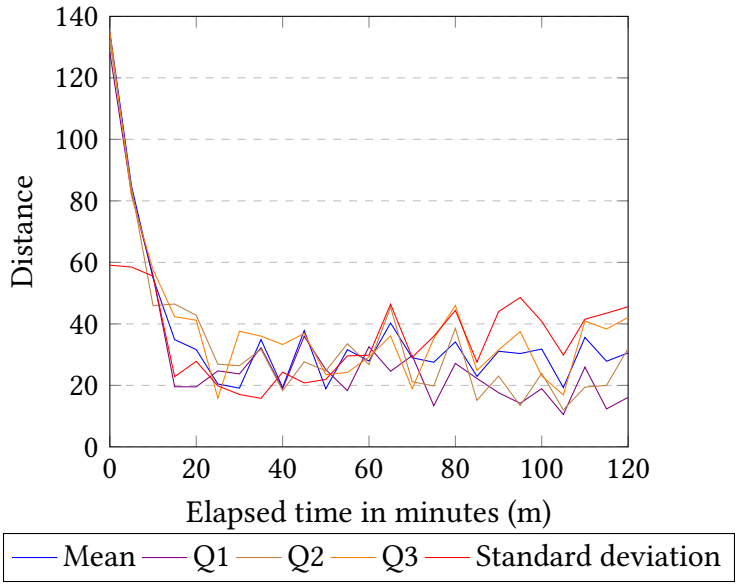
The graphs about the Wasserstein distance and the KS test depict the same trend, all metrics decrease over time. Consequently, the accuracy of the derived models increases over time. These observations are consistent with the results of the CoCoME case study. However, in the case of the TeaStore, the results are more meaningful, as the increasing



(a) Wasserstein distance over time



(b) KS test over time



(c) Distance between conventional statistical measures over time

Figure 5.15.: Overview of the accuracy metrics over time for the TeaStore case study (Experiment 2)

response time of the considered service adds an additional degree of difficulty. The classic statistical measures should be treated with caution, as a trend towards increasing gaps between monitoring and analysis can be observed. However, these are due to the fact that the average response time increases (see Figure 5.14), hence the absolute distances tend to become larger. In relation, this is not the case and therefore does not signal a loss of accuracy. Therefore, the meaningfulness of this graph is very limited.

During this experiment no change scenarios were applied, this is covered by Experiment E3 (see Section 5.7.3).

Summary Summarizing, it can be stated that for both case studies (CoCoME and TeaStore), the quality (accuracy) of the derived PCM models increases over time and then remains stable at a good level. Furthermore, the changing behaviour of the TeaStore case study's "confirmOrder" service was taken into account and the models were adapted properly.

5.7.3. Experiment 3 (E3)

In the third and final experiment on the accuracy of the derived models, only the TeaStore case study was considered. In contrast to the second experiment, we now also include the change scenarios introduced in Section 5.6.2. The procedure for generating and orchestrating the change scenarios is summarized graphically in Figure 5.2. The setup and configuration of the case study is exactly the same as in Experiment 2. The values of the pipeline configuration parameters have already been summarized in Table 5.6.

One run of the experiment lasts for 120 minutes and 40 randomly generated change scenarios are applied. Every 3 minutes the next change scenario is executed. This corresponds to the interval at which the pipeline is executed, so after each time the pipeline has been executed, a change is triggered. The changes to the workload and user behavior (*U5* and *U6*) are rolled back when the next change scenario is executed. The goal is to simulate peak loads. Like *E2*, the experiment was executed several times. In other words, we randomly selected 40 change scenarios multiple times. For each set of change scenarios, the experiment was performed several times, in order to eliminate outliers and obtain meaningful results. In the following, we will discuss individual results in detail, but also show aggregated data. Table A.1 in the appendix shows an exemplary list of change scenarios that are executed during an experiment run.

First, we investigated the accuracy of the models by comparing them with the generated reference models and calculating the Jaccard coefficient (JC). Table 5.7 shows the lowest observed JC for the different change types and the considered model types. As already mentioned, the Usage Model was not examined here, as it has already been intensively assessed in the context of iObserve [30, 29]. The experiment was executed 50 times and each run applied 40 change scenarios. Because the minimal Jaccard coefficient is equal to one in all cases, it indicates that there was not a single situation where the derived models differed from the reference models. Thus, research question **Q1.3** can be answered as well: in the considered experiment no deviations and consequently no errors in model adaption could be identified over a period of 50 runs. Thus it can be concluded that the model transformations provided accurate results and reflected the software adaptions at run-time.

Change Type	Minimum Jaccard Index	
	System	Allocation / Resource Environment
Allocation	1.0	1.0
De-Allocation	1.0	1.0
Replication	1.0	1.0
De-Replication	1.0	1.0
Migration	1.0	1.0
System Composition	1.0	1.0
Usage Behavior	1.0	1.0
Workload	1.0	1.0

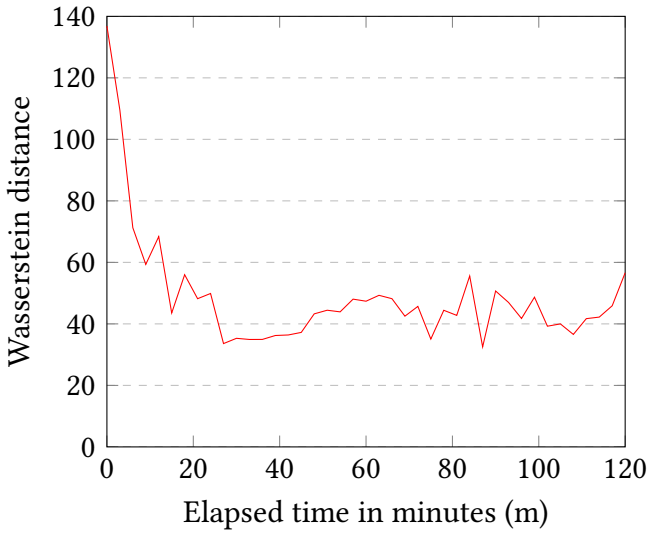
Table 5.7.: Overview of the minimum Jaccard coefficients for Experiment 3, representing the accuracy of the different models when executing change scenarios at runtime

In the second step, as in Experiment 2 (E2), the accuracy of the combination of all models was examined. The monitoring data is used as ground truth and compared to the simulation results. The applied metrics are also the same as in Experiment E2: the Wasserstein distance, the KS test and the classical statistical measures. As already mentioned, the experiment was executed 50 times, i.e. we simulated 50 different lists of change scenarios by means of the TeaStore. For each list of change scenarios multiple iterations were performed to eliminate outliers and measurement errors. We first calculated the median for each list of change scenarios and then the median of the metrics over time for all experiment runs. Hence we obtain results comparable to those of Experiment 2. Figure 5.16 summarizes the metrics over time of the experiment runs.

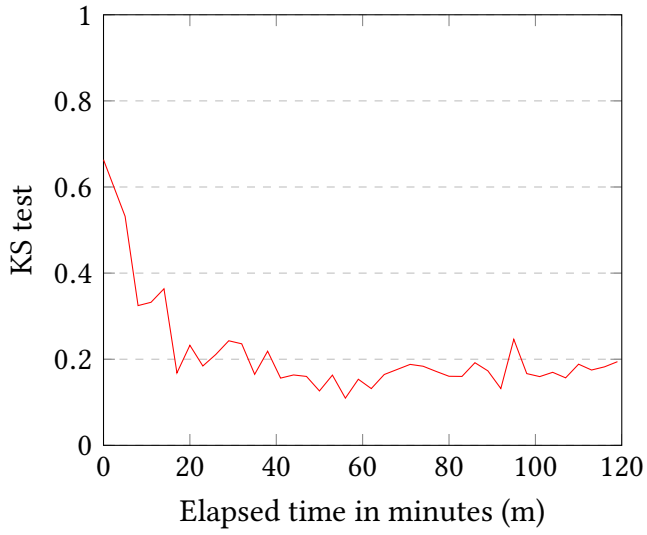
It can be seen that the results are almost identical to those of Experiment 2. Consequently, it is reasonable to assume that the change scenarios have no significant influence on the accuracy of the updated models. However, it has to be considered that the influence of certain change scenario types can be hidden by forming the median. Especially because the change scenarios are randomly selected in each experiment run. As a result, if certain change scenarios would have a strong negative influence on the accuracy, this could be hidden in these graphs. Therefore, the meaningfulness of these figures is limited and in a subsequent step the impact per change scenario type was analyzed.

A distinction was made between each change type and the influence on the accuracy was examined. For this purpose the Wasserstein distance and the KS test were used as metrics. The classical statistical measures were omitted, because it already became clear that the significance is limited.

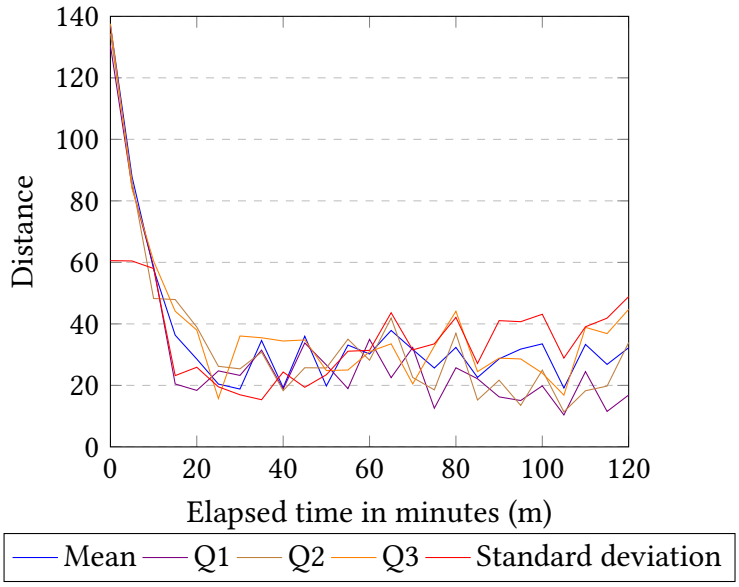
The procedure to measure the loss of accuracy is slightly different. We can not use the data from Experiment 3 here because we do not know when the effect of a certain change scenario will become apparent. For example, the increase of the load does not have an immediate effect, but rather a delayed one. Since we cannot estimate the timing of these effects, an additional experiment is required. In this experiment we used a system configuration where all different change scenarios can be executed (i.e. there are components that have already been replicated and can be de-replicated). For each change



(a) Wasserstein distance over time



(b) KS test over time



(c) Distance between conventional statistical measures over time

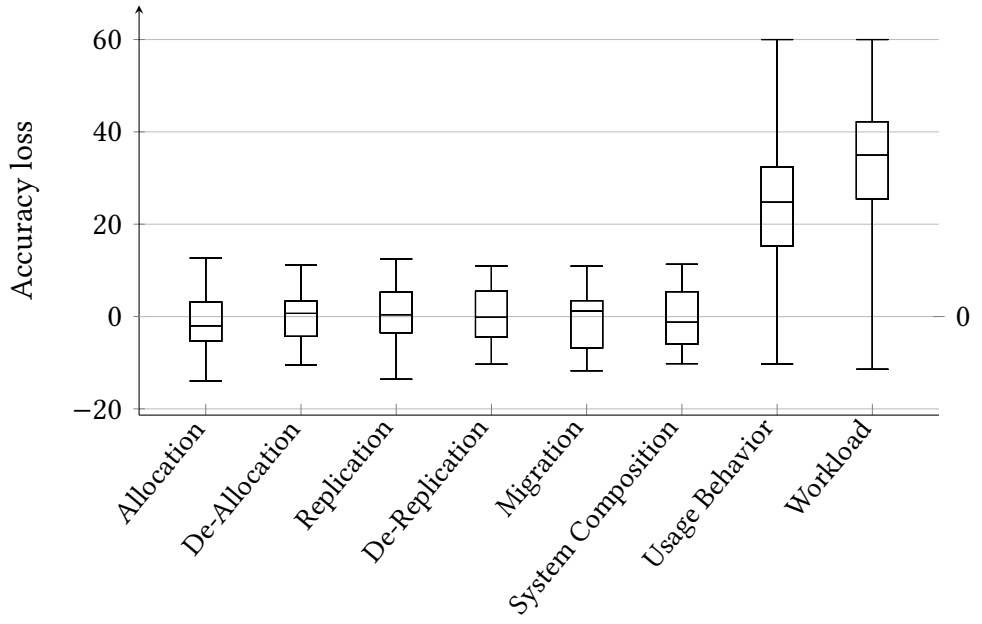
Figure 5.16.: Overview of the accuracy metrics over time for the TeaStore case study (Experiment 3)

type, we carry out 10 runs and only perform one change of the specified type. In contrast to the previous experiment setup, the changes of the types *U5* (Workload) and *U6* (Usage behavior) are not rolled back. Before we execute the change, the unchanged system is observed for 45 minutes. After the change has been executed, the system is observed for another 45 minutes. Finally, to quantify the loss of accuracy, we calculate the distances between the metrics before the change and those after the change. In order to avoid extreme outliers due to measurement errors, the upper and lower 5% of the measured distances were removed. Without this correction the resulting box plot would be very confusing and difficult to interpret.

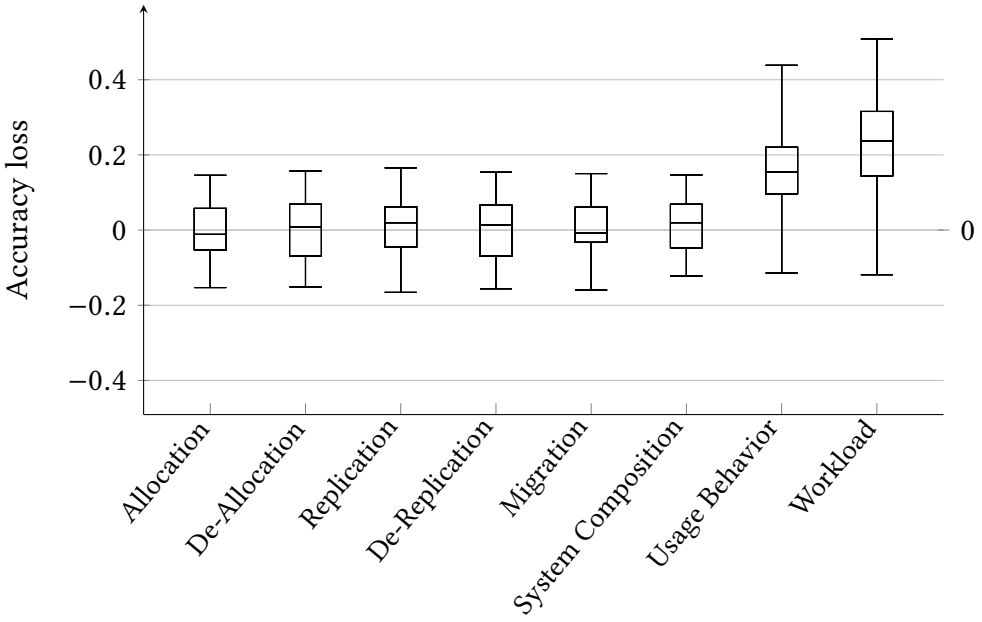
The box plots in Figure 5.17 show the ranges of the loss of accuracy, grouped according to the change types. As already mentioned, the Wasserstein distance and the KS test were used and shown separately. For the graphics concerning the Wasserstein metric, the maximum was capped at 60 to improve the readability.

The graphs indicate that both the allocation and the de-allocation of resource containers have no significant negative or positive effect on the accuracy of the run-time models. The same applies for replications, de-replications, migrations and changes to the system composition. In the case of workload changes and changes to the user behavior, it can be seen that the accuracy of the models decreases noticeably. The reason for this is that the TeaStore can encounter an overload situation due to these changes. In this case, the performance characteristics depend on many factors (operating system, hardware, ...). For a performance model it is therefore very difficult to make precise statements about the response times. It does indeed depict the overload, but the metrics then logically deviate significantly, because the model can no longer represent the high complexity of the software system in reality. Since workload changes and changes of the usage behavior can lead to exactly these excessive loads, the results are reasonable. It should be noted that we have not examined the effect of combinations of changes here. Due to the fact that certain types of changes only have an effect after an indefinite period of time, more complex experiments would be necessary.

Summary In summary, it can be stated that the accuracy of the models can be preserved or improved to a large extent, even when change scenarios occur. For certain change types a negative impact on the accuracy of the models was observed, caused by bringing the system into an overload situation. However, this is not a problem of our approach, but rather a general problem. In such situations, it is almost impossible to predict the exact performance behavior of today's systems, because numerous factors influence it. These include, for example, the operating system, the scheduling strategy and the hardware. In future work, the influence of certain change scenarios could be investigated further, especially the effect of specific combinations of change scenarios has not been considered yet.



(a) Wasserstein distance



(b) KS test

Figure 5.17.: Aggregated representation of the loss of accuracy caused by the different change types by means of box plots, which visualize the quartiles

5.8. Transformation Pipeline Performance

The second objective of the evaluation (G2) concerns the performance of the transformation pipeline under realistic conditions. Furthermore, it is intended to examine the overhead that arises from monitoring the application under consideration. Additional attention is also paid to the observation of the fine and coarse granular monitoring. In this context, it is interesting to see how the Instrumentation Model (IM) can be used to minimize the extent of the fine-granular monitoring and how this affects the execution times of the transformations. For this purpose, no separate experiments were performed, instead the observations were integrated into the experiments already shown. In other words, we evaluate the performance of the pipeline on the basis of the experiments on the accuracy of the derived models (see Section 5.7). In the following sections the results are presented, split up according to the two case studies.

5.8.1. CoCoME

First of all, we analyzed the number of monitoring records entering the pipeline as input for CoCoME. For this purpose, after each pipeline iteration the number of processed monitoring records was recorded. This allows us to plot the amount of monitoring data over time. It should be noted that a sliding window is used, that decides which monitoring data is actually considered (see Section 4.7.2). Since we collect the performance data together with the previous data about the accuracy, measurements from several experiment executions are available. Therefore, as in the evaluation of the accuracy of the models, we calculate medians to eliminate outliers and measurement errors. Figure 5.18 shows the number of monitoring records in the sliding window over time. It also shows the point in time when the fine grained monitoring was deactivated for the “bookSale” service (indicated by a dashed vertical line). We used the performance measurements from Experiment 2 and formed the median values (for both the number of monitoring records and the time until the first deactivation of the fine granular monitoring).

In the graphic it can be seen that the fine grained monitoring is deactivated after approximately 25 minutes of operation. Within this time span 5 pipeline executions were triggered in the experiment. After deactivating the fine-grained monitoring for the “bookSale” service the number of monitoring records decreases to about 22000 and remains steady thereafter. The time span of 25 minutes until the first deactivation of fine granular monitoring is relatively short when considering that it takes time until the sliding window is filled with representative monitoring data. At the peak of the curve there are 31518 monitoring records in the sliding window. After deactivating the fine-granular monitoring the number drops to an average of 22304, which is a reduction of about **29.23%**. This clearly shows that a significant improvement can be achieved by controlling the fine granular monitoring via the IM. The reduction in the number of monitoring records reduces the load on the pipeline and also lowers the monitoring overhead.

Figure 5.19 shows the medians of the execution times of the pipeline parts over time. For reasons of simplicity, not all pipeline executions were shown, only every second one. The execution time is divided into the three parts with the highest shares: the model

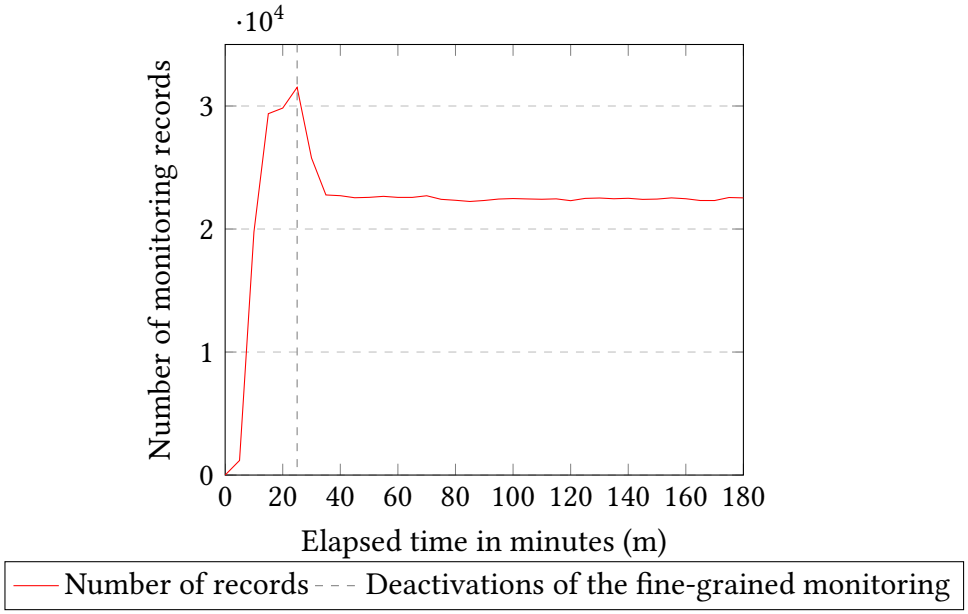


Figure 5.18.: Number of monitoring records in the sliding window over time while observing CoCoME

validations, the usage model derivation and the repository model calibration. All other parts have been grouped together as " miscellaneous".

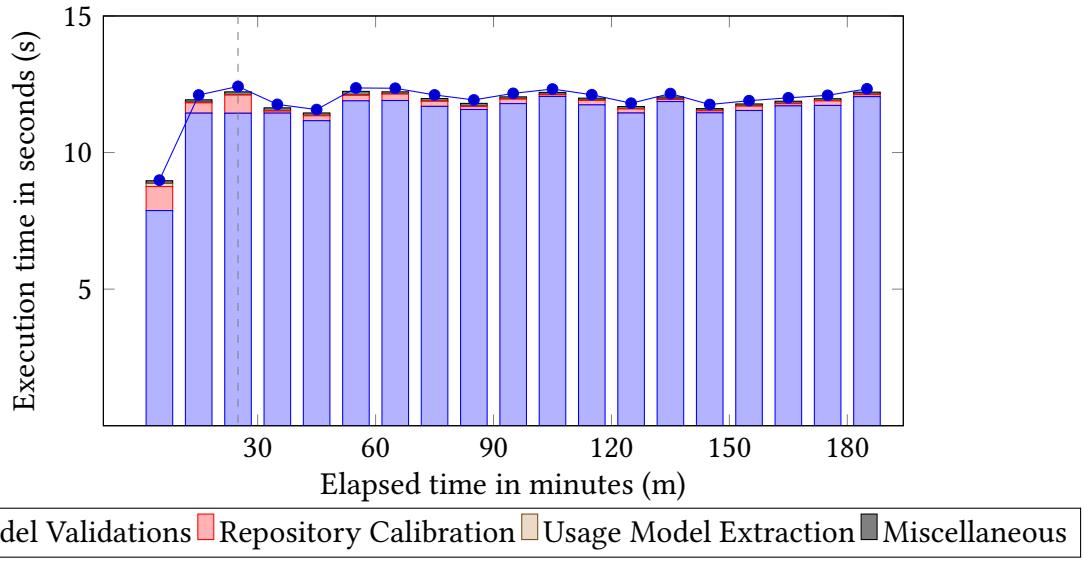


Figure 5.19.: Detailed performance information on the transformation pipeline over time (while executing Experiment 2 for CoCoME).

At first glance, it can be seen that the execution times of the pipeline are dominated by the validations. More precisely, by the simulations of the models within the pipeline. For each pipeline execution, four simulations are performed which cause the major part of the execution time in the experiment with CoCoME. Furthermore, it is apparent that

the deactivation of the fine granular monitoring does not have a significant effect on the execution time. The reason for this is that a reduction of the monitoring data volume mainly affects the calibration of the repository model. Since the execution time of the calibration is negligible in this case, the improvement by lowering the monitoring records is negligible as well. Table 5.8 recapitulates the execution times of the individual parts of the pipeline.

Pipeline Part	Min	Q1	Q2	Q3	Max
Model Validations	7.876s	11.441s	11.691s	11.866s	12.060s
Repository Calibration	0.097s	0.111s	0.144s	0.161s	0.877s
Usage Model Extraction	0.004s	0.006s	0.010s	0.013s	0.118s
Miscellaneous	0.006s	0.065s	0.071s	0.079s	0.167s
Accumulated	**7.983s**	**11.623s**	**11.916s**	**12.119s**	**13.222s**

Table 5.8.: Overview of the aggregated execution times of the individual pipeline parts in the context of Experiment 2 for the CoCoME case study

Finally, the monitoring overhead was examined for CoCoME, i.e. the time required to generate and send the monitoring records. This is particularly relevant since it directly affects the application under observation and may affect its performance characteristics. For the "bookSale" service an average monitoring overhead of **1.45ms** (fine-grained) and **259μs** (coarse-grained) per call was measured. Given an average execution time of 250ms (of the "bookSale" service) this corresponds to **0.58%** (fine-grained) and **0.1036%** (coarse-grained) of the total execution time. In this specific case it can be seen that deactivating the fine granular monitoring significantly reduces the monitoring overhead. In the CoCoME case study, the overhead was reduced by up to **82.14%** on average.

5.8.2. TeaStore

The performance analysis for the TeaStore is structurally identical to the previous one for CoCoME. First, the number of monitoring records per pipeline execution was analyzed, then the execution times of the functional units in the pipeline and finally the arising monitoring overhead. We only present the results for Experiment 3 and not for Experiment 2, because they are similar to a large extent. Experiment 3 also deals with a more difficult case (with change scenarios), so the results are more meaningful.

Figure 5.20 shows the number of monitoring records that are entered as input into the pipeline at certain points in time.

As with CoCoME, it can be seen that the number of monitoring records in the sliding window increases at the beginning until the point is reached where the finely granular monitoring is deactivated. Afterwards, the amount of monitoring data decreases and then settles at a constant level. At the peak, there are 120546 monitoring records in the sliding window and after deactivating the fine granular monitoring, the average decreases to approximately 90000, which corresponds to a reduction of the monitoring data by **25.34**%.

Figure 5.21 shows the execution times for the complete pipeline and for the respective functional units.

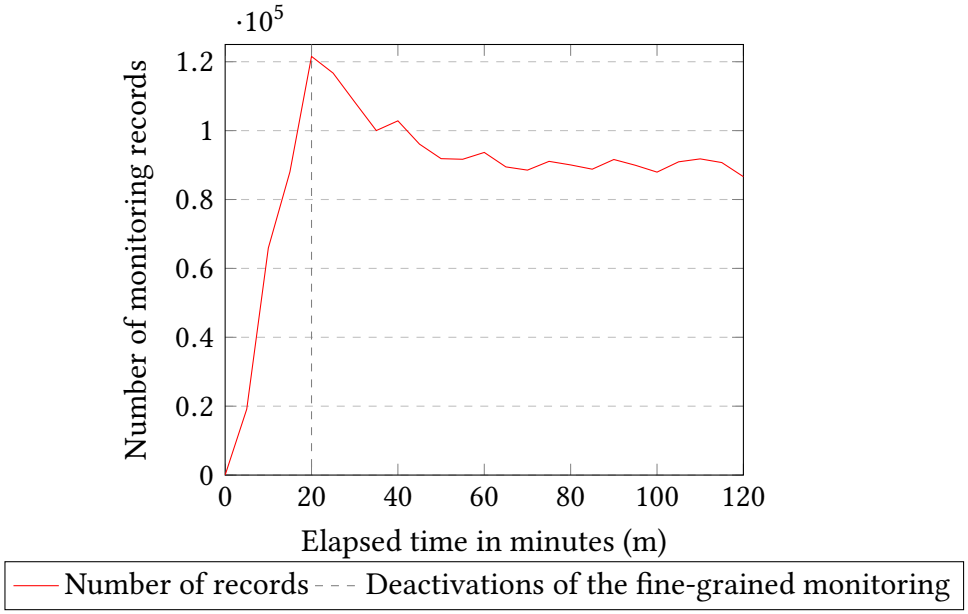


Figure 5.20.: Number of monitoring records in the sliding window over time while observing TeaStore

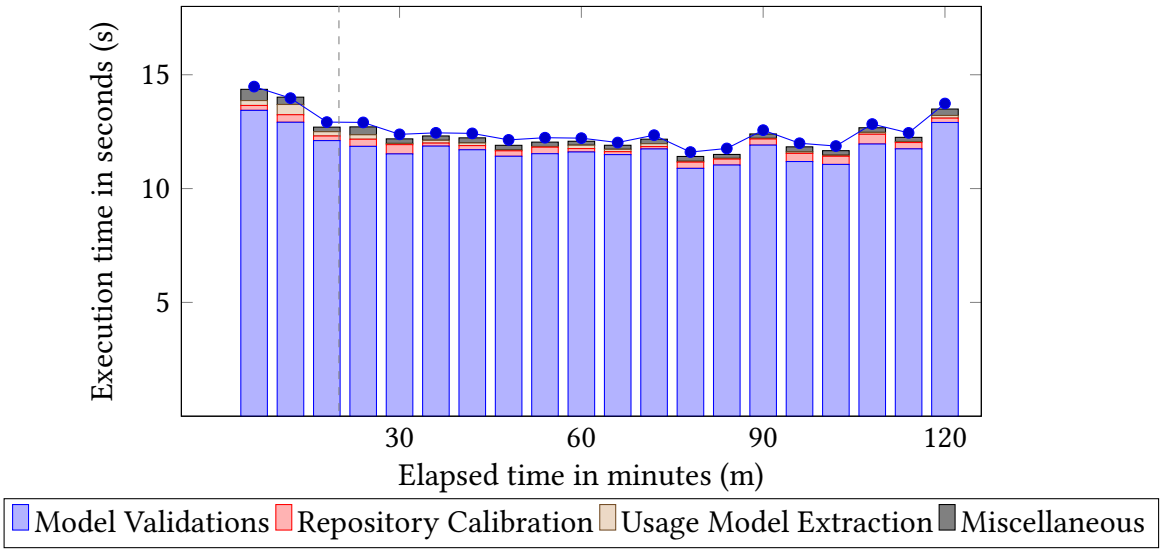


Figure 5.21.: Detailed performance information on the transformation pipeline over time (while executing Experiment 3 for TeaStore).

The conclusion here is similar to CoCoME, it can be clearly seen that the execution time of the pipeline is dominated by the validations of the models. Furthermore, it can be seen that the execution times tend to be slightly higher than with CoCoME, which is mainly due to the fact that the PCM model of the TeaStore is more complex. Logically, this also leads to an increase in simulation time. Again, the effect of the decreasing monitoring data volume on the execution time cannot be observed. The reason for this is again the very short execution time of the transformations within the pipeline. Because only these

are positively influenced by the decreasing number of monitoring records, no significant impact on the execution time is visible in the chart. Table 5.9 summarizes the aggregated data about the execution times once again in tabular form.

Pipeline Part	Min	Q1	Q2	Q3	Max
Model Validations	10.745s	11.245s	11.705s	12.228s	14.635s
Repository Calibration	0.102s	0.141s	0.208s	0.295s	0.704s
Usage Model Extraction	0.021s	0.043s	0.074s	0.126s	0.449s
Miscellaneous	0.095s	0.190s	0.205s	0.239s	0.651s
Accumulated	**10.963s**	**11.619s**	**12.192s**	**12.888s**	**16.439s**

Table 5.9.: Overview of the aggregated execution times of the individual pipeline parts in the context of Experiment 3 for the TeaStore case study

The monitoring overhead was also examined for the TeaStore case study. The analyzed service is "confirmOrder", like in Experiment 2 and 3. The following values were obtained regarding the overhead caused by the monitoring:

- **Absolute (fine-grained)**: 2.8551ms
- **Absolute (coarse-grained)**: 949.59μs

Assuming an average response time of 500ms of the "confirmOrder" service, the following relative values (percentages) for the ratio between monitoring overhead and total execution time of the service result:

- **Relative (fine-grained)**: 0.571%
- **Relative (coarse-grained)**: 0.19%

It can be stated that neither the coarse granular nor the fine granular monitoring has a significant influence on the response time of the service. Furthermore, as with CoCoME, it can be seen that the effort for the coarse-granular monitoring is significantly lower than for the fine-granular monitoring. Comparing the two types of monitoring in this case shows that coarse-grained monitoring causes up to **66.741%** less overhead. Therefore, it can also be concluded for the TeaStore that the monitoring overhead can be significantly reduced by controlling the fine granular monitoring via the Instrumentation Model.

5.8.3. Summary

By means of Experiment 2 and 3 of the evaluation (E2 and E3), sufficient metrics are available to answer all scientific questions regarding Goal 2 (G2).

For both case studies (CoCoME and TeaStore), we measured the time needed for the pipeline to complete. In both cases the execution time was almost always less than 15 seconds and was dominated by the model validations. It can therefore be concluded that with typical time intervals between pipeline executions (several minutes) no bottlenecks occur and the data can always be processed in time (**Q2.1**). If the performance of the

simulations would be improved even further, the execution times of the complete pipeline could be reduced in the future (see Section 7).

Regarding the control of the monitoring via the Instrumentation Model (**Q2.2**), it became clear that in both case studies the models were accurate enough to deactivate parts of the fine granular monitoring after about 25 minutes. In both cases this resulted in a reduction of the monitoring volume. However, a drop of the execution time of the transformations within the pipeline could not be observed. As already mentioned in the previous sections, the execution times of the considered transformations were too low, so no clear correlation could be proven.

The third and last question (**Q2.3**) addresses the overhead caused by the monitoring. In both case studies it turned out that the overhead is negligible for the considered services. Furthermore, it became clear that the overhead can be significantly reduced by switching from fine to coarse granular monitoring. In the presented scenarios, it was possible to save up to 90% of the monitoring overhead.

5.9. Scalability of the transformations

The third and final goal of the evaluation (G3) focuses on the theoretical scalability of the transformations within the pipeline. It is especially interesting how the execution times behave when we generate the input data in such a way that a worst-case scenario occurs. Because even under these circumstances it is important that the performance of the pipeline does not collapse. To achieve this, it must be ensured that the transformations within the pipeline have acceptable execution times even in realistic worst-case scenarios. In order to evaluate this, we examined all transformations individually. The procedure is identical for all of them, first the worst-case scenarios are identified and then synthetically generated monitoring data is used as input. The monitoring data is built in such a way that it covers exactly the identified worst-case scenarios. Next, the amount of monitoring data is successively increased and the execution times of the transformations are observed. This provides insights on how the execution times of the transformations scale in the worst-case scenarios. In order to reduce measurement errors and outliers, the analyses were carried out several times and the mean value was calculated. We decided to do 20 repetitions per experiment. With the help of this analysis, reliable statements can be made about the upper limits of the execution times.

5.9.1. Repository Model Transformation

First of all, the scalability of the repository model transformation has been analyzed. It is responsible for the calibration of the resource demands and the adaptation of other stochastic expressions in the model (for example, the number of loop iterations). The transformation can roughly be divided into two parts. In the first part the validation results are analyzed and the results of this analysis are used as input for the second step which executes the regressions. The analysis of the validation results is negligible regarding the execution times. The results are iterated only once and scaling factors are calculated. Even if the simulations are configured with excessive simulation times and measuring points,

an execution time of **one second** is never exceeded. Therefore we will only consider the second part in the following scalability analysis.

A regression is performed for each stochastic expression that needs to be calibrated. The number of data points within a regression is variable and depends on the monitoring data. This can be well illustrated using the example of internal actions whose resource demands need to be calibrated. There are two factors that influence the execution time of the transformation: the number of internal actions that were observed and the number of data points that were recorded for each internal action. The number of observed internal actions corresponds to the number of triggered regressions and the number of data points directly affects the duration of the regressions. Based on this, we built the scenarios that are considered in the scalability analysis. First, we examined the execution times of the transformation for an increasing number of internal actions. Subsequently, we observed the run-times for an increasing number of data points for a single internal action. The results are summarized in Figure 5.22.

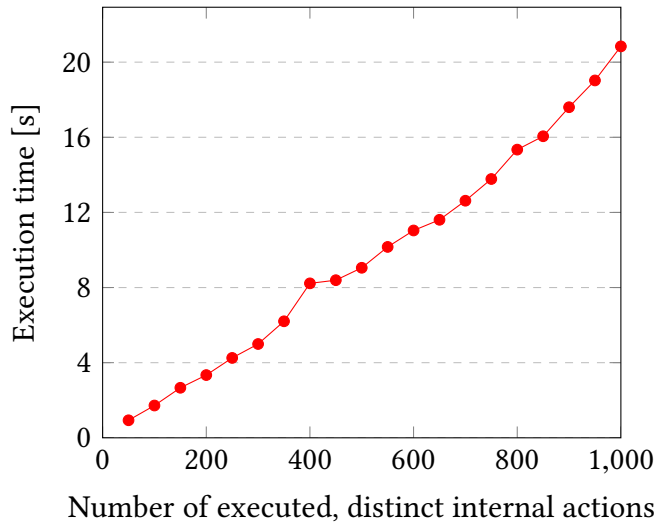
In both cases it can be seen that the duration of the transformation scales linearly with the increasing parameter. Even for a high number of Internal Actions (a), the growth of the execution time remains linear. In addition, the considered number of Internal Actions will probably not occur in practice. In the PCM model of CoCoME there are **15** Internal Actions and in the PCM of the TeaStore there are **17**. Exactly the same can be observed when increasing the data points per internal action. It should also be noted here, that the duration of the regression is entirely dependent on the used implementation. In our realization we use the Waikato Environment for Knowledge Analysis (Weka)[6]. Again, the numbers of considered data points are unrealistic, in both case studies such a data volume was never reached. The transformation took less than 4 seconds to execute in both cases (see Section 5.8).

In summary, it can be concluded that the transformation also scales linearly in the worst-case scenarios and therefore no unexpected side-effects emerge.

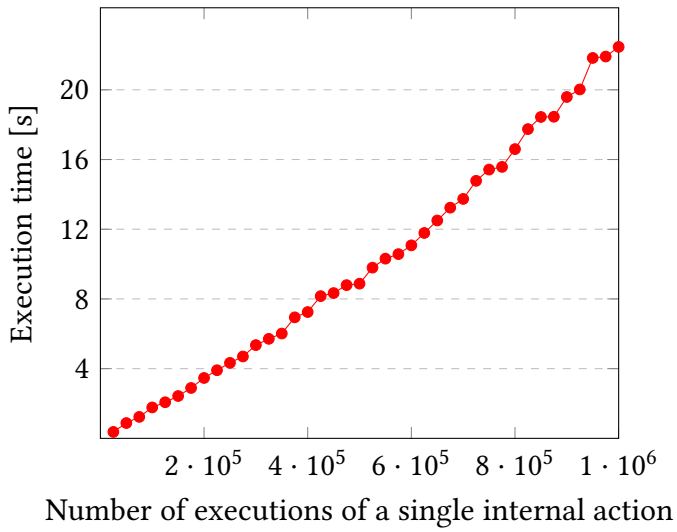
5.9.2. Resource Environment Transformation

The resource environment transformation identifies changes to the hosts and the network connections within the run-time environment. The first step is to scan all service call traces for new hosts and new connections. The detected hosts and connections are then inserted into the run-time Environment Model (REM). Using the consistency rules based on Vitruvius, the corresponding resource containers and linking resources are created in the Resource Environment Model. In the third and final step, a check is made for each container in the Resource Environment Model to see whether it still exists in the run-time environment. The run-time of the transformation is dominated by the second step, which is the change propagation via Vitruvius. In the following, we will examine the executiom time of the entire transformation. With an increasing number of new hosts and connections in the run-time environment, the execution times rise as well. Therefore, we consider two scenarios in this scalability analysis:

[6] https://www.cs.waikato.ac.nz/ml/weka/



(a) Scalability of the transformation with an increasing number of executed distinct internal actions (each of them is executed 1000 times)



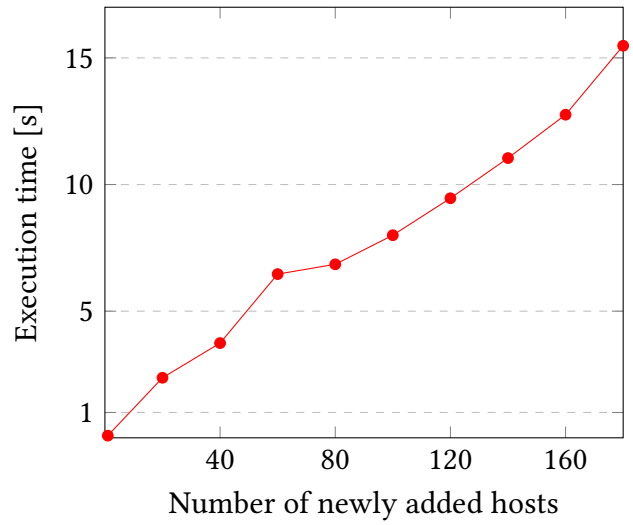
(b) Scalability of the transformation with an increasing number of executions of a single internal action

Figure 5.22.: Exploration of the scalability of the repository transformation under various circumstances

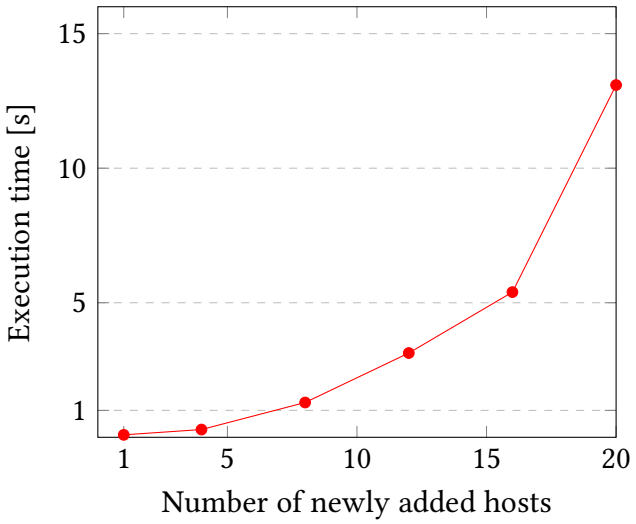
1. Increasing number of new hosts; sparse meshed - indicating that each of the new hosts has only one network connection
2. Increasing number of new hosts; fully meshed - indicating that each of the new hosts has a connection all other hosts

The results for both scenarios are visualized in Figure 5.23.

In the chart that shows the scalability of sparse meshed run-time environments (a), it can be seen that the run-time scales almost perfectly linear with up to 180 new hosts. Such



(a) Scalability of the transformation with an increasing number of newly added hosts; each new host has **only one** connection to another host



(b) Scalability of the transformation with an increasing number of newly added hosts; each new host has a connection **to all** other hosts (fully meshed)

Figure 5.23.: Analysis of the scalability of the resource environment transformation in different scenarios

a scenario seems very unrealistic in practice. For realistic values of about 40 hosts or less, an execution time of 4 seconds is not exceeded. In contrast, the graph on the scalability when adding fully meshed hosts shows that execution times increase exponentially. This is simply because the connections between hosts need to be synchronized one by one. With the hosts, the number of connections increases exponentially when considering a fully meshed network. This results in an exponential increase of the execution time. However, it can still be concluded that the transformation provides adequate execution times for

practical use cases. An appearance of more than 15 new fully meshed hosts between two pipeline executions will probably never occur in practice.

In summary, the scalability analysis for the resource environment transformation has shown that the execution times scale appropriate for common use cases.

5.9.3. System Model and Allocation Model Transformation

The transformations that are responsible for updating the Allocation Model and the System Model were reviewed together within the scalability analysis. The reason for this is that both transformations address the system composition and were designed to operate in conjunction with each other (see Section 4.8.4). For the derivation of updates in the system model, the number of changes in the component composition is crucial. On the other hand, for the derivation of updates in the allocation model, the number of changes in the deployments is crucial. Consequently, these two parameters determine the design of the scalability analysis. First, we synthetically generate a Repository Model with a large number of components and afterwards a Resource Environment Model with a large number of resource containers. The exact number of components and containers is not important, but it must be ensured that sufficiently large change scenarios can be created. Then the actual change scenarios are generated. First, the number of changes is determined, one half is populated with deployment changes and the other half with changes to the component composition. These change scenarios are generated with different sizes and used as input for the combination of both transformations ($T_{Allocation}$ and T_{System}). Figure 5.24 shows the cumulated execution time of both transformations for an increasing number of changes in system composition.

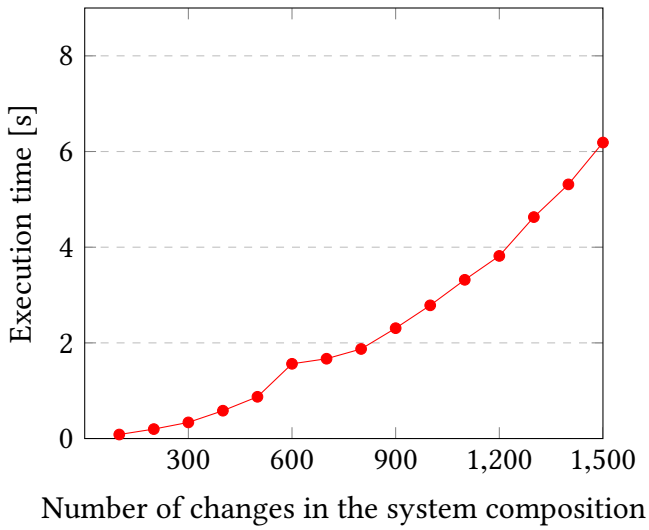


Figure 5.24.: Execution times of System Model transformation with an increasing number of changes in the system composition

The chart shows that the execution times are relatively low for all considered cases. Even for a total number of 1500 Changes (750 deployment changes and 750 component composition changes) the execution time barely exceeds 6 seconds. Because such a number

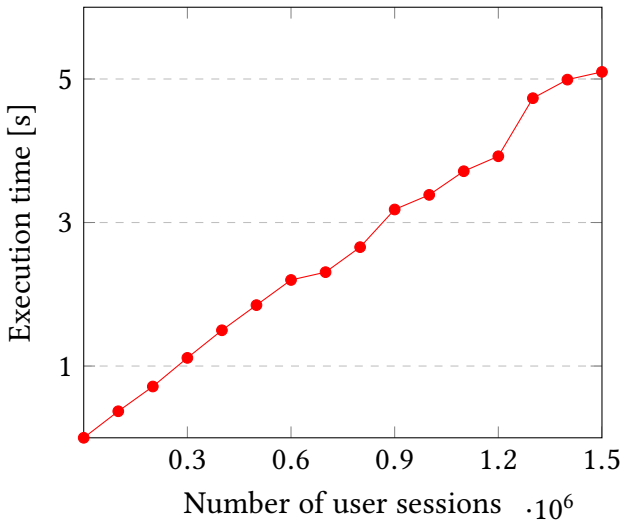
of changes between two pipeline executions probably never occurs in practice, it can be concluded that both transformations scale well. For a realistic number of changes of less than 300, the execution time is significantly lower than one second.

5.9.4. Usage Model Transformation

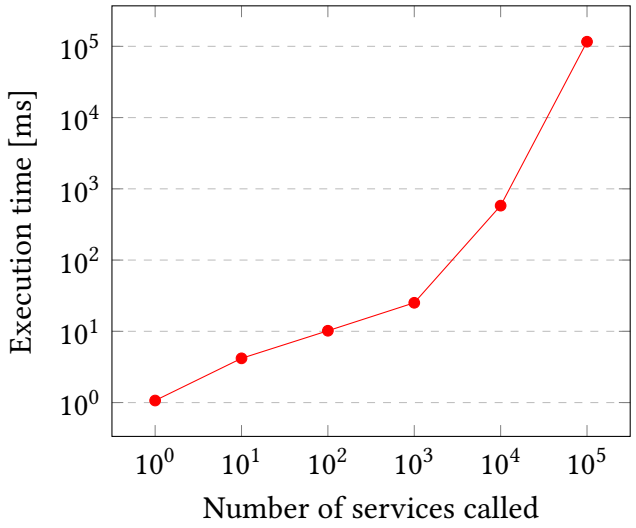
The usage model transformation was adopted from iObserve and ported to our monitoring data structure. Hence, both approaches are conceptually identical. A detailed scalability analysis has already been done for iObserve [30]. The goal of the scalability analysis in the context of this thesis is to show that the results are consistent with those of iObserve. We consider two different cases. Firstly, an increasing number of users, all of them triggering exactly one service call and secondly, an increasing number of service calls triggered by a single user. Figure 5.25 shows the scalability analysis for both scenarios. Here, (a) shows the increase in execution times for a rising number of users and (b) shows the growth for an increasing number of service calls initiated by a single user. When looking at the sub figure (b) it should be noted that the axes are scaled logarithmically. In this way, we wanted to ensure that the results can be compared to those obtained from the iObserve scalability analysis.

The first experiment shows that the execution time scales almost perfectly linear with an increasing number of users (sessions). The same conclusion can be drawn from the results of iObserve's scalability analysis [30]. We also obtained consistent results when analyzing the execution time for an increasing number of service calls initiated by a single user. For 100 or less initiated service calls, the execution time increases sublinearly and thereafter superlinearly with an explosive growth in execution time above 10000 initiated service calls. In the superlinear segment, the execution time is dominated by the loop detection [30]. The extreme increase of the execution time with a high number of triggered service calls is not critical, because such a user behavior is very unlikely in practice.

In summary, it can be stated that the results of our scalability analysis are in line with those of iObserve. Therefore, it can also be concluded that the execution times of the usage model transformation are appropriate.



(a) Scalability of the transformation when the number of users increases; each user triggers a single service call



(b) Scalability of the transformation with an increase in the number of triggered service calls for a single user (axes are scaled logarithmically)

Figure 5.25.: Scalability analysis of the Usage Model transformation

5.10. Evaluation Summary

The evaluation is based mainly on case study research. For this purpose the cloud based webshops CoCoME and TeaStore were used. The objectives of the evaluation focused on ensuring the accuracy of the derived models (**G1**), examining the performance of the transformation pipeline (**G2**), and finally analyzing the scalability of the transformations (**G3**).

First, the accuracy of the models was investigated. Therefore three experiments were performed. In the first experiment the system extraction at design time was examined

and it was shown that the extracted models match with reference models. Furthermore, it became clear that the manual effort, which is necessary in this procedure, can be reduced by a static code analysis. In the second experiment both case studies were used and monitored under load. The monitoring data were then used as input for the transformation pipeline and the output models were evaluated for accuracy. As ground truth the monitoring data was used. For both case studies it was shown that the accuracy of the models increases over time and then remains very solid. The third experiment is conceptually almost identical to the second one, the only difference is that we performed change scenarios to the case study during the monitoring. In this case, only the TeaStore was considered. The results were similar to those of Experiment 2 for the TeaStore and showed that the accuracy is preserved despite changes at run-time. The only exceptions were changes in workload and user behavior, where the TeaStore experienced an overload situation. As a result, the simulation results drifted strongly away from the actual response times. The reason for this is that in such scenarios many parameters have an influence on the performance characteristics. These include, for example, the operating system, the scheduling strategy and the hardware. A performance model can hardly represent these complex interrelationships, which leads to a loss of accuracy. Therefore this is not a problem of our approach, but rather a general problem in performance modeling.

In the second step the performance of the pipeline and the monitoring overhead was evaluated. For this purpose we used the measurements that were already collected during Experiment 2 and 3. In other words, we use the performance measurements that were collected during the evaluation of the accuracy of the models. It was shown that the execution times are low enough to eliminate inconsistencies quickly and to avoid bottlenecks. It became clear that the execution times are dominated by the simulations of the models. In addition, the influence of the Instrumentation Model on the execution times was investigated. No positive effect could be shown, because the execution times of the transformations was too low. To quantify the effect of the monitoring on the application under observation, we also measured the monitoring overhead. It was shown that the monitoring overhead is negligible for both case studies. Furthermore, it was shown that the coarse granular monitoring causes significantly less overhead. Therefore, the Instrumentation Model can help to minimize the monitoring overhead in the long run.

In the third and last step the scalability of the transformations was analyzed in general. For the repository calibration it was shown that it always scales linearly with the number of monitoring records to be processed. For the transformations concerning the resource environment, the allocations and the system composition it was shown that they provide low run-times and a good scalability for realistic scenarios. Finally, for the usage model transformation it was shown that the scalability is identical to iObserve. Although we use large parts of iObserve, this relationship was not trivial. The underlying monitoring data structure is different and so is the necessary preprocessing.

In summary, it can be concluded that the proposed approach is suitable for maintaining consistency for software adaptation and evolution scenarios.

5.11. Threats to validity

The identification of the threats to validity is based on guidelines for evaluations in the context of case study research [64]. Therefore, we distinguish between four dimensions of validity: *internal validity*, *external validity*, *construct validity* and *conclusion validity*. In the following, we discuss the individual aspects in the context of our approach.

- **Internal Validity**: A threat to validity is the selection of metrics within the evaluation. These include the metrics introduced in Section 5.4. Strengths and weaknesses have already been analyzed in that section. For the comparison of distributions we used the Wasserstein distance, the KS-test and conventional statistical measures. These have been used in many related studies and by combining them we minimize the risk that a single metric can distort the evaluation results. By aggregating the results over several experiment executions it is possible that certain effects are hidden. Especially if not every experiment execution is identical (as for example in Experiment 3), then the impact of certain change scenarios can be masked by forming the median. This was compensated by the fact that the results of the experiments were grouped by change scenario type and not by the execution time in an additional experiment. Thus, the impact of the different change scenarios on the accuracy of the models was determined. This risk does not exist for the experiments where each run is identical in terms of execution (Experiment 2). In future work, the influence of the change scenarios on the accuracy of the models could be investigated with further experiments and case studies to make even more meaningful statements. The Jaccard coefficient was used to compare PCM models and has also been used in related work (e.g. iObserve). The equality definition of model elements was defined recursively and static model elements are used as anchors (see Section 5.4). Thus, an equality of two PCM models can be shown.

- **External Validity**: Another threat to validity is the selection of case studies. It may be possible that the results obtained from the case studies are not valid in general. To avoid this, we first performed a requirements analysis for the case studies (see Section 5.5) and then selected suitable case studies. This led to the selection of CoCoME and TeaStore. Both are widely used in research and address common use cases (cloud-based web stores). This ensures that the case studies cover common business use cases. Furthermore, the TeaStore has been designed to compare approaches in the areas of performance modeling and performance analysis [39]. By combining the two selected case studies, the risk of non-representative results is further reduced. In the future, the approach could be tested in further case studies, but the current evaluation based on CoCoME and TeaStore already allows to make substantiated statements.

- **Construct Validity**: In the evaluation we rely on a combination of synthetically generated monitoring data and monitoring data generated directly by executing a case study. For the synthetically generated data, external factors such as the run-time environment or the type of load testing can be excluded. In addition, an arbitrary amount of monitoring data can be generated and thus also extreme cases can be

analyzed. When observing the case study, however, the quality of the monitoring events must be ensured. Since we transfer the monitoring records via the network to the pipeline, it is also necessary that the latency is as low as possible to ensure that the monitoring data is up-to-date. To guarantee the quality of the monitoring events we use the Kieker Framework with extensions that have already been implemented in previous projects [32, 56, 35]. As load driver JMeter [26] is used and executed directly on the container that also handles the execution of the case study. This ensures a low latency. Furthermore, the pipeline and the monitored case study are connected to the same local network, which minimizes the network overhead. Finally, the monitoring was implemented in a way that the resulting monitoring overhead is eliminated from the execution times. As a result, it can be made sure that the behavior of the application with monitoring is almost the same compared to the one without.

- **Conclusion Validity**: The subjectivity of a researcher must be avoided when interpreting the evaluation results. Many different, well known metrics have been used, which are easy to understand and interpret. In addition, these have been introduced and explained in a separate section in order to enhance the reader's understanding. Through the quantification with the help of the metrics there is not much room for interpretation of a researcher. All conclusions and arguments are well structured and based on the metrics, making them easy to understand. Due to this design of the evaluation hardly any other interpretations and conclusions are possible.

6. Related Work

In this section we focus on approaches that also deal with the automated consistency preservation between architectural software artefacts. In addition, we also mention approaches in the area of view-based modeling, since some aspects of our work are also related to this area.

6.1. Consistency Preservation of Architectural Models and Source Code

Konersmann proposed a solution which creates a connection between architecture implementation and architecture specifications (e.g. architecture models) [42]. It integrates information about the architecture model directly into the program code via annotations [42]. The approach uses Intermediate Architecture Description Language (IAL) to decouple the representation of architecture information from concrete architecture and implementation models [42]. The focus of this approach lies on providing an explicit mapping between implementation elements and elements in the architecture model via annotations, whereas our approach concentrates on automated consistency management between software artefacts.

ArchLint [53] is an approach that supports the automated detection of architectural violations. To do so, ArchLint uses static and historical source code analysis techniques. Internally, a graph is built that maps the dependencies between the classes. This graph and heuristics are used to detect "architectural violations" [53]. The consistency check is performed in an extra step at design-time, in contrast to our approach which also considers adaptation scenarios at run-time and adjusts the models automatically.

Just-In-Time Tool for Architecture Consistency (JITTAC) [11] also deals with the consistency checking between architecture models and their corresponding Java implementation. It is organized as an Eclipse plugin and allows the user to visualize the components of the architecture and interdependencies in a graph with nodes and edges. The components can now be mapped to the source code and JITTAC checks if all dependencies are satisfied. Like ArchLint, this is a static approach that helps keeping models consistent at design-time, but also does not consider run-time data and is not able to automatically maintain consistency throughout adaption scenarios.

There are a number of approaches in the area of architecture model extraction at run-time. As we do not want to name them all, we refer to [72] for a detailed list. However, these approaches all work exclusively with run-time data (for example, extraction of information from method call traces). Our approach, on the other hand, takes run-time and design-time into account. Furthermore, these approaches do not take the system composition into account when maintaining consistency; this was included in our approach.

6.2. View-based Consistency

A view-based approach was developed by Meier and Winter [57]. It uses a so-called Single Underlying MetaModel (SUMM) which covers all data within a domain. Additionally, the initial models are migrated to views of the SUM. In contrast to Vitruvius, it supports the elimination of initial inconsistencies. The approach is operator-based, i.e. the transformations between the initial models and the single underlying model (SUM) are broken down into small, understandable operations with the help of the so-called operators [57]. The approach was implemented in Java as a prototype and will most likely be extended in the future. However, consistency is only considered at design-time.

A slightly older approach has been presented by Mens et al. [58], it supports the consistency preservation between high-level structural views and the source code. But only the path from source code to high-level design is supported, i.e. the consistency is unidirectional. Relations between elements in the high-level view are defined and managed with the help of the "Relation Editor". In addition, known deviations between source code and high-level structural view are stored in the Relation Editor. The so-called Relation Checker then verifies these relationships and outputs violations. In contrast to our approach, no run-time data is taken into account and the violations are not fixed automatically.

7. Future Work

While working on the thesis, it turned out that there is still potential for extensions and improvements in many areas. Especially the validations that were integrated within the pipeline open up many opportunities for enhancements. But also the limitations and assumptions from Chapter 3 may be eliminated by future work. The following list names and describes the identified future work tasks with the highest potentials:

- **Extraction of the repository model from the bytecode of the application**: Langhammer's approach to the synchronization between the repository model and the application assumes that the corresponding source code is available. However, it would also be conceivable to derive the repository model directly from the bytecode of the application. Although some information is lost when the source code is translated into bytecode (e.g. loops), there are already analysis tools that can reliably reconstruct this information [73]. In this way, the initial overhead for model-driven development could be lowered even more.

- **Consideration of the performance characteristics of the elements in the Resource Environment Model**: As already mentioned in Section 3.4.1 among the limitations (L2), the current approach does not consider performance attributes of links and containers within the resource environment model. For instance, by fully monitoring the network requests, one could calculate accurate values for latency and throughput. Also, by measuring the performance of the containers itself, one could estimate the capabilities of their hardware resources.

- **Machine learning approaches for model updates within the pipeline**: With the help of machine learning techniques, it would be possible to further refine the transformation of the repository model and the transformation of the usage model. Furthermore, with the validations that have been woven into the pipeline as an integral part, it is possible to use the deviations of the models as input for the transformations. This makes it possible to design continuously learning transformations, for example with Reinforcement Learning [71]. Another place where machine learning can be helpful is directly in the Repository Model. The resource demands within the Repository Model could be designed as neural networks with continuous output. The monitoring data would then be used for training, and as a result the neural network would "learn" the behavior of the resource demand over time. For this, however, the simulation engine would have to be extended, since it must be able to calculate the output of a neural network.

- **Optimization of the PCM simulations in order to reduce the execution time**: In the evaluation it became clear that the simulations of the derived models have

a major impact on the performance of the pipeline. Within this thesis a headless variant of the simulation engines has been used which significantly reduces the execution time. For most use-cases the faster simulation engine SimuLizar [4] is unfortunately not sufficient and SimuCom [9] has to be used. If the simulations would be more performant, more validations could be added to the pipeline and maybe even a world of models could be simulated. Finally, the model that performs best within the validation could be selected.

- **Support of composite components in the automated extraction of the system model**: Another limitation of the current approach is that composite components are not supported in the actual implementation (see L3 in Section 3.4.1). As already mentioned there, this is not a conceptual limitation, but rather an enormous implementation effort, which is why it has been omitted in this thesis. However, it would be conceivable to eliminate this limitation, e.g. when deriving the System Model at run-time, the service call graph could be searched for composite components.

- **Merging the approach with existing work in this area**: There are already many approaches and related projects in the context of this thesis. For example, building blocks of iObserve [29] have been integrated into our approach. For example, Grohmann et. al. have already developed an approach that detects parametric dependencies using feature selection techniques [25]. It would make sense to combine the advantages of the approaches in this area with future extensions.

8. Conclusion

In this thesis we presented an approach for maintaining the consistency between software artifacts, with a special focus on supporting evolution and adaptation scenarios. We pointed out gaps of current approaches and explained how they are closed within this thesis. Our approach combines different approaches concerning consistency maintenance and introduces new features. In particular, it is intended to ensure consistency between architecture models and source code at design-time and between architecture models and a running application at run-time. In this context, the widely used and acknowledged Palladio Component Model (PCM) was applied to represent the architecture models [6]. The implementation is based on iObserve, Vitruvius and the CIPM approach. In a comprehensive case study based evaluation, various aspects were evaluated, including the accuracy of the models, the performance and the scalability.

First of all, the thesis clearly defined the objectives of the work, explained the most important terms and introduced limitations. Afterwards the conception of the approach itself was described. First, the extensions of the Virtual Single Underlying Model (VSUM) were introduced, compared to the already existing approaches of Mazkatli and Langhammer [55, 50]. Subsequently, the changes and extensions of the monitoring were described. The Instrumentation Metamodel was extended in order to be able to fine-tune the monitoring at run-time and thus control the overhead. In addition, the monitoring record types, which define the data structure of the monitoring data, were adapted. By means of the Eclipse Modeling Framework (EMF), two new model types were developed: the Service Call Graph (SCG) and the Runtime Environment Model (REM). The SCGs are used to represent call relationships between services. On the other hand, the REM is used to represent the run-time environment and to synchronize it with the architecture model via consistency rules.

Based on these extensions and adaptations, the concepts of design-time and run-time consistency preservation were introduced. Thereby we focused on the consistency management of the system composition, as this was not covered by existing approaches. At design-time, our approach is largely based on consistency rules, which have been established with the help of Vitruvius (see coevolution approach of Langhammer [50]). At run-time, a transformation pipeline is used, which was inspired by the preliminary work of the CIPM approach and iObserve [56, 29]. To create the mapping between elements of different models, the correspondence model of Vitruvius is used. It replaces the run-time architecture correspondence model (RAC) [62], which was introduced and used by iObserve for this purpose. In total, consistency is supported for all PCM model parts at run-time and at design-time only the Repository Model and the System Model are supported.

In the evaluation, the developed approach was examined using two different case studies: CoCoME [65] and TeaStore [39]. Both are widely used systems that have been applied

for evaluating a number of related approaches in the field of performance modeling and analysis. The accuracy and applicability of the System Model extraction at design-time have been analyzed and demonstrated. Furthermore, the accuracy of the models at run-time was evaluated. It was shown that the quality of the models increases significantly in the beginning and then stabilizes at a very good level. For TeaStore, change scenarios were taken into account during this analysis, i.e. changes were carried out at run-time and their influence on the model accuracy was observed. It turned out that most of the changes have no significant influence. A notable negative impact on the accuracy of the models was found only for workload changes and changes in user behavior. In both cases this is because the changes can cause the system to enter an overload situation, making the response times unpredictable. For both case studies, the performance of the transformation pipeline was examined and it was discovered that the execution times were below 15 seconds in almost all cases. A major part of the execution times were caused by the simulations of the PCM models. However, the results also showed that the times are low enough to not endanger the applicability in practice. The monitoring overhead was also reviewed and we obtained a similar result as in the performance analysis. By continuously adjusting the monitoring, the overhead is reduced as much as possible and therefore the performance characteristics of the observed application are not significantly affected. Furthermore, the scalability of the transformations in extreme situations was evaluated. The results indicated that all transformations scale adequately for a large number of real-world use cases.

In summary, it can be stated that this thesis developed an extensive approach to the consistency management of software artifacts. It combines advantages and features of different approaches and extends them by aspects that have not been considered so far. In particular, it is designed to cover and support design-time (evolution) and run-time (adaptation) changes. Special attention was also paid to extensibility, as the approach was implemented in the form of a framework. This means that new functions can be added and existing features adapted in the future without much effort. By selecting two representative case studies for the evaluation of the approach, the practical applicability could be shown. However, this does not guarantee that our approach works as intended for all possible systems. Therefore, it makes sense to further develop and evaluate the approach in future work.

Bibliography

[1] C. Atkinson and T. Kuhne. "Model-driven development: a metamodeling foundation". In: *IEEE Software* 20.5 (2003), pp. 36–41.

[2] Luciano Baresi. "Activity Diagrams". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 41–45. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_9. URL: https://doi.org/10.1007/978-0-387-39940-9_9.

[3] Victor R. Basili, Gianluigi Caldiera, and Dieter H. Rombach. "The Goal Question Metric Approach". In: vol. I. John Wiley & Sons, 1994.

[4] Matthias Becker, Markus Luckey, and Steffen Becker. "Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2013, pp. 43–52. ISBN: 9781450321266. DOI: 10.1145/2465478.2465489. URL: https://doi.org/10.1145/2465478.2465489.

[5] Steffen Becker. "Coupled model transformations for QoS enabled component-based software design". PhD thesis. 2008. 297 pp. ISBN: 978-3-86644-271-9. DOI: 10.5445/KSP/1000009095.

[6] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio Component Model for Model-driven Performance Prediction". In: *J. Syst. Softw.* 82.1 (Jan. 2009), pp. 3–22. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.03.066. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066.

[7] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. 2nd. Packt Publishing, 2016. ISBN: 1786464969.

[8] Walter Binder, Jarle Hulaas, and Philippe Moret. "Advanced Java Bytecode Instrumentation". In: *ACM International Conference Proceeding Series* 272 (Jan. 2007). DOI: 10.1145/1294325.1294344.

[9] Fabian Brosig et al. "Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures". In: *Software Engineering, IEEE Transactions on* 41.2 (Feb. 2015), pp. 157–175. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2362755.

[10] Yuriy Brun et al. "Engineering Self-Adaptive Systems through Feedback Loops". In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–70. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_3. URL: https://doi.org/10.1007/978-3-642-02161-9_3.

[11] Jim Buckley et al. “JITTAC: A Just-in-Time tool for architectural consistency”. In: *2013 35th International Conference on Software Engineering (ICSE)* (2013), pp. 1291–1294.

[12] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003. ISBN: 0131425420.

[13] Erik Burger. “Flexible Views for Rapid Model-driven Development”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’13. Montpellier, France: ACM, 2013, 1:1–1:5. ISBN: 978-1-4503-2070-2. DOI: 10.1145/2489861.2489863. URL: http://doi.acm.org/10.1145/2489861.2489863.

[14] Erik Burger. “Flexible Views for View-based Model-driven Development”. PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology, July 2014. ISBN: 978-3-7315-0276-0. DOI: 10.5445/KSP/1000043437. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043437.

[15] F. Buschmann. *Pattern-orientierte Software-Architektur: ein Pattern-System*. Professionelle Softwareentwicklung. Addison-Wesley, 1998. ISBN: 9783827312822. URL: https://books.google.de/books?id=o2nuK0qpo3QC.

[16] Jordi Cabot and Martin Gogolla. “Object Constraint Language (OCL): A Definitive Guide”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 58–90. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_3. URL: https://doi.org/10.1007/978-3-642-30982-3_3.

[17] Noureddine Dahmane. “Adaptive Monitoring for Continuous Performance Model Integration”. Master Thesis. Karlsruher Institut für Technologie Fakultät für Informatik, Mar. 14, 2019.

[18] Martin Davis. *Computability & Unsolvability*. Dover Publications, 1958.

[19] Peter J. Denning and Jeffrey P. Buzen. “The Operational Analysis of Queueing Network Models”. In: *ACM Comput. Surv.* 10.3 (Sept. 1978), pp. 225–261. ISSN: 0360-0300. DOI: 10.1145/356733.356735. URL: https://doi.org/10.1145/356733.356735.

[20] Hans-Friedrich Eckey, Reinhold Kosfeld, and Martina Rengers. *Multivariate Statistik*. Jan. 2002. DOI: 10.1007/978-3-322-84476-7.

[21] Otávio Freitas Ferreira Filho and Maria Alice Grigas Varella Ferreira. “Semantic Web Services: A RESTful Approach”. In: *IADIS International Conference WWWInternet 2009*. IADIS, 2009, pp. 169–180. URL: http://fullsemanticweb.com/paper/ICWI.pdf.

[22] The Eclipse Foundation. *Eclipse desktop & web IDEs*. https://www.eclipse.org/ide/. visited on 05/09/2020. 2020.

[23] The Eclipse Foundation. *Xtext - Language Engineering Made Easy!* https://www.eclipse.org/Xtext/index.html. visited on 05/09/2020. 2020.

[24] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. AMS, 2003. URL: http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html.

[25] Johannes Grohmann et al. "Detecting Parametric Dependencies for Performance Models Using Feature Selection Techniques". In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Oct. 2019, pp. 309–322. DOI: 10.1109/MASCOTS.2019.00042.

[26] Emily Halili. *Apache JMeter*. Packt Publishing, 2008. ISBN: 1847192955.

[27] Lucia Happe et al. "Software Engineering Processes". In: *Modeling and Simulating Software Architectures – The Palladio Approach*. Ed. by Ralf H. Reussner et al. Cambridge, MA: MIT Press, Oct. 2016. Chap. 9, pp. 195–225. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[28] Florian Heidenreich et al. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 374–383. ISBN: 978-3-642-12107-4.

[29] R. Heinrich. "Architectural run-time models for performance and privacy analysis in dynamic cloud applications". In: *ACM SIGMETRICS performance evaluation review* 43.4 (2016), pp. 13–22. ISSN: 0163-5999, 1557-9484. DOI: 10.1145/2897356.2897359.

[30] Robert Heinrich. "Architectural runtime models for integrating runtime observations and component-based models". In: *Journal of Systems and Software* 169 (2020). ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110722. URL: http://www.sciencedirect.com/science/article/pii/S016412122030159X.

[31] Robert Heinrich. "Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications". In: *ACM SIGMETRICS Performance Evaluation Review* 43.4 (2016), pp. 13–22. ISSN: 0163-5999. DOI: 10.1145/2897356.2897359. URL: http://doi.acm.org/10.1145/2897356.2897359.

[32] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22–25, 2012: ACM, Apr. 2012, pp. 247–248. ISBN: 978-1-4503-1202-8.

[33] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis". In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pp. 247–248. URL: http://eprints.uni-kiel.de/14418/.

[34] André van Hoorn et al. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Tech. rep. TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009, 27 pages.

[35] Jan-Philipp Jägers. "Iterative Performance Model Parameter Estimation Considering Parametric Dependencies". Master Thesis. Karlsruher Institut für Technologie Fakultät für Informatik, May 13, 2019.

[36] Frédéric Jouault and Ivan Kurtev. "Transforming Models with ATL". In: *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*. MoDELS'05. Montego Bay, Jamaica: Springer-Verlag, 2005, pp. 128–138. ISBN: 3540317805. DOI: 10.1007/11663430_14. URL: https://doi.org/10.1007/11663430_14.

[37] Reiner Jung, Robert Heinrich, and Eric Schmieders. "Model-driven Instrumentation with Kieker and Palladio to Forecast Dynamic Applications". In: *Symposium on Software Performance*. CEUR Vol-1083, 2013, pp. 99–108.

[38] Alexander Keller and Heiko Ludwig. "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services". In: *J. Netw. Syst. Manage.* 11.1 (Mar. 2003), pp. 57–81. ISSN: 1064-7570. DOI: 10.1023/A:1022445108617. URL: https://doi.org/10.1023/A:1022445108617.

[39] Jóakim von Kistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2018, pp. 223–236.

[40] Heiko Klare. "Designing a Change-Driven Language for Model Consistency Repair Routines". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2016. DOI: 10.5445/IR/1000080138. URL: http://dx.doi.org/10.5445/IR/1000080138.

[41] "Kolmogorov–Smirnov Test". In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 283–287. ISBN: 978-0-387-32833-1. DOI: 10.1007/978-0-387-32833-1_214. URL: https://doi.org/10.1007/978-0-387-32833-1_214.

[42] Marco Konersmann. "Explicitly Integrated Architecture - An Approach for Integrating Software Architecture Model Information with Program Code". en. PhD thesis. May 2018. URL: https://duepublico2.uni-due.de/receive/duepublico_mods_00045949.

[43] Heiko Koziolek et al. "Evaluating Performance of Software Architecture Models with the Palladio Component Model". In: *Model-Driven Software Development: Integrating Quality Assurance*. Ed. by Jörg Rech and Christian Bunse. IDEA Group Inc., Dec. 2008, pp. 95–118.

[44] Max E. Kramer, Erik Burger, and Michael Langhammer. "View-centric Engineering with Synchronized Heterogeneous Models". In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO '13. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2. DOI: 10.1145/2489861.2489864. URL: http://doi.acm.org/10.1145/2489861.2489864.

[45] Max Emanuel Kramer. "Specification Languages for Preserving Consistency between Models of Different Languages". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017. 278 pp. DOI: 10.5445/IR/1000069284. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-692845.

[46] Max E. Kramer et al. *Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius*. Tech. rep. Karlsruhe: Karlsruhe Institute of Technology, Department of Informatics, 2015. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-456541.

[47] Max E. Kramer et al. *Realizing Change-Driven Consistency for Component Code, Architectural Models, and Contracts in Vitruvius*. Tech. rep. 4. Karlsruher Institut für Technologie (KIT), 2015. DOI: 10.5445/IR/1000045654.

[48] Klaus Krogmann. "Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis". PhD thesis. 2012. 371 pp. ISBN: 978-3-86644-804-9. DOI: 10.5445/KSP/1000025617.

[49] Ivan Kurtev. "State of the Art of QVT: A Model Transformation Language Standard". In: *Applications of Graph Transformations with Industrial Relevance*. Ed. by Andy Schürr, Manfred Nagl, and Albert Zündorf. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 377–393. ISBN: 978-3-540-89020-1.

[50] Michael Langhammer. "Automated Coevolution of Source Code and Software Architecture Models". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), 2017. 259 pp. DOI: 10.5445/IR/1000069366. URL: http://nbn-resolving.org/urn:nbn:de:swb:90-693666.

[51] Michael Langhammer et al. "Automated Extraction of Rich Software Models from Limited System Information". In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), Venice, Italy, 5–8 April 2016*. IEEE, Piscataway (NJ), 2016, pp. 99–108. ISBN: 978-1-5090-2131-4. DOI: 10.1109/WICSA.2016.35.

[52] Longjie Li et al. "A Linear Approximate Algorithm for Earth Mover's Distance with Thresholded Ground Distance". In: *Mathematical Problems in Engineering* 2014 (Mar. 2014), pp. 1–9. DOI: 10.1155/2014/406358.

[53] Cristiano Maffort et al. "ArchLint: Uma Ferramenta para Detecção de Violações Arquiteturais usando Histórico de Versões". In: *Congresso Brasileiro de Software: Teoria e Prática (CBSoft), Sessão de Ferramentas*. submitted. 2013, pp. 1–6.

[54] Szymon Majewski et al. "The Wasserstein Distance as a Dissimilarity Measure for Mass Spectra with Application to Spectral Deconvolution". In: *18th Int. Workshop on Algorithms in Bioinformatics (WABI 2018)*. Ed. by Laxmi Parida and Esko Ukkonen. Vol. 113. Leibniz Int. Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 25:1–25:21. ISBN: 978-3-95977-082-8.

[55] Manar Mazkatli and Anne Koziolek. "Continuous Integration of Performance Model". In: *Proc. of the 4th International Workshop on Quality-Aware DevOps in Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: ACM, 2018, pp. 153–158. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3186285. URL: http://doi.acm.org/10.1145/3185768.3186285.

[56] M. Mazkatli et al. "Incremental calibration of architectural performance models with parametric dependencies". In: *IEEE 17th International Conference on Software Architecture (ICSA 2020); Salvador, Brazil, November 2-6, 2020*. 17th International Conference on Software Architecture. ICSA 2020 (Salvador da Bahia, Brasilien, Nov. 2–6, 2020). IEEE Computer Society, Los Alamitos, 2020, pp. 23–34. ISBN: 978-1-7281-4659-1. DOI: 10.1109/ICSA47634.2020.00011.

[57] Johannes Meier and Andreas Winter. "Model Consistency ensured by Metamodel Integration". In: *6th International Workshop on The Globalization of Modeling Languages (GEMOC), co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. Ed. by Regina Hebig and Thorsten Berger. Copenhagen: CEUR Proceedings of MODELS 2018 Workshops, Oct. 2018, pp. 408–415.

[58] Kim Mens et al. "Co-evolving Code and Design with Intensional Views". In: *Comput. Lang. Syst. Struct.* 32.2-3 (July 2006), pp. 140–156. ISSN: 1477-8424. DOI: 10.1016/j.cl.2005.09.002. URL: http://dx.doi.org/10.1016/j.cl.2005.09.002.

[59] David Monschein, Robert Heinrich, and Christoph Heger. "Diagnosis of Privacy and Performance Problems in the Context of Mobile Applications". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: ACM, 2018, pp. 167–172. ISBN: 978-1-4503-5629-9. DOI: 10.1145/3185768.3186283. URL: http://doi.acm.org/10.1145/3185768.3186283.

[60] OMG. *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*. Object Management Group, June 2013. URL: http://www.omg.org/spec/MOF/2.4.1.

[61] Felipe Pezoa et al. "Foundations of JSON schema". In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pp. 263–273.

[62] R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl. "Integrating run-time observations and design component models for cloud system analysis". In: *9th Int'l Workshop on Models@run.time* (2014), pp. 41–46.

[63] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628.

[64] Per Runeson et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. Wiley Publishing, 2012. ISBN: 1118104358.

[65] S. Herold et al. "CoCoME – the common component modeling example". In: *The Common Component Modeling Example* (2008), pp. 16–53.

[66] F. Santambrogio. *Optimal Transport for Applied Mathematicians: Calculus of Variations, PDEs, and Modeling*. Progress in Nonlinear Differential Equations and Their Applications. Springer International Publishing, 2015. ISBN: 9783319208282. URL: https://books.google.de/books?id=UOHHCgAAQBAJ.

[67] Yves R. Schneider and Anne Koziolek. "Towards Reverse Engineering for Component-Based Systems with Domain Knowledge of the Technologies Used". In: *Proceedings of the 10th Symposium on Software Performance (SSP)*. Softwaretechnik Trends. 2019, pp. 35–37.

[68] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. ISBN: 9783211811061. URL: https://books.google.de/books?id=DK-EAAAAIAAJ.

[69] Thomas Stahl et al. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2nd ed. Heidelberg: dpunkt, 2007. ISBN: 978-3-89864-448-8.

[70] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.

[71] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[72] Michael Szvetits and Uwe Zdun. "Systematic Literature Review of the Objectives, Techniques, Kinds, and Architectures of Models at Runtime". In: *Softw. Syst. Model.* 15.1 (Feb. 2016), pp. 31–69. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0394-9. URL: http://dx.doi.org/10.1007/s10270-013-0394-9.

[73] Raja Vallée-Rai et al. "Soot: A Java Bytecode Optimization Framework". In: *CASCON First Decade High Impact Papers*. CASCON '10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224. DOI: 10.1145/1925805.1925818. URL: https://doi.org/10.1145/1925805.1925818.

[74] Michael Weiss. "XML Metadata Interchange". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 3597–3597. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_902. URL: https://doi.org/10.1007/978-0-387-39940-9_902.

[75] Yun Xu and Royston Goodacre. "On Splitting Training and Validation Set: A Comparative Study of Cross-Validation, Bootstrap and Systematic Sampling for Estimating the Generalization Performance of Supervised Learning". In: *Journal of Analysis and Testing* 2.3 (July 2018), pp. 249–262. ISSN: 2509-4696. DOI: 10.1007/s41664-018-0068-2. URL: https://doi.org/10.1007/s41664-018-0068-2.

A. Appendix

A.1. System Model Derivation at Runtime

Algorithm 3 System Model extraction at run-time using a Service-Call-Graph

Input: Repository Model (REPO), System Model (SYS), Allocation Model (ALLOC), Resource Environment Model (ENV), Correspondence Model (CM), Service Call Traces (SCTS)

Output: Updated PCM System Model

```
1:  createdAssemblyMapping ← emptyMap()
2:  for all sct ∈ SCTS do
3:      scg ← createServiceCallGraph(sct)
4:      entryPoint ← getEntryNode(scg)
5:      processNode(entryPoint)
6:  end for
7:  updateExposedRoles()
8:  removeUnusedAssemblys()
9:
10: function processNode(node)
11:     for all outgoingEdge ∈ node.outgoingEdges do
12:         fromAssembly ← getCorrespondingAssembly(outgoingEdge.from)
13:         toAssembly ← getCorrespondingAssembly(outgoingEdge.to)
14:         provRole ← getProvidedRole(toAssembly, outgoingEdge.to.service.interface)
15:         reqRole ← getRequiredRole(fromAssembly, outgoingEdge.from.service.interface)
16:         if isNew(fromAssembly) & isNew(toAssembly) then
17:             SYS.createConnector(fromAssembly, reqRole, toAssembly, provRole)
18:         else if isNew(fromAssembly) or isNew(toAssembly) then
19:             if isNew(fromAssembly) then
20:                 oldConnector ← SYS.getConnector(toAssembly, provRole))
21:             else if isNew(toAssembly) then
22:                 oldConnector ← SYS.getConnector(fromAssembly, reqRole))
23:             end if
24:             SYS.deleteConnector(oldConnector)
25:             SYS.createConnector(fromAssembly, reqRole, toAssembly, provRole)
26:         end if
27:         processNode(outgoingEdge.to)                    ▷ start recursion
28:     end for
29: end function
```

```
30: function getCorrespondingAssembly(service, container)
31:     for all ac ∈ ALLOC.allocationContexts do
32:         if ac.container = container  &  ac.assembly.component = service.component
    then
33:                 ▷ according to the assumptions this can only occur once in the for-loop
34:             return ac.assembly
35:         end if
36:     end for
37:     if createdAssemblyMapping.containsKey([service, container]) then
38:         return createdAssemblyMapping.get([service, container])
39:     else
40:         nass ← createAssembly(service.component)
41:         ALLOC.createAllocationContext(nass, container)
42:         createdAssemblyMapping.put([service, container], nass)
43:     end if
44: end function
45:
46: function updateExposedRoles
47:     outerRoles ← SYS.providedRoles
48:     innerRoles ← getOpenProvidedRoles(SYS)
49:     for all outerRole ∈ outerRoles do
50:         matchingRoles ← getMatchingRoles(outerRole, innerRoles)
51:         for all matchingRole ∈ matchingRoles do
52:             if isNew(matchingRole.assembly) then
53:                 oldDelegation ← getDelegationFor(outerRole)
54:                 SYS.removeDelegation(oldDelegation)
55:                 SYS.createDelegation(outerRole, matchingRole)
56:             end if
57:         end for
58:     end for
59: end function
60:
61: function removeUnusedAssemblys
62:     unusedAssemblys ← getUnreachableAssemblys(SYS)
63:     for all unreachable ∈ unusedAssemblys do
64:         markAsUnreachable(unreachable)
65:         if shouldDelete(unreachable) then
66:             ALLOC.removeAllocation(unreachable)
67:             SYS.removeAssembly(unreachable)
68:         end if
69:     end for
70: end function
```

A.2. Change Scenarios of Experiment 3

ID	Type	Concrete Change
#1	Migration	Migration of component *Registry* from container *C6* to *C2*
#2	Deallocation	Deallocation of container *C6*
#3	Migration	Migration of component *WebUI* from container *C1* to *C3*
#4	Behavior Change	Change usage behavior
#5	Allocation	Allocation of container *C7*
#6	Deallocation	Deallocation of container *C7*
#7	Migration	Migration of component *Registry* from container *C2* to *C5*
#8	Behavior Change	Change usage behavior
#9	Replication	Replication of component *Recommender* (*Recommender#2*)
#10	Allocation	Allocation of container *C8*
#11	System Composition	Change of recommender implementation to *Order Based*
#12	Replication	Replication of component *Auth* (*Auth#2*)
#13	Workload Change	Change workload to *30* concurrent users
#14	Allocation	Allocation of container *C9*
#15	System Composition	Change of recommender implementation to *Preprocessed Slope One*
#16	Migration	Migration of component *WebUI* from container *C3* to *C2*
...	...	...
#27	Workload Change	Change workload to *40* concurrent users
#28	System Composition	Change of recommender implementation to *Popularity Based*
#29	Behavior Change	Change usage behavior
#30	Allocation	Allocation of container *C10*
#31	Dereplication	Dereplication of component *Auth#2*
#32	System Composition	Change of recommender implementation to *Order Based*
#33	Dereplication	Dereplication of component *Recommender#2*
#34	System Composition	Change of recommender implementation to *Preprocessed Slope One*
#35	Allocation	Allocation of container *C11*
...	...	...

Table A.1.: Truncated example of an exemplary set of change scenarios used for Experiment 3 of the evaluation