Karlsruhe Institute of Technology

Commit-Based Continuous Integration of Performance Models

Master's Thesis of

Martin Armbruster

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

Reviewer:	Prof. Dr.-Ing. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf H. Reussner
Advisor:	M.Sc. Manar Mazkatli
Second advisor:	M.Sc. Timur Sağlam

01. April 2021 – 14. September 2021

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.

Karlsruhe, 14.09.2021

. . . . . .

(Martin Armbruster)

Abstract

Architecture-level performance models such as the Palladio Component Model (PCM) can be used for, e. g., performance predictions to explore design alternatives. It combines the aspects of architecture and performance models. Having an up-to-date architecture model can support the software maintenance by reducing the architectural degradation while performance models allow the investigation of the software performance without the need to implement or change the system. However, keeping them up-to-date requires manual effort which hinders their adoption. Especially in the agile software development which is characterized by incremental and iterative development cycles, no or short design phases prevent manual modeling activities.

Addressing the aforementioned issues with automatized activities, the Continuous Integration of Performance Models (CIPM) approach proposes a pipeline for the Continuous Integration to keep performance models up-to-date. A commit-based integration strategy extracts changes from a commit and incrementally updates a performance model. To estimate the performance model parameters, the source code is adaptively instrumented and monitored. The taken measurements are used to calibrate the performance model. The CIPM approach's realization is based on Vitruvius, Java, and the PCM. Parts of the pipeline were prototypically implemented and evaluated in previous work without forming a complete pipeline.

This thesis presents an approach to solve open issues. In the approach, a Java model for a commit is created and compared to the model of the previous commit to obtain a change sequence. Afterwards, the changes are propagated within Vitruvius to update the PCM, and the source code is adaptively instrumented. Additionally, the approach is evaluated with the TeaStore which is a web-based store for tea and related products. The evaluation results indicate the correct operation of the approach, but reveals several limitations as a base for future work. As a consequence, the approach leads to an improved usability of the CIPM approach and a reduced effort through automatization.

Zusammenfassung

Leistungsmodelle auf architektonischer Ebene wie das Palladio Component Model (PCM) können für z. B. Leistungsvorhersagen genutzt werden, um Designalternativen zu erkunden. Dabei kombiniert es die Aspekte von Architektur- und Leistungsmodellen. Ein aktuelles Architekturmodell kann die Softwarepflege unterstützen, indem es den Architekturzerfall reduziert, während Leistungsmodelle es erlauben, die Softwareleistung ohne die Implementierung oder Änderung eines Systems zu untersuchen. Allerdings benötigt die Aktualisierung der Modelle manuellen Aufwand, der eine Nutzung verhindert. Besonders in der agilen Softwareentwicklung, die durch inkrementelle und iterative Entwicklungszyklen gekennzeichnet ist, wird dies durch keine oder kurze Designphasen verstärkt.

Der Ansatz der kontinuierlichen Integration von Leistungsmodellen (engl. Continuous Integration of Performance Models, CIPM) adressiert die vorherigen Probleme mit automatisierten Aktivitäten, die in einer Verarbeitungs-Pipeline für die Continuous Integration ausgeführt werden, mit der Leistungsmodelle aktuell bleiben. Eine commit-basierte Integrationsstrategie extrahiert Änderungen aus einem Commit und aktualisiert damit das Leistungsmodell inkrementell. Um die Parameter des Leistungsmodells abzuschätzen, wird der Quellcode adaptiv instrumentiert und überwacht. Die entstandenen Messungen werden genutzt, um das Leistungsmodell zu kalibrieren. Die Realisierung des CIPM-Ansatzes basiert auf Vitruvius, Java und dem PCM. Teile der Verarbeitungs-Pipeline wurden in früheren Arbeiten prototypisch implementiert und evaluiert, ohne eine komplette Verarbeitungs-Pipeline zu bilden.

Diese Thesis präsentiert einen Ansatz, um die offenen Probleme zu lösen. Im Rahmen des Ansatzes wird ein Java-Modell für einen Commit erzeugt und mit dem Modell des vorherigen Commits verglichen, um eine Änderungssequenz zu erhalten. Anschließend werden die Änderungen in Vitruvius propagiert, um das PCM zu aktualisieren, und der Quellcode wird adaptiv instrumentiert. Zusätzlich wird der Ansatz mit dem TeaStore, einem web-basierten Geschäft für Tee und dazugehörige Produkte, evaluiert. Die Ergebnisse deuten auf die korrekte Funktionsweise des Ansatzes hin, aber offenbaren auch einige Limitierungen, die eine Basis für zukünftige Arbeiten bilden. Damit führt der Ansatz zu einer verbesserten Nutzbarkeit des CIPM-Ansatzes und einem reduzierten Aufwand durch Automatisierung.

Contents

List of Figures

List of Tables

1. Introduction

Architecture-level performance models, for instance, the Palladio Component Model (PCM), allow performance predictions to evaluate and compare design alternatives [7, 73] combining the aspects of architecture and performance models.

In particular, software architectures degrade over time, i. e., the implemented architecture in the source code differs from the conceptual architecture [5]. This phenomenon is also known as architectural drift [84], architectural erosion [84], architectural degeneration [50], and other terms. In open source software as example, the possible reasons for the degradation include a lack of architecture documentation, time pressure, and frequent changes due to new or updated requirements increasing the complexity [5]. As a result, the time and corresponding costs to implement changes rise [25, 101] while the modularity declines [25]. This indicates that a system becomes harder to maintain. In certain cases, the architectural degradation can lead to a big ball of mud, a software system without a recognizable architecture, [27] which happened to, e. g., the Mozilla web browser [35] that evolved into Firefox [22]. At a certain point of time, such systems require their re-development from scratch [27, 101]. A method to hinder the degradation is the explicit modeling of the currently implemented architecture in order to observe the impact of changes [5, 22, 51]. However, keeping architecture models up-to-date requires manual effort [51] which can prevent their adoption. Furthermore, in agile software development processes, e. g., Scrum, characterized by an iterative and incremental development with a focus on the product [93], no or short design phases reduce modeling activities which can be addressed by automatically executing them.

Software performance is a non-functional requirement which assesses the temporal behaviour and resource utilization of a software system [19]. It is expressed in, e. g., the response time or throughput. The earlier performance issues are identified, the earlier they can be addressed and solved avoiding costly rework at later points in time. Using measurement-based techniques such as the monitoring of an application during its runtime [47] or performance testing [61], the software performance can be evaluated. Nevertheless, it requires the implementation of the system. In contrast, model-based approaches enable performance predictions without the need of an implemented system [19]. On the other hand, as previously mentioned, in the agile software development, short development cycles limit modeling activities and, thus, the use of performance models. Therefore, the automatic execution of such activities reduces the effort and offers performance predictions during the development.

To close the gap between the development and having up-to-date architecture-level performance models, the Continuous Integration of Performance Models (CIPM) approach has been proposed [73]. It incorporates automatically executed activities into a DevOps pipeline to keep performance models up-to-date. During the development, extracted changes of a commit are utilized to incrementally update the performance model. In order

to estimate the performance model parameters (PMPs), the CIPM approach adaptively instruments and monitors the source code. The taken measurements are used to calibrate the performance model. Realized with Vitruvius which is a view-centric approach for model-driven software engineering that represents a system in a Virtual Single Underlying Model (V-SUM) combining different models [67], the V-SUM includes Java models and a PCM instance [73].

In previous work, several pieces of the development part of the pipeline were prototypically implemented [14, 21, 75] and partly evaluated with artificial projects [14, 21]. The monitoring and parameter estimation are available as a pipeline and validated with case studies [75, 74]. A pipeline combining the incremental model updates, adaptive instrumentation, and the existing pipeline is absent. Additionally, such a complete pipeline has not been evaluated with a real world application. As a result, this thesis addresses the aforementioned issues. It presents an approach to solve the problems by adapting and extending the existing implementations to complete the pipeline. Based on the extensions, the combined prototypical implementations are evaluated with a case study as the representation of a real world application.

In the following chapter 2, the foundations of this thesis are explained. Afterwards, the approach is presented in chapter 3 and in detail in chapter 4 and evaluated in chapter 5. chapter 6 gives an overview over related work. Finally, this thesis is concluded in chapter 7.

2. Foundations

This chapter gives an overview over the foundations of this thesis. The basic concepts are covered in section 2.1 (model-driven software development) and section 2.2 (agile software development). The state-based model comparison in section 2.5 supplements the concept of model-driven software development. A modeling environment for Java is described in section 2.3 and for component-based software architectures in section 2.4 and in section 2.8 focused on the Service Effect Specifications (SEFFs). The platform Vitruvius is introduced in section 2.6. Based on Vitruvius, the co-evolution approach is explained in section 2.7, the CIPM approach in section 2.9, and the integration of existing source code in section 2.10.

2.1. Model-Driven Software Development

Compared to model-based software development in which models are mostly used for documentation purposes, models are an integral part of the development process in model-driven software development (MDSD) [96]. Models are as important as the source code and even the source code can be seen as a model of the application. Moreover, transformations enable the conversion between different models with varying abstraction levels and allow automatic source code generation. This leads to the goals of MDSD: it improves software quality by separation of concerns between abstract models and their transformations and by a resulting higher maintainability. Additionally, reuseable abstract models reduce the complexity of an application and contribute to a lower development time.

Three features characterize a model according to Stachowiak [95]. At first, models represent originals (mapping feature) which in turn can also be models. Secondly, only attributes of the represented original that are relevant for the model creator or user are contained in a model (reduction feature). At last, models cannot be unambiguously assigned to their originals and take a replacement function for specific subjects, during specific time intervals, and with specific operations (pragmatic feature).

In order to create models, meta-models are defined that represent a specific domain and describe how models are constructed within the domain [96]. It is said that a model instantiates its meta-model. To achieve their function, meta-models consist of an abstract syntax, at least one concrete syntax, static semantics, and dynamic semantics. The abstract syntax specifies the structure of models, i. e., the constructs and their relationships, while a concrete syntax is one realization of the abstract syntax and describes how models are expressed, for example, in a domain-specific language (DSL). The static semantics declare additional restrictions on models that cannot be expressed with the abstract or concrete synatx. Finally, the meaning of the constructs and relationships in the abstract syntax are explained in the dynamic semantics.

The Object Management Group, Inc. (OMG) standardized the Meta Object Facility (MOF) 2 as a self-describing meta-model [77]. It consists of the Complete MOF (CMOF) and Essential MOF (EMOF) intended for object oriented programming languages and acts as the meta-model for other meta-models specified by the OMG, for instance, the Unified Modeling Language (UML) [78] or the Object Constraint Language (OCL) [76]. The OMG's *XML Metadata Interchange (XMI) Specification* provides an XML-based format to store MOF-based models for exchange [79]. The Eclipse Modeling Framework (EMF) contains various tools for the MDSD [52]. This includes Ecore which is an implementation of an adapted EMOF version, a source code generator, and complete support for XMI.

2.2. Agile Software Development

Agile software development aims at the flexible development of software compared to processes which are based on detailed plans [30]. Agile methods address the problem of constantly changing requirements by employing iterative and incremental development cycles enabling the possibility to react fast to changes. Examples for agile processes are Extreme Programming (XP) and Scrum. In XP, several agile practices are provided with a recommended cycle length between one and two weeks. In contrast, Scrum defines accountabilities, artifacts, and events to build a framework allowing the incorporation of further methods and setting the iteration length to one month [93].

Continuous Integration (CI) is a methodology originating from XP in which every developer regularly, usually daily, commits their changes and updates the repository [28]. Then, the repository is used to automatically build and test the application on a separate CI server. Only if the build is successful, the changes are integrated. Otherwise, the build needs to be fixed. A deployment or build pipeline enables the execution of multiple builds at the same time. Based on CI, the extending Continuous Delivery ensures that the resulting artifacts can be deployed into the production environment [29]. In Continuous Deployment, the artifacts are deployed into the production environment. DevOps combines techniques of the development and operation of applications in order to create a close cooperation between development and operation teams and to reduce the time for incorporating feedback from the operation into the development [45]. Applied methodologies include CI and Continuous Deployment.

According to Fowler and Lewis, Microservices or Microservice architectures are an architectural pattern to compose an application out of a set of small services where each service defines an API and communicates with other services over their API [31]. Every service can be developed and deployed independently by a cross-functional team and using Continuous Delivery. As a result, a Microservice is built for a specific business capability. Additionally, it allows the decomposition of monolithic applications into interconnected services to reduce the application's complexity while rising the complexity for the distributed production environment and for the increased communication [91]. In this context, Fowler and Lewis view services as components and Microservice architectures as a technique to componentize an application whereby they define a component as independently replaceable and upgradeable software unit [31].

A Representational State Transfer (REST) API or RESTful API is an API following the REST architecture style and built upon HTTP [90]. It allows access to and control of identifiable resources by exchanging the state of a resource in a specific representation in a HTTP request and response while the general HTTP communication is state-less.

2.3. Java Model Parser and Printer

The Java Model Parser and Printer (JaMoPP) provides in its original version an EMF-based environment for modeling Java source code [48]. Therefore, JaMoPP contains an Ecore-based Java meta-model conforming to *The Java Language Specification - Third Edition* (JLS 3) specifying the syntax of Java 5 and 6 [41, 55]. JaMoPP also defines the Java syntax in the CS specification language of EMFText [48]. EMFText generates a parser based on ANTLR to create models from source code files and a pretty printer to write source code files from models out of the CS specification [49, 48]. In order to establish the connections between different Java models introduced by, e. g., imports, JaMoPP includes a mechanism to resolve such references [49]. In the remainder of this thesis, the *original JaMoPP version* refers to the previously described version.

In previous work of this thesis' author without a publication, JaMoPP was adapted. As a result, the meta-model contains all features added from *The Java Language Specification - Java SE 7 Edition* (JLS 7) [38] to *The Java Language Specification - Java SE 15 Edition* (JLS 15) [37], e. g., lambda expressions [39] or modules [40]. The dependency to EMFText was removed by replacing the CS specification and generated parser and printer implementations with manual implementations. The manual printer implementation generates valid Java syntax and preserves the semantics, but neither the layout nor the documentation. The manual parser implementation uses the Eclipse Java Development Tools (JDT) Core to transform source code files into an abstract syntax tree (AST) [80] which is converted to the actual model. Here, the following underlying assumption is made: an application is available with its complete source code and dependent libraries, and the application is parsed at once. Based on this assumption, the `ASTParser` of the JDT Core resolves the references between source code files to bindings [17] which are used to establish the references between the models and are also converted to models if there is no corresponding source file for the binding. As a consequence, the reference resolution mechanism of the original JaMoPP version was removed. In the remainder of this thesis, the previously described adapted JaMoPP version is referred to as the *adapted JaMoPP version.*

2.4. The Palladio Component Model

The PCM offers developers a meta-model for describing component-based software architectures [7]. Figure 2.1 shows parts, roles, and outputs within the creation and usage of a PCM instance. It can be used for the analysis of quality features, e. g., for performance predictions. In the context of the PCM, components employ characteristics as defined by

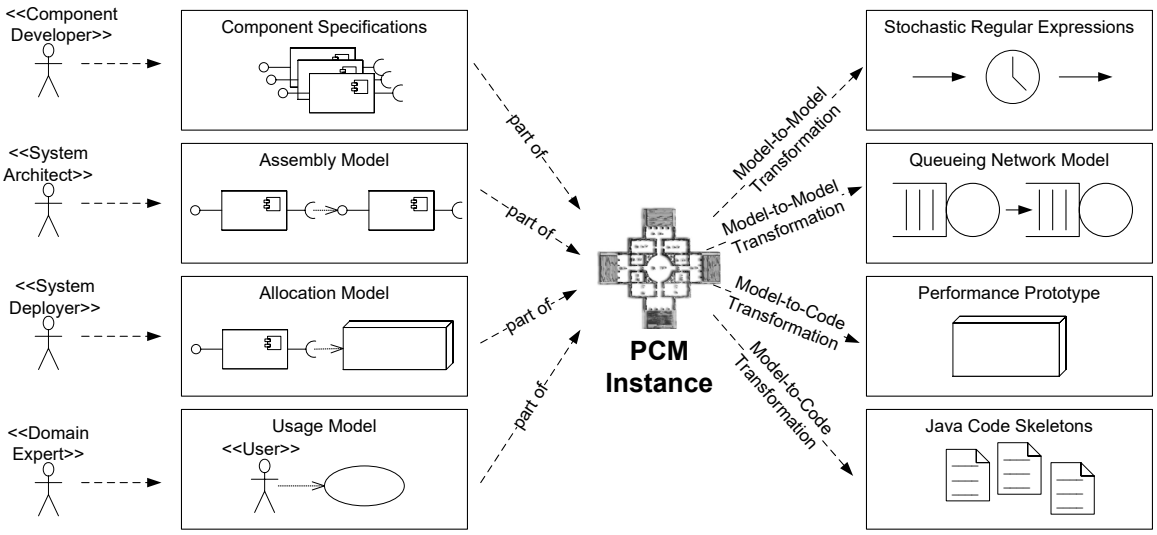


Figure 2.1.: Parts, roles, and outputs associated with the PCM [7].

Szyperski. According to them, components are units for compositions, are independently deployable, and have explicit interfaces and dependencies [98].

During the creation of a PCM instance, component developers specify `BasicComponents` which are stored in the `Repository` [7]. Components provide services by implementing `Interfaces` declaring the services. Additionally, interfaces can be required by components that use their services in their implementation. As interfaces are stored in the repository and are independent from components, `ProvidedRoles` and `RequiredRoles` of components point to the interfaces representing the provided and required interfaces. All interfaces contain signatures for their declared services while the abstract behaviour of services is expressed in SEFFs which are attached to components and described in detail in subsection 2.8.1. At last, a repository contains `DataTypes` representing the parameter and return types of service signatures. The data types are differentiated in `CollectionDataTypes`, `CompositeDataTypes` representing complex and composed data types, and `PrimitiveDataTypes`. Figure 2.2 shows an excerpt from the repository metamodel depicting the aforementioned parts and their connections.

Software architects use the components to build the actual software architecture represented in a `System` [7]. Because a component can be instantiated multiple times in a system, component instances are contained within unique `AssemblyContexts`. Furthermore, `AssemblyConnectors` of `AssemblyContexts` connect provided and required roles of components. At the same time, systems also have provided and required roles exposing the corresponding services. Systems and repositories do not include concrete resources and reference abstract resource types instead. Therefore, system deployers specify `ResourceEnvironments` with concrete resources organized in `ResourceContainers` and `Allocation` models which map `AssemblyContexts` to `ResourceContainers`. In order to complete a PCM instance, domain experts express the user behaviour in usage models. This includes calls to the system's services, the calls' order, and the workload on the system.

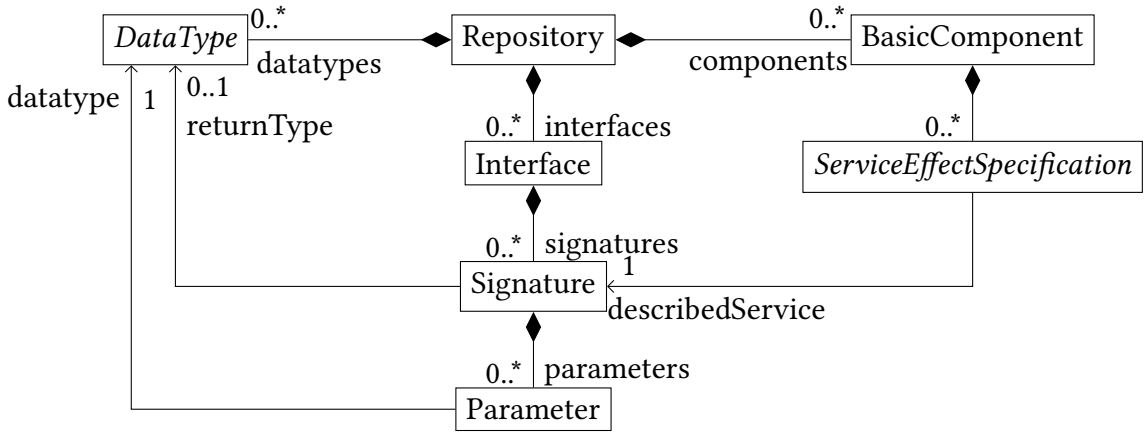


Figure 2.2.: Excerpt from the `Repository` meta-model based on [7].

The PCM is implemented as an Ecore-based meta-model and offers various tools and transformations in Eclipse [7]. Some tools simulate the execution of the system according to the modeled user behaviour. As a result, reponse time distributions are obtained estimating the system's performance.

2.5. State-Based Model Comparison

In order to obtain the differences between two models to enable, e. g., model versioning, the state of the models can be compared by executing a matching, differencing, and representing [8].

During the model matching, corresponding elements in the models are identified [8]. Approaches for determining a match are categorized into the following four categories: identity-based matching in which every element gets an immutable UUID which is used for the matching, signature-based matching in which a signature is calculated for every element based on its features and compared to other signatures, similarity-based matching in which a similarity metric between two elements is calculated, and custom language-specific matching in which the matching rules can be customized to reflect the semantics of the underlying domain.

Based on the matched elements, their actual differences are calculated [8]. Corresponding elements and their features are compared on a fine-grained level. If there is a deviation, a description for the difference is created. If an element has no correspondence, it was added or removed so that the difference's description covers the complete element. All detected differences are atomic operations (add, delete, move, change) represented in a specific style and can form composite operations.

After the differences between two model states have been identified, they can be used to merge both models into one [8].

EMF Compare is a tool for comparing and merging EMF models [105]. Its default similarity-based matching algorithm tries to look up an identifier for an element [24]. If such an identifier cannot be found, EMF Compare computes the distance between the element and other ones using a proximity algorithm. On the resulting matches, the

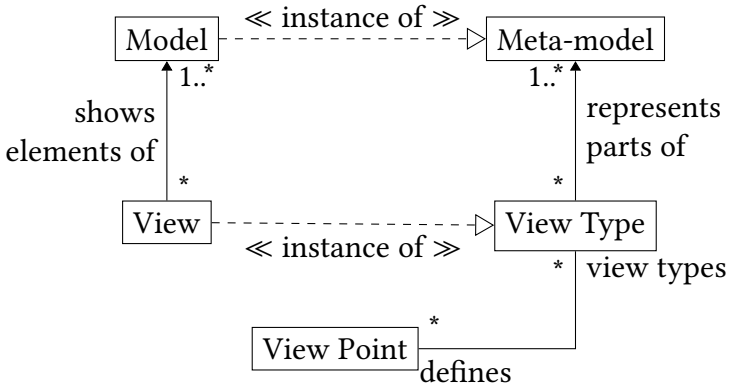


Figure 2.3.: The relationship between views, view types, and view points including models and meta-models based on [69, 67, 36].

differences are calculated and represented in a model conforming to EMF Compare's own Ecore-based meta-model. All algorithms can be replaced with custom implementations or extended by post-processors.

SPLevo is an approach to consolidate custom product copies into software product lines with a prototypical implementation [68]. The first step in the process is a difference analysis of the product copies in which they are compared to derive their differences before the differences are processed further. In detail, the comparison starts with the matching of source code elements by using a hierarchical matching algorithm. During the depth-first traversal, the `SimilarityCheck` compares every element with potential candidates to find similar and thus matching elements. SPLevo's process is independent of a concrete programming language while it provides links for language-specific extensions. The prototypical implementation contains extensions for Java based on the original JaMoPP version. For example, the `SimilarityChecker` considers Java-specific properties when comparing JaMoPP elements [94]. Furthermore, all classes in the prototypical implementation of SPLevo related to the difference analysis build upon EMF Compare.

2.6. VIew-cenTRic engineering Using a VIrtual Underlying Single model

VIew-cenTRic engineering Using a VIrtual Underlying Single model (Vitruvius) is a view-centric approach for the MDSD [69]. In view-based modeling, developers work in views that present a specific part of a software system with information relevant for the developer in order to reduce the complexity. As views are also models, a view type denotes the meta-model of a view. Several view types are grouped into a view point [67] addressing a certain concern [36]. The relationship of views, view types, and view points is depicted in Figure 2.3.

Orthographic Software Modeling (OSM) is a view-based modeling approach in which different dimensions of a system are orthogonal to, i. e., independent of, each other and views are dynamically generated from a Single Underlying Model (SUM) [4].

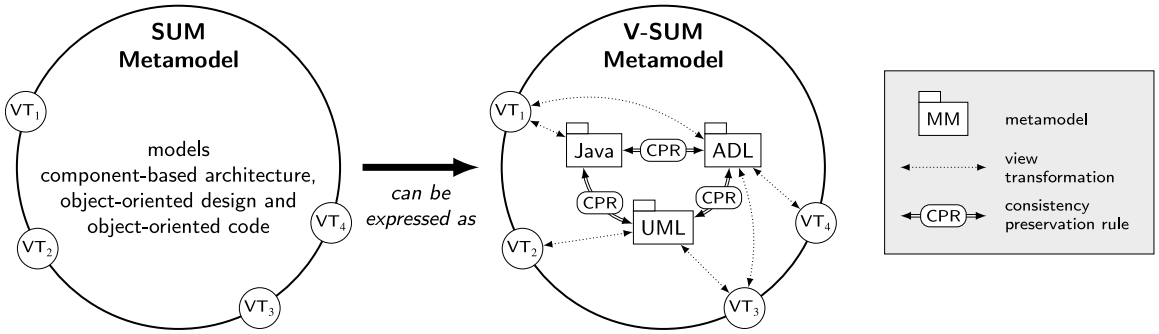


Figure 2.4.: Difference between a SUM meta-model and a V-SUM meta-model [67].

> "A SUM is a complete definition of a system and contains all known information about it. It contains no redundant or implicitly dependent information and is thus always free of [...] inconsistencies." [67]

In Vitruvius which is based on OSM, a virtual SUM (V-SUM) meta-model is realized which externally acts as a SUM meta-model [67]. Internally, the V-SUM meta-model is modularized and consists of several different EMOF-based meta-models allowing the reuse of existing meta-models. To ensure consistency between the different models in the V-SUM, a delta-based consistency preservation process is established using consistency preservation rules (CPRs) where a CPR is specified between two meta-models. As a consequence, if a change occurs modifying a model and causing inconsistencies with other models, CPRs update corresponding models keeping them consistent.

The difference between a SUM meta-model and V-SUM meta-model is shown in Figure 2.4 in the context of object-oriented source code, object-oriented design, and component-based software architectures [67]. The SUM meta-model combines all three concepts in one meta-model. In contrast, the V-SUM meta-model contains a Java meta-model, an UML meta-model, and an Architecture Description Language (ADL) meta-model with CPRs for each pair of meta-models. VT_1, VT_2, VT_3, and VT_4 define view types.

Developers perform their actual changes in views [67]. Therefore, one of two strategies has to be included for obtaining the deltas that are applied on the underlying models. In the first strategy, a change monitor observes the views and records all modifications so that the deltas are directly available and can be applied on the models in the V-SUM. The second strategy is a state-based model comparison (see section 2.5). The differences between the models without and with changes are investigated in order to receive a sequence of deltas that can be applied on the models.

Vitruvius offers the *Reactions language* and the *Mappings language* to formulate CPRs [67]. While it is possible to define declarative, bidirectional specifications within the Mappings language which are converted to specifications in the Reactions language, the Reactions language allows imperative and unidirectional specifications. A Reaction within the Reactions language consists of the following three steps: *Triggering - Matching - Actions*. A trigger defines when a Reaction is going to be executed. Therefore, it declares change types to which it reacts and an optional check expression testing properties of the change. The following atomic change types are differentiated:

"1. Replacements of a single attribute or reference value
2. List changes which affect a single list entry
3. Insertions and removals of root elements
4. Creations and deletions of model elements

[...] the change types 2 and 3 can be combined with change type 4." [67]

At last, a trigger specifies Reaction routines which are called if a change matches the trigger [67]. A Reaction routine separates the triggering step from the matching and actions steps by providing a match and action part. The match part contains match checks and retrievals for obtaining corresponding elements. A retrieval is either a presence retrieval or an absence retrieval specifying which elements have to be present or absent. Vitruvius stores correspondences, i. e., the connection between corresponding model elements, explicitly in a correspondence model which is queried for the retrievals. The action part lists statements that are executed to restore the consistency. Within the actions, model elements can be created, removed, or updated, correspondences can be registered or deregistered, and other Reaction routines can be called. Additionally, an API in the Reactions language allows interaction with the developer for cases in which the consistency cannot be preserved automatically.

Vitruvius is prototypically implemented in EMF supporting Ecore-based meta-models [67].

2.7. The Co-Evolution Approach

The co-evolution approach presented by Langhammer enables developers to keep source code and software architecture models consistent during development [71]. It is prototypically implemented in Vitruvius for Java using the original JaMoPP version and the PCM. Figure 2.5 depicts the co-evolution process: An architect or developer modifies their artifact in the corresponding editor (step 0). The changes are reflected in the editor's underlying representation (also step 0). Monitors observe these underlying representations (step 1) and trigger Vitruvius upon changes (step 2). How a monitor exactly reports the changes to Vitruvius depends on the editor. In the case of the PCM, the existing PCM editors are based on EMF so that they directly operate on the PCM instance. As a result, a generic EMF model monitor listens on the PCM instance using built-in EMF mechanisms that report changes. In the case of Java, the default Eclipse Java code editor is reused where the underlying representation is the JDT Core AST [56]. Therefore, the Java monitor classifies changes on the JDT Core AST and transforms them to changes on the JaMoPP model [71]. So, both monitors trigger Vitruvius through the modification of the EMF models in the V-SUM (step 2). Then, Vitruvius executes the consistency preservation process (step 3). This includes the retrieval of elements from the correspondence model (step 4) and the update of the correspondence model and the opposite model of the modified one (step 5).

The co-evolution approach knows three dimensions of CPRs: the technology-specific, the project-specific, and the element-specific dimension [71]. CPRs are specific for the used technologies, e. g., Plain Old Java Objects (POJOs) and PCM, and can be specific for

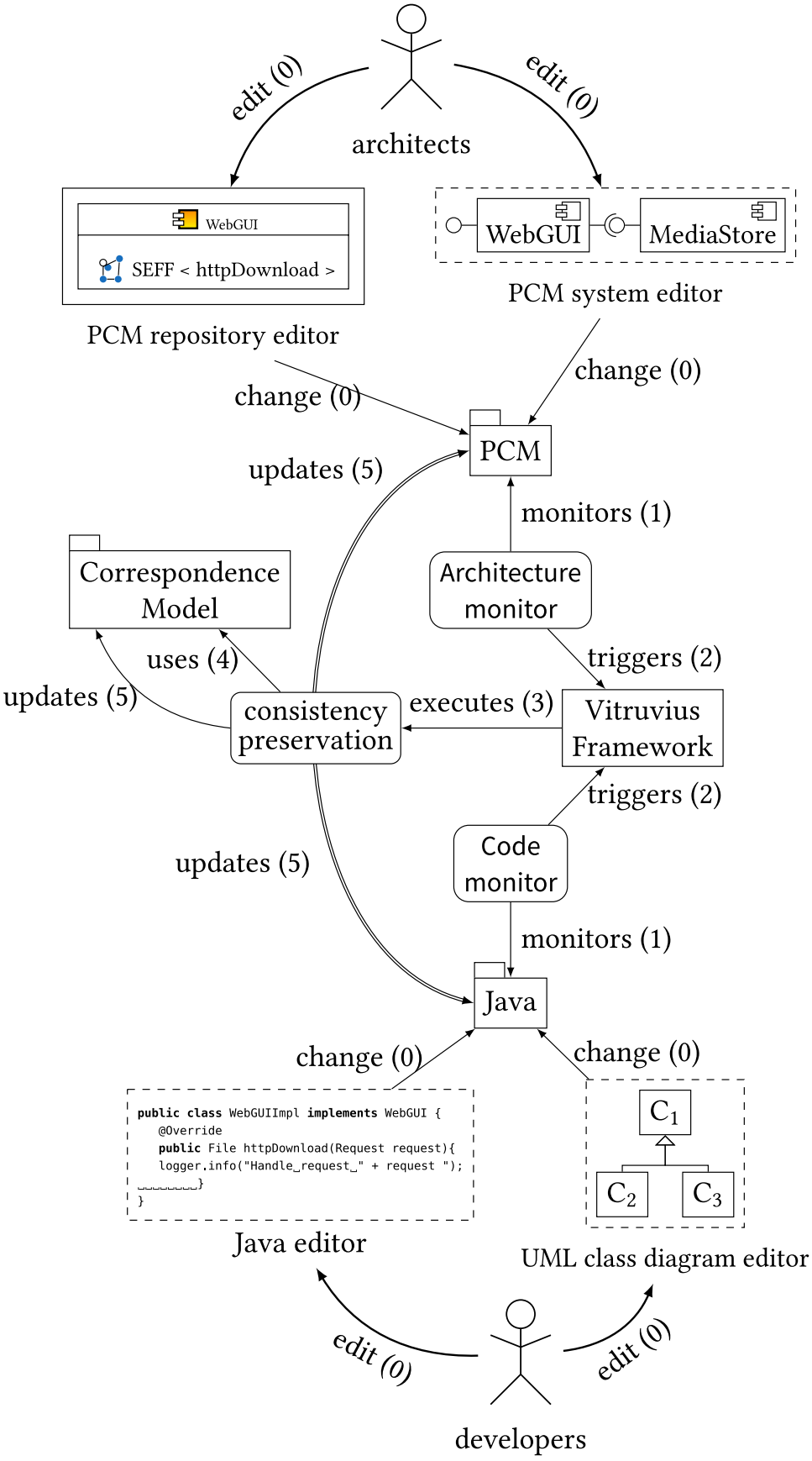


Figure 2.5.: The co-evolution process [71].

projects. The element-specific dimension is further divided into CPRs for a set of elements and CPRs for one element. The prototypical implementation contains CPRs for the PCM and POJOs called Package Mapping CPRs, the PCM and Enterprise Java Beans (EJB)-based source code, and the PCM and source code employing a dependency injection framework. In the Package Mapping CPRs, for example, a PCM repository is mapped to one package for all components, one package for all interfaces, and one package for all data types as listed in Table 2.1. In the component package, every component is represented by a package with the same name as the component and a class in the package realizing the component implementation. This class implements all provided interfaces of the component and contains a field for every required interface with the required interface as type. Further mappings and details can be found in [71]. All CPRs are bidirectionally defined in the Reactions language [67].

PCM metamodel element	**Source code language element**
Repository	Three *packages*: main, contracts, data types
BasicComponent	Package within the main *package* and a public component realisation *class* within the package
OperationInterface	*Interface* in the contracts package
Signature&Parameters	*Methods&parameters*
CompositeDatatype	*Class* with getter and setter for inner types
CollectionDatatypes	*Class* that inherits from a Java collection type (e.g. `ArrayList`)
RequiredRole	*Field* typed with required interface in the component-class and *constructor parameter* for the field in the component-class
ProvidedRole	Main class of providing component *implements* the provided interface
SEFF	*Method* in the component realisation *class* that overrides the corresponding interface method

Table 2.1.: Overview over the Package Mapping CPRs [71].

For the mapping between source code and SEFFs, special CPRs are defined [71]. If a SEFF is modified, it is unclear how code should be generated or changed because a SEFF is an abstraction of the code. Therefore, the prototypical implementation of the co-evolution approach generates a task for the developer and displays it in a task list. The developer can do the task and mark it as done afterwards. Modifications of method bodies cause a change-driven incremental SEFF reconstruction. This process is explained in detail in subsection 2.8.2. It is incorporated into the consistency preservation process as shown in Figure 2.6. After code changes (step 1), the notified monitor classifies the change (step 2). If it is unambiguous (step 3), the consistency preservation process is triggered resulting in the update of the component model (step 5). It the change is ambiguous (step 3'), the developer is asked to clarify its intent (step 4). The intent combined with the change triggers the consistency preservation process. If a method body has been changed (step 3"), the incremental SEFF creator runs generating an updated SEFF (step 4').

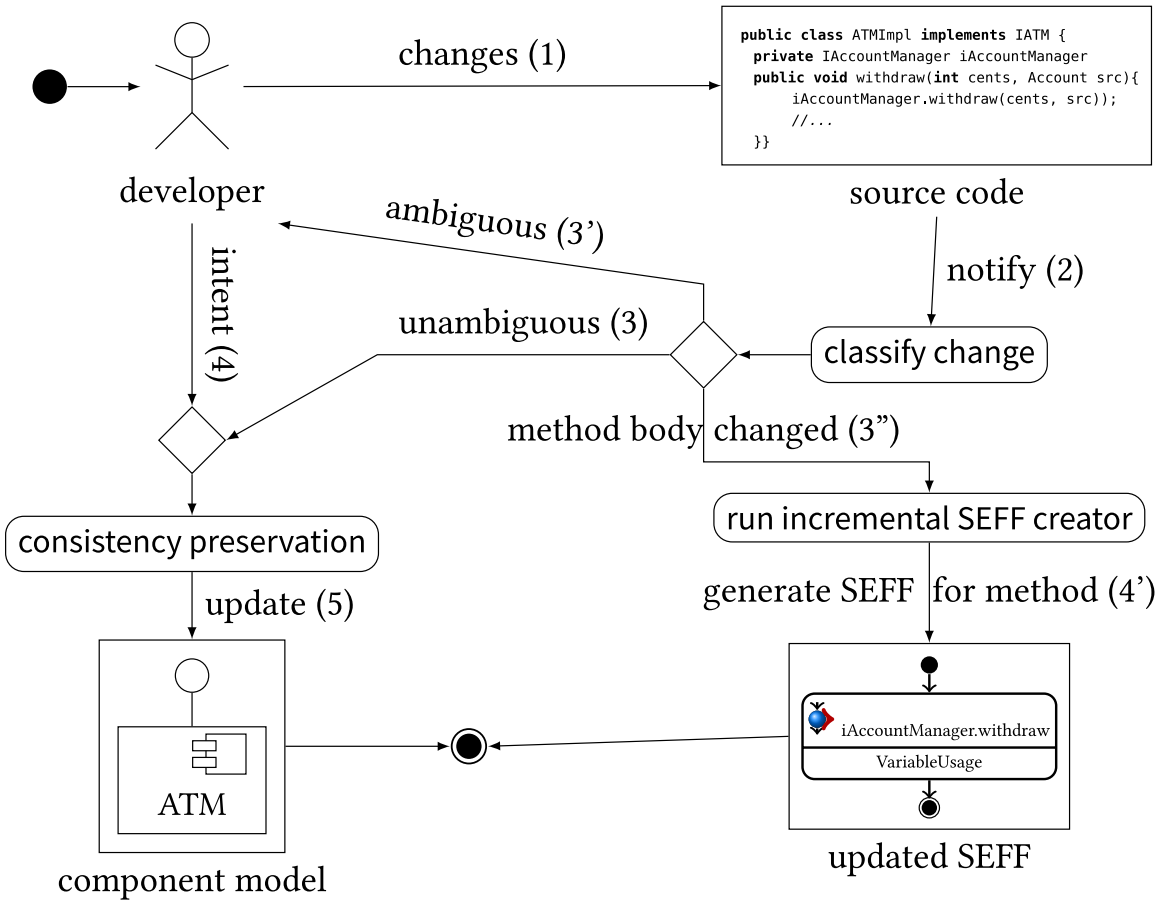


Figure 2.6.: The complete consistency preservation process for source code to PCM including SEFFs [71].

2.8. Service Effect Specifications

This section introduces the SEFFs. subsection 2.8.1 explains their structure while subsection 2.8.2 describes the process of incrementally reconstructing SEFFs from source code. As this process reconstructs complete SEFFs, an approach for the incremental fine-grained SEFF reconstruction is presented in subsection 2.8.3.

2.8.1. Structure of SEFFs

SEFFs, more precisely `ResourceDemandingServiceEffectSpecifications` (RDSEFF) as a specialization of SEFFs, abstractly model the behaviour of a service in a PCM component and act as a grey box view on the implementation [7]. A RDSEFF consists of `AbstractActions` which are depicted in an excerpt from the PCM meta-model in Figure 2.7. Every RDSEFF contains a `StartAction` and `StopAction`. Internal algorithms and details are summarized and represented in an `InternalAction` [7] or `InternalCallAction` [71]. An `InternalCallAction` is a call from an RDSEFF or `ResourceDemandingInternalBehaviour` to a `ResourceDemandingInternalBehaviour` which defines behaviour that is only available within a component [71]. Besides, calls to services of required interfaces are made explicit with `ExternalCallActions` [7]. Control flow involved with such external calls is modeled as an abstract control flow. While `ForkActions` express the concurrent execution of multiple branches, exactly one of multiple branches is executed in a `BranchAction`. Loops are represented by `AbstractLoopActions`. In addition, every action except `ExternalCallActions` can include resource demands [7] which are PMPs [73]. The demands can be expressed in the form of constants, probability distributions, or functions of random variables [7]. Moreover, resource demands are allowed to be parameterized specifying a dependency between the input parameters of the RDSEFF and the resource demand. Such parametric dependencies can also include branch conditions of `BranchActions`, loop iterations of `LoopActions`, and parametric parameter usages between the RDSEFF's input parameters and the input parameters of an `ExternalCallAction`. Throughout this thesis, the term *SEFF* refers to RDSEFFs as previously introduced.

2.8.2. Change-Driven Incremental SEFF Reconstruction

The change-driven incremental SEFF reconstruction is able to generate SEFFs from methods that are integrated within the co-evolution approach and that have changed [71]. Langhammer developed and presented the process in [71] as an extension of the static control flow analysis for reverse engineering SEFFs from source code presented by Krogmann in [70]. The basic principle remains [71]: At first, external calls are identified [70]. Beginning with external calls, relevant control flow statements are transitively marked. Finally, the SEFF control flow structure is created.

Three types of calls are differentiated [70, 71]: external calls as calls to methods in other components, internal calls as calls to methods within the current component, and infrastructure / library calls as calls to methods which are not contained in any component. In the first step of the SEFF reconstruction, all method calls in the changed method are classified as one of the three call types [71]. As the classification of external and library calls

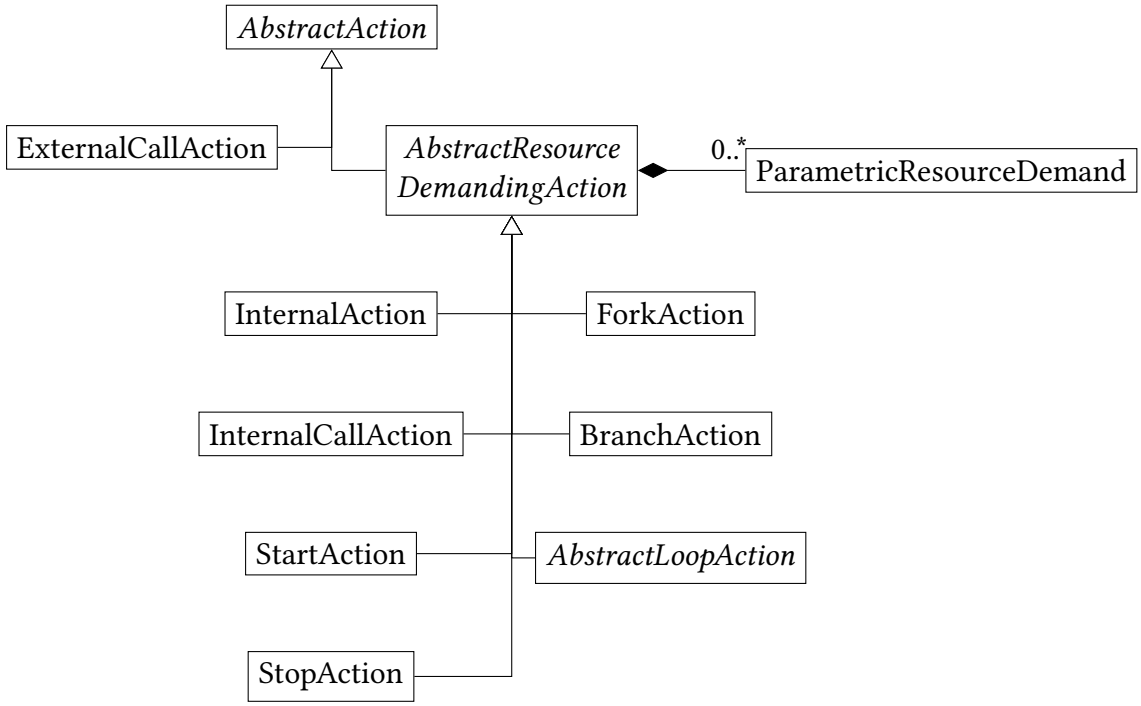


Figure 2.7.: Excerpt from the PCM meta-model showing the actions within RDSEFFs based on [7, 71].

depends on the used CPRs, a mapping-specific external call finder and a mapping-specific library call finder are used. All calls that are not an external or library call are internal calls.

The second step starts at external calls [70] or internal calls [71]. All statements from the call to the method declaration are visited and marked as relevant if the statement is a control flow statement [70]. During the third step, all marked control flow statements are converted to `AbstractActions`. Every SEFF gets a `StartAction` and `StopAction`. External calls are mapped to `ExternalCallActions`, conditional statements are mapped to `BranchActions`, loops are mapped to `AbstractLoopActions`, and remaining statements are mapped to `InternalActions` whereby no `InternalAction` follows another `InternalAction`. Internal calls are mapped to `InternalCallActions` and the called method is represented by a `ResourceDemandingInternalBehaviour` if it is a component-private method [71].

In order to find the signatures and `RequiredRoles` of external calls during the SEFF reconstruction, a mapping-specific `RequiredRole` finder needs to be defined [71].

The prototypical implementation of the co-evolution approach includes mapping-specific external call, library call, and `RequiredRole` finder for the implemented CPRs [71]. The Package Mapping CPRs, for instance, consider external calls as calls to interface methods with a corresponding signature in the PCM or as calls to methods in classes residing in a package corresponding to another component. Library calls are calls where the called method is inside a class whose package does not correspond to a component. More details on specific or further finders can be found in [71].

Internally, the change-driven incremental SEFF reconstruction establishes correspondences between the SEFF elements and the source code [21]. Dahmane added a further step after the actual reconstruction which stores the correspondences in the correspondence model of Vitruvius.

2.8.3. Incremental Fine-Grained SEFF Reconstruction

Dahmane proposes an approach for the incremental fine-grained SEFF reconstruction [21]. It allows the reuse of SEFFs by changing only the SEFF elements which correspond to changed code within a method. At first, all correspondences of the old SEFF elements are removed from the correspondence model. Afterwards, the change-driven incremental SEFF reconstruction is used to get a new SEFF for the changed method. The old and new SEFFs are compared to find matching elements. This includes a differentiation between equal and non-equal elements. Two SEFF elements are equal if their corresponding statements are equal. Else, the SEFF elements match, but are non-equal because their code has changed. Based on the matched elements, all elements in the old SEFF without a matching are identified as deleted. Analogously, all elements in the new SEFF without a matching are identified as added. Based on the difference between the old and new SEFF, the old SEFF is updated differentiating three cases. In the first case, all deleted elements are removed from the old SEFF. The second case considers non-equal matching elements. They are not changed. Added elements are added to the old SEFF in the third case. If the SEFF element has no predecessor with a matching element in the old SEFF, the SEFF element is added at the second position of the old SEFF. Otherwise, the SEFF element has a predecessor with a matching element in the old SEFF so that the SEFF element is added after the matching element. After the old SEFF has been updated, correspondences between the changed method and the updated old SEFF and between the SEFF elements and the statements are created.

The approach is prototypically implemented and evaluated [21]. `LoopActions` cannot be correctly compared as their corresponding statements cannot be correctly compared.

2.9. Continuous Integration of Performance Models

The CIPM approach updates performance models during development to keep them up-to-date [73]. The approach is introduced in more detail in subsection 2.9.1. subsection 2.9.2 handles the change analysis and propagation performed after every source code commit. After the Instrumentation Meta-Model (IMM) for the adaptive instrumentation is explained in subsection 2.9.3, the instrumentation process is described in subsection 2.9.4. Finally, a short overview over the monitoring of the instrumented source code, calibration, and self-validation using the monitoring data and simulation results is given in subsection 2.9.5.

2.9.1. Introduction to the CIPM Approach

The CIPM approach incrementally updates performance models on an architectural level and calibrates the PMPs after every source code commit in order to obtain up-to-date

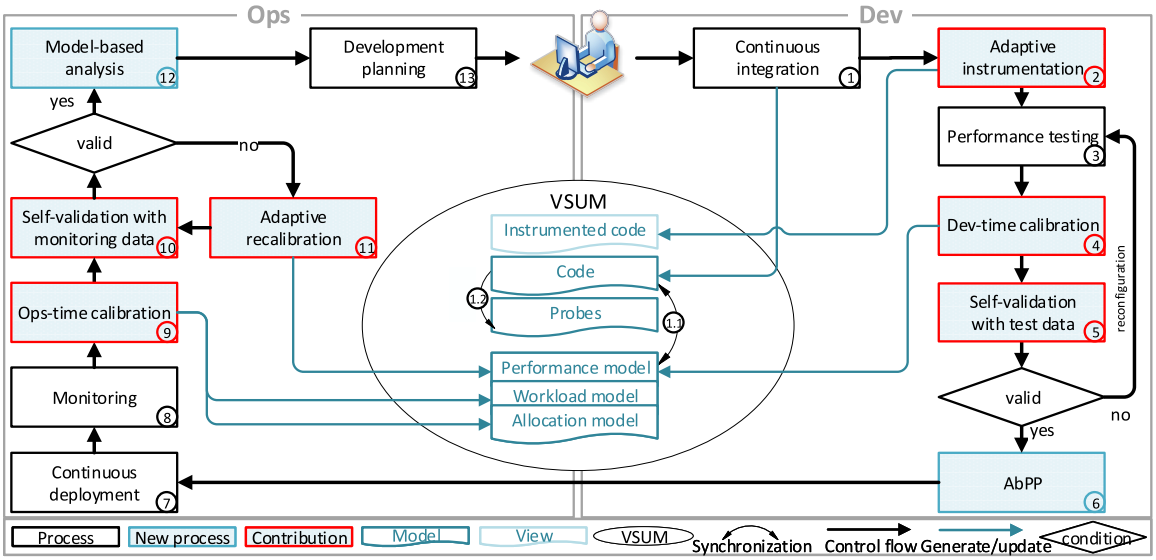


Figure 2.8.: The model-based DevOps pipeline within the CIPM approach [73].

performance models [73]. All activities of the CIPM approach are integrated into a model-based DevOps pipeline. Figure 2.8 displays the DevOps pipeline. The first part which takes place at development time (Dev-time) starts with the CI (1) in which the source code changes are incorporated into a code model. The realization of the CIPM approach is based on Vitruvius extending the co-evolution approach. Therefore, Java code and the PCM as performance model are used. The changes of the code model trigger the consistency preservation process updating the performance model (1.1) and an Instrumentation Model (IM) (1.2). Because the PCM instance contains no or outdated PMPs, for example, resource demands, after the update, the following activities concentrate on estimating the PMPs based on the monitoring of the code. In order to measure the required data, the code is adaptively instrumented with measurement instructions (2). The instrumentation points from the IM determine where the code is instrumented. It only includes code parts that have changed. After the instrumentation, the measurements are taken during the performance testing (3) and are divided into a training and validation set. The training set is used in the Dev-time calibration to estimate the PMPs (4). The validation set is used for a self-validation (5). In this activity, the performance model is simulated, and the error between the simluation result and the validation set is calculated. If the performance model is not accurate enough, the developer can reconfigure the performance test which is executed once more to recalibrate the performance model. If the performance model is accurate, the developer can perform Architecture-based Performance Predictions (AbPPs) (6).

The second part of the DevOps pipeline executed during the Operations time (Ops-time) begins with the Continuous Deployment of the code (7) [73]. The code is monitored in the production environment (8) to retrieve run-time measurements. During the following Ops-time calibration (9), the usage and allocation models are updated and calibrated based on the measurements. Afterwards, the performance model is self-validated (10) similar to the self-validation in the Dev-time part by using the run-time measurements.

If the performance model is not accurate enough, it is adaptively recalibrated (11). If the performance model is accurate, further model-based analyses can be performed (12) whose results can be used in the development planning (13).

2.9.2. Change Analysis and Propagation

Chupakhin prototypically implemented parts for the incremental model update [14]. The implementation extracts changes of a Git commit and applies them on a JDT Core AST. This JDT Core AST is observed by the Java monitor which updates the corresponding JaMoPP models in Vitruvius triggering the consistency preservation process for the PCM. The used CPRs are based on the Package Mapping CPRs from the co-evolution approach. Except the CPRs for method bodies, the CPRs are adapted for the CIPM approach including the addition of new CPRs for the removal of classes, interfaces, and packages. The implementation's evaluation tests all atomic changes which correctly update the JaMoPP models except in cases in which a compilation unit is deleted. In several cases, CPRs for updating the PCM are missing.

2.9.3. Instrumentation Meta-Model

Dahmane introduced the IMM with a prototypical implementation of CPRs between JaMoPP models and an IM [21]. The CPRs generate instrumentation points which identify code parts that have changed and will be instrumented during the adaptive instrumentation so that only the changed code is monitored. SEFFs are re-generated by the change-driven incremental SEFF reconstruction of the co-evolution approach (see subsection 2.8.2) and are completely replaced if their code has changed. As a result, all re-generated SEFFs are monitored independent of the extent of the changes in the corresponding code.

Monschein extended the IMM which is depicted in Figure 2.9 [75]. The `InstrumentationModel` acts as the root element of an extended IM. It contains several `ServiceInstrumentationPoints` representing a SEFF for the instrumentation and pointing directly to the SEFF. Every `ServiceInstrumentationPoints` contains further `ActionInstrumentationPoints` representing and pointing to `AbstractActions` within the SEFF. An `ActionInstrumentationPoint` allows the monitoring of `InternalActions`, `AbstractLoopActions`, and `BranchActions`. The actual type of the `AbstractAction` is attached to the `ActionInstrumentationPoint`. All `InstrumentationPoints` have an `active` variable controlling if the monitoring of an instrumentation point is activated or deactivated.

2.9.4. Adaptive Instrumentation

There are two prototypical implementations for the adaptive instrumentation. Dahmane implemented it based on the original JaMoPP version and the IM [21]. Beside the source code and IM, the implementation takes the correspondence model of Vitruvius as input. Afterwards, the instrumentation process copies the source code which is parsed with the original JaMoPP version. Then, the instrumentation points are mapped to their corresponding statements in the copied source code. The search utilizes the location of the original statements, the service containing the statements, and the class containing

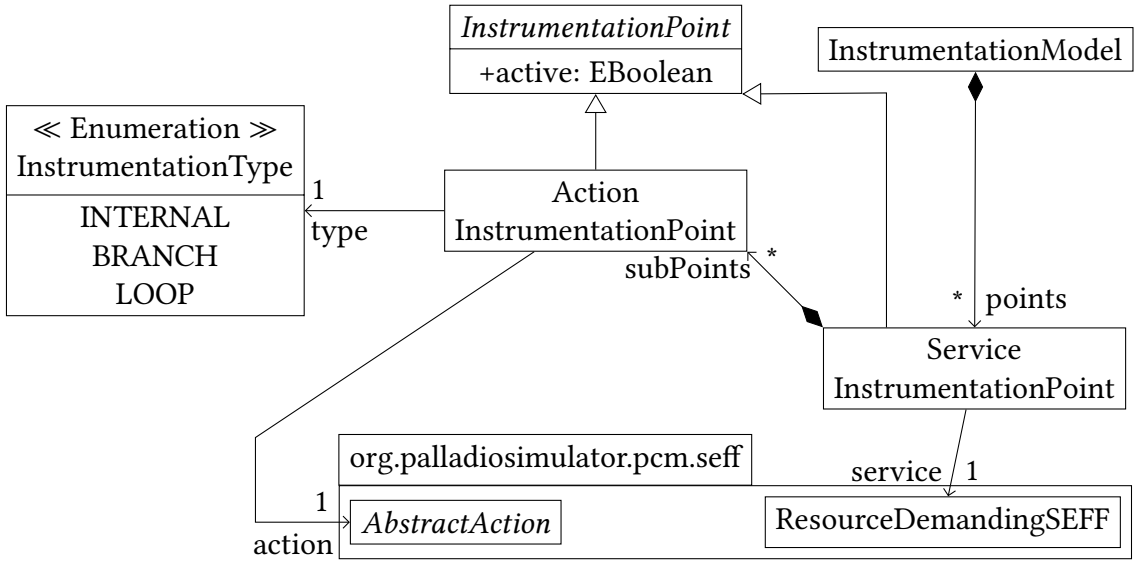


Figure 2.9.: The extended IMM [75].

the service to speed up the statement finding. All found statements are instrumented based on the type of instrumentation point. It differentiates between fine-grained and coarse-grained instrumentation where coarse-grained instrumentation means that a SEFF as a whole unit is instrumented and fine-grained instrumentation generates statements for the monitoring of specific `AbstractActions`. After all instrumentation points have been instrumented, all non-instrumented SEFFs are instrumented coarse-grained because they can be called by the changed code parts. Algorithm 1 summarizes the instrumentation process in pseudo-code.

Algorithm 1 Source Code Instrumentation based on [21] and [12].

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

1: SCC ← copySourceCode(SC)
2: SCCmodel ← parse(SCC)
3: statements: (InstrumentationPoint → Statements) ← findStatements(IM, CM, SCCmodel)
4: instrumentSourceCode(statements, CM)
5: instrumentCoarseGrained(statements, CM, SCCmodel)

The second implementation by Monschein uses the extended IM and the JavaParser [75] which offers a parser and AST for Java from version 1 to version 15 with a non-Ecore-based meta-model [60]. The instrumentation process takes the extended IM, the correspondence model of Vitruvius, and the source code as input [75]. As a result, the adaptive instrumentation does not copy the source code and modifies it directly instead. For every `ServiceInstrumentationPoint`, the SEFF is instrumented coarse-grained. If the `ServiceInstrumentationPoint` includes `ActionInstrumentationPoints`, these instru-

mentation points are instrumented fine-grained. The instrumentation process is depicted in Algorithm 2 in pseudo-code.

Algorithm 2 Source Code Instrumentation from [75].

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

```
1: for all SIM ∈ IM.serviceInstrumentationPoints do
2:     sourceCodeElements ← getSourceCodeElements(SIM.service, CM)
3:     instrumentServiceCall(sourceCodeElements)
4:     for all AIP ∈ SIM.actionInstrumentationPoints do
5:         actionSourceCodeElements ← getSourceCodeElements(AIP.action, CM)
6:         instrumentAbstractAction(sourceCodeElements, AIP.type)
7:     end for
8: end for
```

2.9.5. Monitoring, Calibration, and Self-Validation

The previous work in [73] and [104] consider the estimation of PMPs. Monschein extended the calibration and self-validation with a Validation Feedback Loop and Transformation Pipeline [75]. While the Validation Feedback Loop incorporates the self-validation results into the next validation run, the Transformation Pipeline is responsible for updating the PCM instance based on self-validations. Moreover, Monschein's extensions are combined with the monitoring, calibration, and self-validation in a pipeline which allows its automatic execution during Dev-time and Ops-time [74].

2.10. Integration of Existing Source Code into Vitruvius

The co-evolution approach and the CIPM approach assume that Vitruvius has been used since the beginning of the development [71, 73]. However, this assumption does not hold in the real world. Therefore, several approaches have been proposed to integrate existing source code into Vitruvius [70, 71]. The approaches have in common that they reverse engineer a PCM instance from the source code at first.

> "Reverse engineering is the process of analyzing a subject system to
> - identify the system's components and their interrelationships and
> - create representations of the system in another form or at a higher level of abstraction." [13]

Afterwards, the code model and PCM instance are added to the V-SUM while establishing correspondences between source code elements and PCM elements in the correspondence model of Vitruvius [71].

3. Approach

This chapter presents an overview over the approach of this thesis following the **PRICoBE** principle [86]. The **P**roblems are introduced in section 3.1. section 3.2 describes the arising **R**esearch questions followed by the **I**dea in section 3.3 to solve the problems and answer the research questions. The idea's detailed realization is illustrated in chapter 4. The resulting **Co**ntributions and **B**enefits are explained in section 3.4 and section 3.5, respectively. At last, the results of the performed **E**valuation are shown in chapter 5.

3.1. Problem

The presented approach is embedded in the context of the CIPM approach and Microservice-based applications. As outlined in section 2.9, parts of the Dev-time part are prototypically implemented without forming a complete pipeline [14, 21, 75] and partly evaluated with artificial projects [14, 21]. Moreover, the implementations are based on the original JaMoPP version [14, 21] which only supports Java code conforming to the JLS 3 [48]. This leads to the following central problem statement.

Central Problem Statement The current prototypical implementations for parts of the CIPM approach cannot be used for the automatic execution of the Dev-time part at once and are not evaluated in combination regarding their feasibility for real world applications (**P0**). Furthermore, they do not support Java source code conforming to the JLS 7 or a later version (**P1**).

Problems Regarding the Commit-Based Integration Strategies As mentioned in section 2.6, there are two strategies for deriving changes for the consistency preservation process in Vitruvius [67]. These strategies are the delta-based and state-based approaches. In the context of the CIPM approach, they are extended to extract changes from a commit and to propagate them [14]. Hence, both strategies are called commit-based integration strategies.

The delta-based change propagation realized by the Java monitor is not fully evaluated [14] and the state-based change propagation is not evaluated (**P2**).

There are approaches for integrating existing source code into Vitruvius [71, 70]. However, they are not suitable for the CIPM approach (**P2.1**). As a consequence, the commit-based integration strategies are adapted for the integration.

Problems Regarding the CPRs The Package Mapping CPRs of the co-evolution approach were adapted for the CIPM approach, but their evaluation indicates that they require further adjustments (**P3.0**) [14]. In addition, they are not designed for Microservice-based applications [71] so that they need a strategy for discovering components in this context

(**P3.1**). Furthermore, the CPRs for the IM are not compatible with the extended IM (**P3.2**) [21]. Therefore, the problem **P3** considers the absence of complete CPRs for the CIPM approach.

Dahmane proposed an approach for the incremental fine-grained SEFF reconstruction, prototypically implemented it with respect to the IM, and evaluated it [21]. Concerning the extended IM and the evaluation results, the prototypical implementation requires an adaptation and extension (**P3.2.1**).

Problems Regarding the Adative Instrumentation **P3.2** and **P3.2.1** show that the extended IM is only partially adapted in the prototypical implementations [21, 75]. Beside the CPRs, the prototypical implementation of the adaptive instrumentation by Dahmane is based on the IM and the original JaMoPP version [21] while Monschein's prototypical implementation is based on the extended IM and the JavaParser [75]. This means that there is no prototypical implementation for the adaptive instrumentation which is based on the extended IM and JaMoPP (**P4**).

Aim Considering all problems, the main aim of the approach is the evaluation of the Dev-time part of the CIPM approach with a case study as the representation of a real world application.

3.2. Research Questions

The previously described problems lead to the following research questions.

R0 How do the prototypical implementations for the first step (incremental model updates and adaptive instrumentation) of the CIPM approach have to be adjusted to be applicable in combination to real world applications?

R1 How can newer versions of Java be supported?

R2 Which commit-based integration strategy is suitable for the CIPM approach?

Concerning the commit-based integration strategies, **R2** represents the central research question containing and summarizing several further aspects. As a consequence, the following questions arise covering these aspects.

- How do the commit-based integration strategies have to be extended to be able to update the Java models correctly?
- How can the commit-based integration strategies support arbitrary commits?
- How well do the commit-based integration strategies perform compared to each other?

R3 How can the CPRs be reused and adapted for updating the PCM and extended IM?

R4 How does the adaptive instrumentation have to be adjusted in the context of the extended IM and Java models?

3.3. Idea

The basic idea of the approach is to complete the prototypical implementation of the Dev-time part of the CIPM approach to support CI and evaluate it with a case study. For integrating existing source code, the commit-based integration strategies are extended so that a specific commit is integrated as the initial commit. In order to support newer Java versions within the prototypical implementations, the new features of Java are incorporated into the code models.

3.4. Contributions

Based on the idea, the following contributions are made.

C0 Evaluation of the Dev-time part of the CIPM approach for real world usage

C1 Support for newer versions of Java

C2 Extension, evaluation, and comparison of the commit-based integration strategies

C2.1 Adaptation and evaluation of commit-based integration strategies for the integration of existing source code

C3 Adaptation, extension, and evaluation of CPRs for the CIPM approach

C3.0 Adaptation, extension, and evaluation of the CPRs of the co-evolution approach for the PCM

C3.1 Extension and evaluation of a component discovery strategy for the PCM update and case study

C3.2 Extension and evaluation of CPRs for the extended IM

C3.2.1 Extension and evaluation of the prototypical implementation of the proposed approach for the incremental fine-grained SEFF reconstruction

C4 Extension and evaluation of the prototypical implementation of the adaptive instrumentation

3.5. Benefits

Based on the idea, the following benefits are achieved.

B0 Automatic updates to keep models up-to-date for, e. g., performance predictions

B1 Improved usability of CIPM by supporting newer Java versions

B2 Integration of CIPM with CI for improved usability

B2.1 Existing source code can be used with the CIPM approach

B3 Reduction of the overhead for updating the PCM and extended IM

B3.0 Correct update of the PCM

B3.1 Abstraction from source code

B3.2 Correct instrumentation points are set

B3.2.1 Reduction of the monitoring overhead

B4 Reduction of the overhead for the source code instrumentation by automatizing the process

4. The Commit-Based CIPM Approach

This chapter explains the approach in detail. Based on the further development of JaMoPP described in section 4.1, the change propagation is introduced in section 4.2. Afterwards, the CPRs for the PCM and extended IM are presented in section 4.4 and section 4.5, respectively. At last, the adaptive instrumentation is covered in section 4.6.

Parts of the approach are oriented on the TeaStore which is a web-based store for tea and related products [65] and the selected case study for the evaluation (see section 5.3).

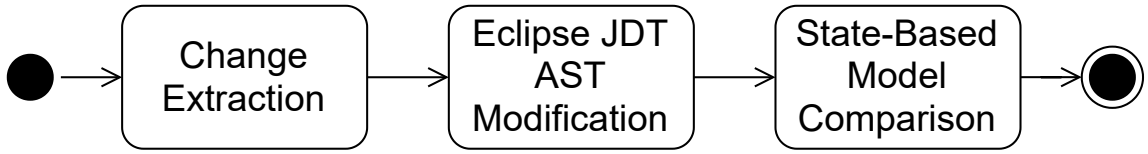
4.1. Development of JaMoPP

The Java monitor requires the capability of JaMoPP to parse single files [103]. However, the adapted JaMoPP version assumes that it parses the complete source code with its dependencies. Therefore, the adapted JaMoPP version was extended to support single file parsing. Instead of setting the references directly, the converter between the JDT Core AST and JaMoPP model creates proxy objects which are resolved to the actual model elements on demand [52]. To find these elements, the reference resolution mechanism of the original JaMoPP version was re-introduced and extended to cover the new features of the meta-model. Furthermore, the parsing process was enhanced with a binding-based resolution in which proxy objects are directly resolved after the parsing using the generated bindings. As a result, the implementation of the binding-based resolution offers a combination of the principles of the original and adapted JaMoPP versions. The aforementioned extended version of JaMoPP is referred to as the *updated JaMoPP version* in the remainder of this thesis.

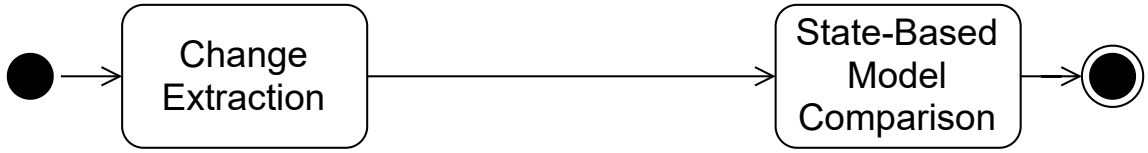
4.2. Updating the Java Model

Since the change extraction and propagation were prototypically implemented by Chupakhin [14], the Java monitor has been developed further replacing the delta-based change propagation with the state-based change propagation [66]. As a result, the updated procedure including the Java monitor is shown in Figure 4.1 in comparison with the state-based change propagation. As displayed, the difference between both procedures is an additional step in which the JDT Core AST is modified. Therefore, instead of employing the Java monitor, the state-based change propagation is directly applied.

In Figure 4.2, the process for extracting a commit's changes is displayed. In a local copy of the repository, a specific commit, usually the latest one, is checked out if there are changes in Java files. Otherwise, the propagation stops because there are no changes to propagate. After the checkout, the project is build to ensure that the instrumented code can



(a) Change extraction and propagation with the updated Java monitor.



(b) Change extraction and propagation by directly applying the state-based change propagation.

Figure 4.1.: Comparison of the change propagation with the updated Java monitor and state-based change propagation.

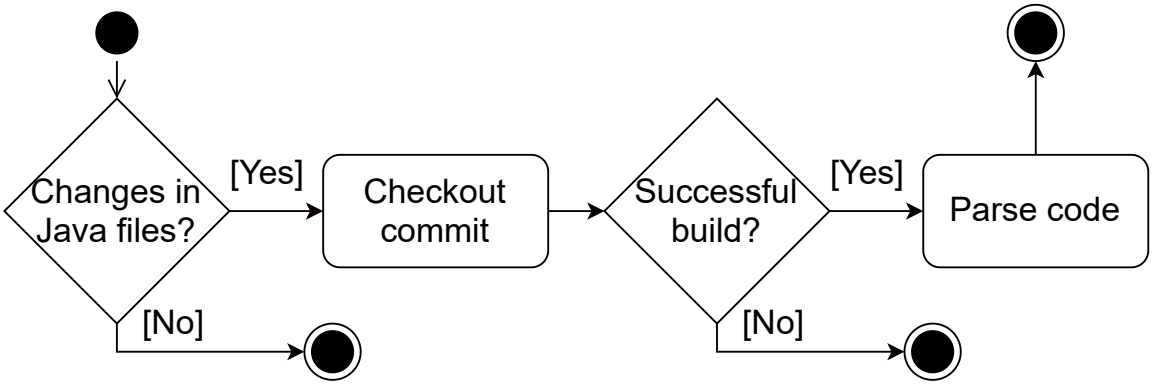


Figure 4.2.: Detailed and updated change extraction.

be build and to collect the dependencies. If the build fails, the propagation ends. In contrast, if the build succeeds, the source code is parsed with JaMoPP to generate a complete Java model including the dependencies. In the resulting model, every compilation unit, package, and module element is contained within a separate EMF `Resource`. Technically, only one `Resource` can be propagated to the V-SUM at once [102]. The references between Java models lead to dependencies between the `Resources`. Therefore, they need to be propagated in a order which respects the dependencies and avoids the usage of elements before they are propagated to the V-SUM. However, some `Resources` can include dependency cycles so that they still need to be propagated at once which is not possible. Hence, all root objects are combined into one `Resource` representing the Java code model which is propagated to the V-SUM. Additionally, it includes the models for the source code dependencies to also propagate their changes if, e. g., a dependency is upgraded.

Without modification, the default state-based change resolution strategy would be used in Vitruvius to create a fine-grained change sequence for the code model based on EMF Compare [102, 103]. It takes two models as input for their comparison [24] which are the Java model in the V-SUM and the Java model of the newly parsed commit in this case. Then, EMF Compare performs the matching of the elements to find equivalent model

objects. However, the default matching cannot correctly relate Java model elements to each other due to a lack of knowledge about Java-specific properties of the models. Therefore, the strategy was adjusted to utilize parts of SPLevo. Instead of the default matching algorithm, the adjusted strategy uses the hierarchical matching algorithm of SPLevo in combination with the `SimilarityChecker`. This class is extended to support the updated JaMoPP version and modules in particular. After the Java-specific matching, the default differencing and merge are performed. While the detected differences are merged with the code model in the V-SUM, the change recorder of Vitruvius records the changes [102]. Because EMF Compare ensures the model integrity during the merge [24], the recorded changes have the same property. As a result, a model element is always created before it is set as, e. g., the target of a reference. At last, the created Vitruvius changes are propagated in the V-SUM [102].

By propagating fine-grained changes, specific coarse-grained changes such as the update of a method are not generated and propagated. If, for instance, a statement is added in a try block inside of a method, the creation and addition of the statement are reported, but not on the level of the method which is required by some CPRs at a later point in time. Hence, a post-processor for the comparison of EMF Compare is installed in which changed methods are identified. After the fine-grained change sequence has been created, the name of a changed method is set to an empty string and back to its actual name to generate additional Vitruvius changes that are automatically appended to the existing change sequence and allow the detection of changed methods in CPRs. By appending the new changes, it is ensured that these additional changes do not infer with the existing changes.

In the complete process of updating the Java model, a specific target commit is checked out. It is independent of which commit was propagated before allowing a developer, e. g., to skip commits. In the context of the integration of existing source code, the Java model for the initial commit is compared to an empty model representing an empty repository. As a result, an arbitrary commit can serve as the initial commit.

4.3. Discovering Components

Within the CPRs for the PCM, the first necessary step is the creation of components requiring a component discovery strategy. In the Package Mapping CPRs of the co-evolution approach, a clear organization of packages is given and required [71]. However, in general, package structures vary. Thus, components need to be detected differently.

The strategy described in the following focuses on Microservice-based applications. As a consequence, before the strategy is explained, the characteristics of components in the PCM are compared with the definition of Microservices by Fowler and Lewis to clarify what a component is in a Microservice architecture. As mentioned in section 2.2, in the definition of Microservices, Fowler and Lewis view Microservices as components [31] so that the characterizations of components and Microservices by Fowler and Lewis are compared to the context of the PCM. They have in common that a component is a software unit [31, 7]. Furthermore, a component can be independently deployed in the context of the PCM [7] enabling its independent replacement and upgrade conforming to the component

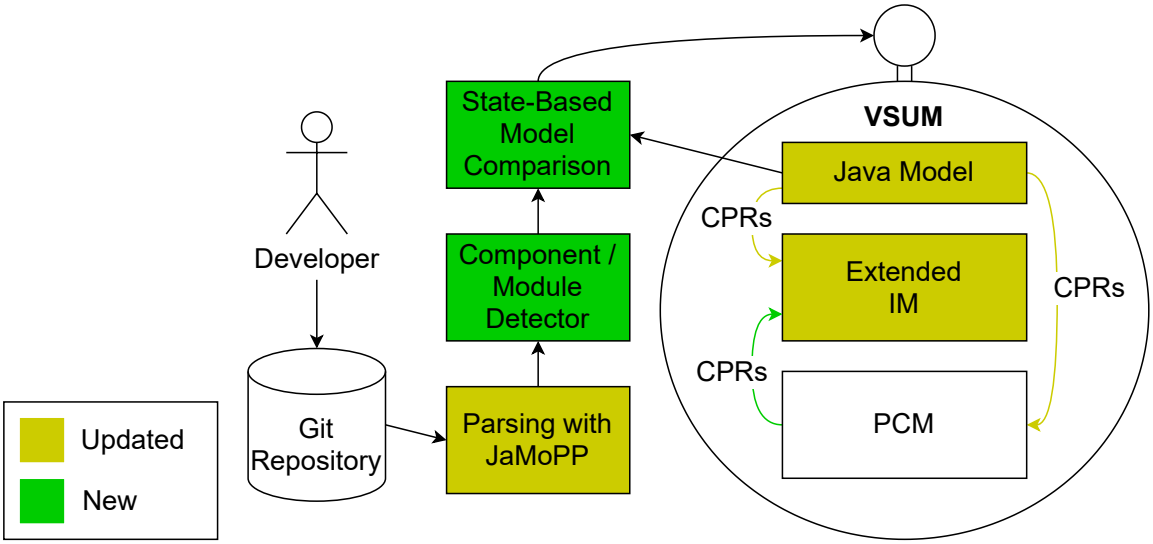


Figure 4.3.: Model update with updated and new parts.

definition of Fowler and Lewis [31]. Microservices expose an explicit API and have explicit dependencies as components in the PCM [7]. To summarize, Microservices exhibit the same characteristics as components in the PCM. Therefore, Microservices are components, and the main goal of the component discovery strategy is to find Microservices. Nevertheless, regular components can be still included in the source code.

During the change propagation, a CPR reacts to exactly one change. Thus, it cannot consider the complete source code structure. Moreover, the structure is only partly available because the change propagation is ongoing. In the CPRs, components cannot be detected for this reasons, and a new step was added before the changes are propagated. As shown in Figure 4.3, the *Component / Module Detector* realizing the component discovery and allowing the consideration of the complete source code and additional files in the repository is located after the parsing and before the state-based model comparison.

The developed component discovery strategy depicted in Figure 4.4 is oriented on the TeaStore and its organization and structure. In the TeaStore, every Microservice is contained within its own directory wich corresponds to a module of the employed Maven build tool [99]. As a consequence, Maven modules and build projects in general are considered for components by looking at the configuration files of build tools, e. g., a `pom.xml` for Maven [100] or `build.gradle` for Gradle [42]. While the Component / Module Detector checks for every compilation unit in the code model originating from the source code its relation to a component, the files in the repository are investigated. If the Java file corresponding to the compilation unit model is contained within a directory which includes only a POM file, the Maven module is identified as a component candidate because it can describe a Microservice in development without a deployment configuration. If the directory includes a `Dockerfile` which is a deployment configuration file [85] in addition to a POM file, the Maven module is assumed to be a Microservice. After a component and component type has been found, the Component / Module Detector collects the classes of the considered compilation unit in a set representing the Maven module and component.

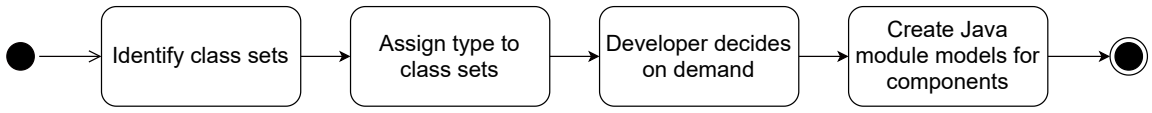


Figure 4.4.: Process of the component discovery strategy.

By iterating over all root model elements and because every significant compilation unit is contained in exactly one Maven module, the Component / Module Detector creates disjoint sets of classes.

Based on the previous results, for component candidates, the developer is asked to decide which actual type the Maven module has. Available options are Microservice component, regular component, part of another component, and no component. Depending on the decision, the set of classes is differently handled. For a Microservice or regular component, an actual component in the PCM will be created. If the component candidate is part of another component, the developer has also to decide to which component the set of classes belongs. In contrast, if the classes correspond to no component, they are ignored. The decisions of the developer are stored and loaded to reduce the frequency of required interactions. At last, the options for component candidates can be extended to allow, e. g., multiple components so that the set of classes is split into multiple sets. In this case, other techniques are needed to find reasonable partitions.

After all actual components have been selected, a mechanism is still required to identify the components in the CPRs during the change propagation. With the introduction of modules in Java 9 [40] and their availability in the updated JaMoPP version, modules are considered as an appropriate means for representing a component in the code model. Therefore, a module model object is created for every component and assgined to the classes of the component. In order to avoid inferences with existing modules, all modules are removed before components are discovered.

4.4. The CPRs for the PCM

This section introduces the CPRs for the PCM implemented in the Reactions language.

Based on the component discovery strategy, the CPRs for the components were adjusted. If the CPRs encounter a class within a module and the module for the first time, a component is created for the module. Afterwards, a correspondence between the module and the component and between the class and the component is created. In case of a class within a module for which a component exists, only a correspondence between the class and the component is added. On the other hand, a component is removed if the corresponding module is deleted or if the last class corresponding to the component is deleted.

Based on the origin of a module, Microservice and regular components are distinguished although there is no explicit differentiation in the PCM.

Beside the component detection, the interface detection is another important step in the CPRs. It relies on the type of a component. For regular components, their public classes are modeled as interfaces. For Microservices, the TeaStore defines REST APIs [65] so that the interface detection concentrates on identifying classes realizing a REST API. In its

concrete implementation, the TeaStore builds upon the *Java Platform, Enterprise Edition (Java EE) Specification, v7* and especially uses the *Java Servlet Specification Version 3.1* and *Java API for RESTful Web Services (JAX-RS) Version 2.0* for implementing the REST APIs [23, 99].

JAX-RS is designed for REST APIs in which a class can be turned into a REST API with different annotations [83]. Every class annotated with `Path` or `ApplicationPath` is an API class. Additionally, if a class contains a method annotated with `Path` or a request method designator which is an annotation that is annotated with `HttpMethod`, the class also realizes a REST API. In the CPRs for the PCM, all cases are checked for a class. If one case is true, an PCM interface is created for the class. Nonetheless, there is a limitation in the CPRs. They do not consider the dynamic addition or removal of annotations so that they discover interfaces only for already annotated classes and delete the interface when the class is removed.

In the Servlet specification, a `HttpServlet` as a specialization of a `Servlet` provides protected methods for the default HTTP methods, e. g., `doGet`, to implement the handling of HTTP requests with the corresponding methods [11, 54]. Overriding the methods, a REST API can be implemented in the subclass of a `HttpServlet`. As a result, every class which inherits from `HttpServlet` is coverted to a PCM interface. Additionally, the superclass `GenericServlet` of the `HttpServlet` [54] and the interface `Servlet` which is implemented by the `GenericServlet` [53] are also represented as interfaces in the PCM to recreate the class hierarchy of servlets as an interface hierarchy.

Both previously presented specifications evolved into the *Java Servlet Specification Version 4.0* [10] and *JAX-RS Version 2.1* [9] and were renamed to *Jakarta Servlet 4.0* and *Jakarta RESTful Web Services 2.1* [58]. Furthermore, the renamed specifications developed further to the *Jakarta Servlet Specification 5.0* [59] and *Jakarta RESTful Web Servcies 3.0* [18]. In the course of this specification releases, no major changes occurred to the underlying principles [10, 59, 9, 18, 58] on which the CPRs are based. Therefore, the CPRs can be used in the context of all aforementioned versions of the specifications.

After the PCM interfaces have been detected, provided and required interfaces can be identified. An interface is provided by a component if the PCM interface corresponds to a class or is implemented by a class. In the case of required interfaces, similar to the co-evolution approach, an interface from a component is required by another component if there is a field in this component with a type corresponding to the interface or if the type corresponding to the interface is imported by a type in this component.

Within classes which are modeled as PCM interfaces, all public methods are converted to `OperationSignatures`. For all `OrdinaryParameters` which are added in such methods, a PCM parameter is created. The type of the `OrdinaryParameter` determines the PCM data type for the parameter by searching a corresponding data type at first. If none can be found, a new data type is created based on the strategy of the co-evolution approach. As a consequence, array types and subtypes of the `Map` or `Collection` interface with explicit type arguments are modeled as `CollectionDataTypes` where the array's element type or the explicit type argument is utilized as the collected data type so that the data type creation is recursively applied on the type. Primitive types are represented as `PrimitiveDataTypes`. In other cases, a `CompositeDataType` is generated. For all fields of an source code type, an `InnerDeclaration` is created and added to the `CompositeDataType`. The data type for the

InnerDeclaration is also recursively created with the field type. However, the recursive application of the data type creation on InnerDeclarations can lead to a large number of data types including a modeling of private fields within classes of the Java standard library or dependencies. Therefore, InnerDeclarations are only created for the parsed source code. It limits the number of data types and hides implementation details of classes outside of the observed source code while the details of the source code are covered.

A special case for methods corresponding to OperationSignatures is the reduction of their visibility. If, e. g., a public method is changed to a private method, the OperationSignature is removed because the method is considered to be non-architectural-relevant after the change.

If a concrete class method corresponds to an OperationSignature or is the implementation of a method corresponding to an OperatonSignature, a SEFF is created for the method. As outlined in section 4.2, the propagated change sequence contains name changes for all changed methods. Therefore, if the name of a method is set to a valid string, i. e., not null and not an empty string, and the method has a corresponding SEFF, the SEFF reconstruction is executed. Moreover, in case that ResourceDemandingInternalBehaviours are generated during the incremental SEFF reconstruction, they are added to the SEFF. For the incremental fine-grained SEFF reconstruction, ResourceDemandingInternalBehaviours are handled differently. After the old and new SEFFs have been merged, the ResourceDemandingInternalBehaviours in the old SEFF are replaced by the ResourceDemandingInternalBehaviours of the new SEFF. Additionally, it is checked that every InternalCallAction points to one of the newly created ResourceDemandingInternalBehaviours. If there is an InternalCallAction which references an removed ResourceDemandingInternalBehaviour, the reference is updated to the corresponding new ResourceDemandingInternalBehaviour.

The CPRs for the PCM are summarized in Table 4.1. It is assumed that there is only one Repository corresponding to the code model so that it is created once and has no corresponding Java element.

PCM meta-model element	**Java element**
Repository	-
BasicComponent	At least one *class* in a *module*
OperationInterface	*Class* annotated with Path or ApplicationPath, *class* containing a method annotated with Path or a request method designator, *subclass* of HttpServlet, HttpServlet, GenericServlet, Servlet, public *class*
OperationSignature & Parameters	Public *methods* & OrdinaryParameters
CompositeDataType	*Type* of a *parameter* which is not a PrimitiveDataType or CollectionDataType
CollectionDataType	*Subtype* of Map or Collection interface with explicit type argument, *array*
RequiredRole	*Field* typed with an OperationInterface from another component, *import* of OperationInterface from another component

PCM meta-model element	Java element
ProvidedRole	*Class implementing* an `OperationInterface`, *class* corresponding to an `OperationInterface`
SEFF	*Method* with a corresponding `OperationSignature`

Table 4.1.: Overview over the implemented CPRs for the PCM.

4.5. The CPRs for the Extended IM

The CPRs between the Java model and the IM were updated to support the updated JaMoPP version and the extended IM. If the simple transformation is notified about a changed method, it looks up the corresponding SEFF and service instrumentation point. If there is no service instrumentation point, a new one is created and added to the extended IM. In case of an existing service instrumentation point, all action instrumentation points are removed. After an empty service instrumentation point has been obtained, a new action instrumentation point is created for every action in the SEFF.

A temporal constraint for these CPRs was identified: the instrumentation points can only be created after the SEFF has been created and reconstructed. Otherwise, there are no SEFF or SEFF actions during the execution of the CPRs, and the extended IM contains no instrumentation points after the change propagation. To ensure that this temporal constraint is met, the CPRs between Java and the extended IM extend the incremental SEFF reconstruction so that they are always directly executed after the incremental SEFF reconstruction.

An alternative to mitigate the cohesion between the CPRs introduced by the temporal constraint is the usage of CPRs between the PCM and extended IM which were also defined. The CPRs are implemented in the Reactions language and create or delete an instrumentation point if a SEFF or SEFF action is created or deleted. As a consequence, if the SEFF reconstruction leads to changes in the PCM, they are transitively propagated to the extended IM. Internally, correpsondences between the SEFF or SEFF action and its instrumentation point are established to find, e. g., the corresponding service instrumentation point to a SEFF and to add an action instrumentation point to this service instrumentation point. However, the CPRs for the removal of instrumentation points consider two cases. At first, they check for corresponding instrumentation points and delete them. Secondly, they delete all instrumentation points without a set SEFF or SEFF action or with a set proxy oject because the SEFF or SEFF action can be deleted in certain cases before the changes are transitively propagated and the instrumentation point is deleted.

To support further SEFF actions in the extended IM, the extended IMM has been extended. It includes the additional options `EXTERNAL_CALL` (for `ExternalCallActions`) and `INTERNAL_CALL` (for `InternalCallActions`) in the `InstrumentationType`.

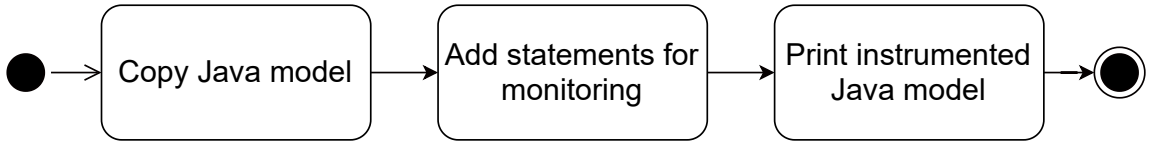


Figure 4.5.: Process of the adaptive instrumentation.

4.6. Adaptively Instrumenting the Source Code

Algorithm The adaptive instrumentation presented in this thesis is a combination of the existing prototypical implementations. The resulting algorithm is listed in 3 and visualized in Figure 4.5.

Algorithm 3 The newly implemented Source Code Instrumentation as the combination of the existing Source Code Instrumentations (see Algorithm Algorithm 1 and Algorithm Algorithm 2).

Input: Extended IM, Correspondence Model (CM), Source Code Model (SCM), Perform Full Instrumentation Flag (PFI)
Output: Instrumented Source Code

```
1:  SCMC ← copySourceCodeModel(SCM)
2:  for all SIM ∈ Extended IM.serviceInstrumentationPoints do
3:      sourceCodeElements ← getSourceCodeElements(SIM, CM)
4:      copiedElements ← findCopiedStatements(sourceCodeElements, SCMC)
5:      instrumentService(SIM.service, copiedElements)
6:      for all AIP ∈ SIM.actionInstrumentationPoints do
7:          if AIP.active or PFI then
8:              instrumentAbstractAction(AIP.type, AIP.action, copiedElements)
9:          end if
10:     end for
11: end for
12: printModel(SCMC)
```

At first, the adaptive instrumentation copies the code model in the V-SUM. Afterwards, every service instrumentation point is handled separately. For each service instrumentation point, the corresponding method and source code statements of its action instrumentation points are obtained. Based on the position of the code elements in the original model, their equivalent elements within the copied code model are looked up. Then, the SEFF and selected actions are instrumented. An action is only instrumented if the action instrumentation point is active or a full instrumentation shall be performed in which all instrumentation points are instrumented. During the actual instrumentation, model elements for instrumentation statements are created and added to the copied code model. The instrumentation statements contain method calls to a monitoring library which generates and transmits monitoring probes to the calibration pipeline [15]. At last, the copied and instrumented code model is printed. To generate a compilable instrumented version of the source code, the local clone of the repository is copied. Every root model object originating

from a source file is related to the equivalent source file in the copied repository. Next, the root model object is printed into the found source file. The instrumentation statements require the monitoring library during the compilation so that the references can be resolved by the compiler. To simplify the injection of the monitoring library and because the model objects of the instrumentation statements also require a model of the monitoring library in the copied code model, a minimal model of the monitoring library is generated. It consists of the called methods without an implementation and further elements which are not referenced. In addition, the model of the monitoring library is printed into every build project for the compilation and removed afterwards for the deployment.

In case that a method corresponding to a SEFF is instrumented, several statements for the method are generated. At the beginning of the method, one statement reports the entering of the method while other statements transfer the method arguments. After the original statements, the last instrumented exit statement signals the end of the method. If a `BranchAction` is instrumented, only an enter statement within the corresponding statement is added. For `ExternalCallActions`, a statement before the method call signals the external call. The number of iterations of `AbstractLoopActions` is counted. As a result, a statement before a loop initializes a counter which is incremented within the loop. After the loop has ended, an exit statement finishes the instrumentation of an `AbstractLoopAction`. `InternalCallActions` are instrumented in the same way as `InternalActions`. Corresponding statements will be surrounded by an enter and exit statement. All instrumentation statements deliver the id of the SEFF or SEFF action.

For `InternalActions` and `InternalCallActions`, two edge cases are considered. If the last statement is a return statement, the addition of the exit statement after the return would result in code which is not compilable. Therefore, the return value is stored in a new local variable within a separate statement. After this statement, the exit statement for the `InternalAction` is added followed by the return statement with the new local variable as return value. Additionally, if the corresponding statement of an `InternalAction` is the statement of an if statement, it is swapped with a block. Then, the original statement is added to the block and instrumented.

Example To illustrate the approach, a small example is given. It considers the `CacheManagerEndpoint` in version 1.3.1 of the TeaStore. As indicated by the package declaration shown in the class' excerpt in Listing 4.1, the class is contained in the Maven module `tools.descartes.teastore.persistence` [99]. Furthermore, the Maven module includes a Dockerfile so that a module is created for this Persistence service.

```
1  package tools.descartes.teastore.persistence.rest;
2  [...]
3
4  @Path("cache")
5  [...]
6  public final class CacheManagerEndpoint {
7  [...]
8
9    @DELETE
10   @Path("/cache")
```

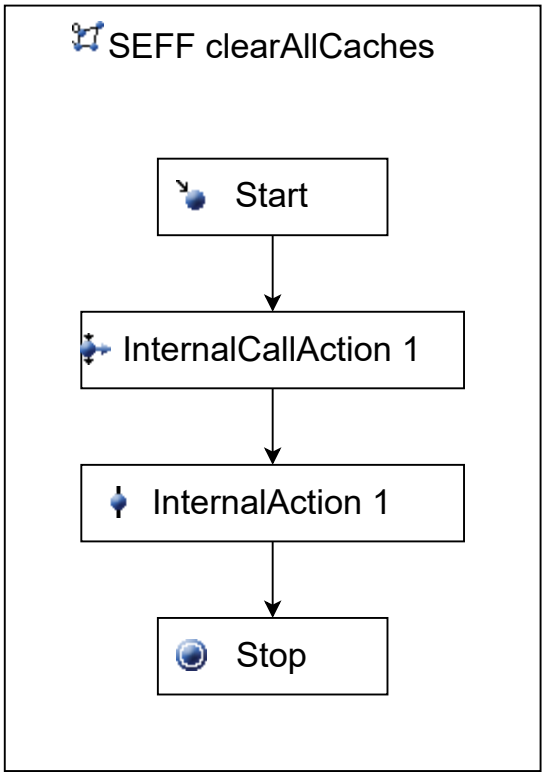


Figure 4.6.: SEFF of the `clearAllCaches` method.

```
11   public Response clearAllCaches() {
12       [...]
13   }
14 }
```

Listing 4.1: Excerpt of the `CacheManagerEndpoint` [99]).

When the CPRs for the PCM encounter the `CacheManagerEndpoint`, the associated module is detected, and a component is generated for it. Then, the `CacheManagerEndpoint` is annotated with `Path` [99]. As a result, the CPRs create a PCM interface for the class which is provided by the Persistence component at the same time. The public method `clearAllCaches` is also annotated with `Path` and the request method designator `DELETE` [99, 83]. If the class had not been annotated with `Path`, the class would have still been modeled as an interface because of the annotated method. For `clearAllCaches`, a SEFF is generated and reconstructed. It includes an `InternalCallAction` and `InternalAction` as depicted in Figure 4.6. In the CPRs for the extended IM, a new action instrumentation point is created for every action.

During the adaptive instrumentation, the corresponding statement of the `InternalAction` is enhanced by an enter and exit statement as listed in Listing 4.2. However, the code would not be compilable because the exit statement is located after the return statement. Therefore, as described in section 4.6, the return value is stored in a local variable which is returned after the exit statement. The improved instrumented code is shown in Listing 4.3.

```
1 monitoringController.enterInternalAction("_YHXHhwzdEeyhr8BpjCJSUQ");
2 return Response.ok("cleared").build();
3 monitoringController.exitInternalAction("_YHXHhwzdEeyhr8BpjCJSUQ");
```

Listing 4.2: Direct instrumentation of the `InternalAction` in the `clearAllCaches` method (partly from [99]).

```
1 monitoringController.enterInternalAction("_YHXHhwzdEeyhr8BpjCJSUQ");
2 Response resp1 = Response.ok("cleared").build();
3 monitoringController.exitInternalAction("_YHXHhwzdEeyhr8BpjCJSUQ");
4 return resp1;
```

Listing 4.3: Improved instrumentation of the `InternalAction` in the `clearAllCaches` method (partly from [99]).

5. Evaluation

This chapter covers the evaluation of the previously presented approach. Before section 5.2 introduces the evaluation plan, the used metrics are defined in section 5.1. Based on the plan, the case study is described in section 5.3, and the planned experiments with the case study are explained in section 5.4. Their results are investigated in section 5.5 followed by an assessment of the threats to validity in section 5.6.

5.1. Metrics

This section defines metrics which are used throughout the evaluation.

5.1.1. Jaccard Coefficient

Originally defined by Jaccard [57], the Jaccard similarity coefficient (JC) is defined as

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

and measures the similarity of two sets A and B [44]. In the case of a JC of 1.0, both sets are equal whereas lower values towards zero indicate more dissimilar sets. Analogous to Monschein [75], in the context of models, every model element is seen as a part of a set to enable the calculation of the JC for models.

In this thesis, the JC is utilized to compare Java and PCM `Repository` models using a generalized approach based on the comparison result of EMF Compare. Here, matched elements are part of the intersection and union of two models while unmatched elements are only part of the union. As a result, the number of matched elements is counted and divided by the number of matched and unmatched elements.

For Java models, the matching of the state-based model comparison from section 4.2 is reused. For PCM `Repository` models, a matching algorithm for structural equality and with a focus on the relevant elements was implemented. Therefore, ids are ignored, and named elements must have the same name to be equal. Furthermore, the positions of SEFF actions are compared, and certain referenced elements need to be similar, e. g., the parent interfaces of an interface. A complete list is given in Table 5.1.

Model element	Required similar referenced elements
`OperationInterface`	Parent interfaces
`OperationSignature`	Parameters and return type
`Parameter`	Data type
`CollectionDataType`	Inner type

Model element	Required similar referenced elements
`CompositeDataType`	Parent type
`InnerDeclaration`	Data type
`OperationProvidedRole`	Provided interface
`OperationRequiredRole`	Required interface
SEFF	Described service
`CollectionIteratorAction`	Collection parameter
`ExternalCallAction`	Called service

Table 5.1.: Overview over `Repository` model elements and their referenced elements which need to be similar so that the containing elements can be considered to be equal.

5.1.2. Instrumentation Point Matching Score

To evaluate the correct update of the extended IM, the Instrumentation Point Matching Score (IPMS) is defined. Every SEFF and SEFF action has a corresponding instrumentation point and, reversely, every instrumentation point has a SEFF or SEFF action if the models are correctly updated. Thus, a matching between the SEFF, SEFF actions, and instrumentation points is performed. If there is an unmatched object, the extended IM is not correct. As a result, the IPMS is the sum of all unmatched SEFF, SEFF actions, and instrumentation points.

5.1.3. Lower and Upper Bound of Expected Number of Added Statements During the Instrumentation

An indicator for the correct instrumentation is the number of added statements during the instrumentation. As outlined in section 4.6, in certain situations, additional statements are generated to avoid non-compiling code so that the exact expected number of added statements cannot be calculated. Instead, a lower and upper bound are determined. For every service and action instrumentation point, the minimum number of added statements is known and listed in Table 5.2. In addition, the minimal monitoring library model generates 10 statements. The combination of the previous values with one counted statement per service parameter results in the lower bound of expected statements. For the upper bound, the lower bound is extended by two statements for every return statement corresponding to an `InternalAction` or contained within a statement corresponding to an `InternalAction`. However, if a new block is set as the statement of an if statement caused by a return statement as the old statement of the if statement, the instrumentation creates three instead of two additional statements for the `InternalAction`. So, the upper bound can underestimate the actual number of added statements which can be above the upper bound in turn.

Instrumented element	Minimum number of added statements
SEFF	7
`ExternalCallAction`	1
`BranchAction`	1

Instrumented element	Minimum number of added statements
`LoopAction`	3
`InternalAction`	2

Table 5.2.: Minimum number of added statements per instrumented element.

One remark needs to be given: after the evaluation has been performed, it was discovered that the lower and upper bound can underestimate the number of added statements in another situation. As previously mentioned, for every service parameter, one statement is counted for the bounds. In this case, the service parameters are related to the parameters of the service corresponding to the instrumented service instrumentation point which are only created for `OrdinaryParameters` leaving other parameter types, e. g., variable length parameters, out. However, every parameter of an instrumented method is registered within its own statement so that the number of added statements can be higher if non-`OrdinaryParameters` are involved.

5.1.4. Accuracy Metrics

Based on the previous work of Monschein, the accuracy of a PCM instance is assessed by comparing the simulation results with monitoring data [75]. During the comparison, the following metrics are calculated:

- the differences of convential statistical measures (average, minimum, maximum, standard deviation, variance, and all quartiles),
- the Kolmogorov–Smirnov test (KS test), and
- the Wasserstein distance.

Additionally, the accuracies of two PCM instances are compared by calculating the metric

$$\text{"}CompareMetrics(M1, M2) = \sum_{i=1}^{|M1|} w_i * sign(compare(M1_i, M2_i))\text{"}\ [75]$$

where $M1, M2$ are the sets of accuracy metrics for both PCM instances with $|M1| = |M2|$, w_i is the priority for a specific metric, and *compare* calculates the difference between two metrics [75]. Conceptually, for every metric comparison, the more accurate model gains one point. The resulting score can be used, for example, to recognize an increasing or decreasing accuracy.

5.2. GQM Plan

In the following, the Goal Question Metric (GQM) plan for the evaluation is presented [6]. It starts with the declaration of the goals that shall be achieved. Every goal contains questions to assess if the goal has been achieved. To answer the questions, metrics are defined which are measured in the evaluation.

The GQM plan makes one assumption about the case study which is a requirement for the case study at the same time: the case study has a history with several commits available.

G0 The combined prototypical implementations for the first step of the CIPM approach are applicable to real world applications.

G0 defines the main goal of this thesis which is the execution of the Dev-time part of the CIPM approach with a case study as the representation of a real world application.

Q0.1 Can the combined prototypical implementations for the first step of the CIPM approach be automatically executed at once?

The automatic execution is one aspect of the CIPM approach. Therefore, **Q0.1** is used to assess if the combined prototypical implementations can be automatically executed.

M0.1.1 Satisfaction of **G2**, **G3**, and **G4** (Yes/No):

G2, **G3**, and **G4** regard the different prototypical implementations. As a consequence, their successful achievement is a requirement for their combination and **G0** so that the expected answer is yes.

M0.1.2 Execution time of the combined prototypical implementations (Time)

If the combined prototypical implementations can be automatically executed at once, their execution time is measurable and measured.

Q0.2 Are the combined prototypical implementations for the first step of the CIPM approach applicable to a case study?

While **Q0.1** checks that the combined prototypical implementations can be automatically executed, **Q0.2** checks their applicability on a case study.

M0.2.1 Comparison of the accuracies of the calibrated PCMs after executing the Dev-time part of the CIPM approach for a series of the case study's commits containing architectural-relevant changes (*CompareMetrics*)

The outcome of the Dev-time part of the CIPM approach is a calibrated PCM instance. Hence, for a series of the case study's commits with architectural-relevant changes, a series of calibrated PCM instances is generated. In addition, the successful execution of the CIPM approach results in accurate PCM instances. As a consequence, consecutive PCM instances in the obtained series are compared with the *CompareMetrics* function. If all PCM instances in the series are accurate, the result of the *CompareMetrics* function shall be in the interval $[-1, 1]$ indicating a successful execution.

G1 Vitruvius supports newer versions of Java.

To enable the usage of newer Java versions in Vitruvius and for the other goals, **G1** is defined.

Q1.1 Is Java source code containing features of Java 7-15 correctly propagated to the V-SUM?

The introduction of the support for newer Java versions allows the usage of features of Java 7-15. Therefore, **Q1.1** checks the correctness of the propagation of Java models containing these features.

M1.1.1 Equality between the Java code models in the VSUM and Java code models created by parsing the propagated source code (JC)

After the propagation of a Java code model with features of Java 7-15, the model in the V-SUM shall be equal to a model of the directly parsed source code. As a result, the JC for both models is calculated and is expected to be one.

G2 The state-based change propagation is capable of automatically applying changes from Git commits to Java code models.

The CIPM approach starts with the propagation of a commit's changes whose evaluation is covered by **G2**.

Q2.1 Does using the state-based change propagation correctly update the Java code models with changes from a Git commit?

Q2.1 covers the correctness of the state-based change propagation.

M2.1.1 Equality between the Java code models updated by using the state-based change propagation and Java code models created by parsing the complete state with the Git commit (JC)

Similar to **M1.1.1**, the Java code models in the V-SUM after the change propagation shall be in the same state as if the complete model of the source code with the commit's changes would be integrated into the V-SUM. Thus, the source code is parsed again, and its model is compared to the the code model in the V-SUM to calculate their JC. It is expected to be one.

G2.1 The commit-based integration strategy integrates existing source code into Vitruvius for the CIPM approach.

A special case of the state-based change propagation is its usage for the integration of existing source code into Vitruvius leading to **G2.1**.

Q2.1.1 Are the changes between an empty repository and the initial commit correctly propagated to the V-SUM?

In the first step, **Q2.1.1** is responsible for the correct propagation of the initial commit's changes into the V-SUM.

M2.1.1.1 Satisfaction of **G2** (Yes/No)

As the changes are propagated using the state-based change propagation, its correct operation is required which is covered by **G2**. Therefore, a yes as answer is expected.

M2.1.1.2 Equals **M2.1.1** (JC)

Analogous to **M2.1.1**, the JC for the code model in the V-SUM and the model of the propagated and repeatedly parsed source code is calculated which shall be one.

Q2.1.2 Do the CPRs correctly update the PCM and extended IM after changes in the Java code models?

In the second step, **Q2.1.2** checks that there are CPRs correctly updating the PCM and extended IM.

M2.1.2.1 Satisfaction of **G3** (Yes/No)

Achieving **G3** regarding the CPRs for the PCM and extended IM fulfills **Q2.1.2**. Therefore, the expected answer is yes.

Q2.1.3 Is existing source code integrated after using the commit-based integration strategy?

In addition to **Q2.1.2**, **Q2.1.3** ensures that the PCM and extended IM are correctly generated and integrated beside the Java model.

M2.1.3.1 Similiarity between the PCM after the integration and a reference model (JC, manual inspection)

For the PCM, the generated instance is compared to an independently and manually created reference model by calculating their JC. It is expected that they are not equal, but similar. As a consequence, the similarities and differences are investigated.

M2.1.3.2 Instrumentation points in the extended IM after the integration compared to all SEFFs and SEFF elements (IPMS)

For every SEFF and SEFF action, there has to be an instrumentation point in the extended IM so that the value zero is expected for the calculated IPMS.

Q2.1.4 Is there a difference between the integration of a commit and the propagation of multiple commits towards the integrated commit?

With the adaptation of the CPRs for the PCM, there is no differentiation between the integration and propagation of a commit. Therefore, **Q2.1.4** checks that there is no difference.

M2.1.4.1 PCM after the integration of a commit compared to the PCM after the propagation of multiple commits up to the integrated commit (JC, manual inspection)

The PCM after the integration of a commit is expected to be equal to a PCM which is generated by propagating multiple commits up to the integrated commit. Hence, the JC for both PCMs is calculated which shall be one.

G3 There are CPRs for the PCM and the extended IM.

G3 covers the update of the PCM and extended IM based on changes in the Java code models by summarizing the subgoals **G3.0**, **G3.1**, and **G3.2**.

Q3.1 Are there CPRs for the PCM?

Q3.1 aims at the CPRs for the PCM.

M3.1.1 Satisfaction of **G3.0** and **G3.1** (Yes/No)

The successful achievement, i. e., the expected answer yes, of **G3.0** and **G3.1** regarding the CPRs for the PCM result in their availability.

Q3.2 Are there CPRs for the extended IM?

In contrast to **Q3.1**, **Q3.2** regards the CPRs for the extended IM.

M3.2.1 Satisfaction of **G3.2** (Yes/No)

Analogous to **M3.1.1**, **M3.2.1** checks that **G3.2** is reached leading to the expected answer yes.

G3.0 The CPRs between Java code and the PCM from the co-evolution approach are adapted for the PCM update triggered by changes in the code models.

G3.0 defines the general goal for the CPRs between Java code and the PCM.

Q3.0.1 Is the PCM correctly updated after architectural-relevant changes in the source code models?

Q3.0.1 is concerned with the correctness of the CPRs and the updated PCM after architectural-relevant changes.

M3.0.1.1 PCM after architectural-relevant changes compared to a manually updated PCM (JC, manual inspection)

In **M3.0.1.1**, the architectural-relevant changes of a commit are analyzed to manually update the PCM in the state before the changes are applied. Then, the JC for this manually and the automatically updated PCM is calculated which shall be one. If there is a derivation, the differences are inspected to assess whether the differences are acceptable.

Q3.0.2 Does the PCM remain unchanged after non-architectural-relevant changes in the source code models?

In contrast to **Q3.0.1**, non-architectural-relevant changes are not allowed to change the PCM. Thus, **Q3.0.2** checks this condition.

M3.0.2.1 Equality of the PCM before and after non-architectural-relevant changes (JC)

The JC for the updated PCM and the PCM before the non-architectural-relevant changes are applied is calculated. It is expected to be one.

Q3.0.3 Is there a difference between the propagation of multiple commits and the propagation of these commits as one commit?

There is no differentiation in the the number of propagated commits because the CPRs depend only on the propagated changes. Therefore, **Q3.0.3** investigates if there is a difference in the propagation of multiple commits and these commits as one commit.

M3.0.3.1 PCM after the propagation of multiple commits as one commit compared to the PCM after the propagation of the mutliple commits (JC, manual inspection)

The PCM after the propagation of multiple commits shall be equal to a PCM obtained after the propagation of these commits as one commit. Thus, the JC for both PCMs is calculated which shall be one.

G3.1 Components of the case study are discovered during the PCM update in the first step of the CIPM approach.

The more specialized goal **G3.1** regards the component discovery for the PCM.

Q3.1.1 Does the addition of components in the Java code models result in the addition of components in the PCM?

The most important part of **G3.1** is the component discovery so that **Q3.1.1** checks that added components in the code are also added in the PCM.

M3.1.1.1 Difference between the number of added components in the PCM and the number of added components in the Java code models (Number)

Every added component in the code shall be added in the PCM. Therefore, the difference in the number of added components in the code and PCM is calculated which shall be zero.

M3.1.1.2 Added components in the PCM compared to the added components in the Java code models by comparing the updated PCM to a manually updated PCM (JC, manual inspection)

While **M3.1.1.1** checks that the number of added components in the code equals the number of added components in the PCM, **M3.1.1.2** investigates if the added components are created in the PCM and correspond to the added components in the code. As a consequence and similar to **M3.0.1.1**, the PCM before the addition of the components is manually updated with the added components only. The JC for the manually and automatically updated PCM is calculated. It is expected to be not one so that an additional manual inspection is performed.

Q3.1.2 Does the removal of components in the Java code models result in the removal of the corresponding components in the PCM?

Another part of the component discovery is the detection of removed components which is covered by **Q3.1.2**.

M3.1.2.1 Difference between the number of removed components in PCM and the number of removed components in the Java code models (Number)

Analogous to **M3.1.1.1**, every removed component in the code shall result in the deletion of the corresponding component in the PCM. Thus, the difference in the number of removed components in the code and PCM is calculated which shall be zero.

M3.1.2.2 Removed components in the PCM compared to the removed components in the Java code models by comparing the updated PCM to a manually updated PCM (JC, manual inspection)

Analogous to **M3.1.1.2**, the PCM before the removal of components is manually updated by deleting the components. Then, the JC for the manually and automatically updated PCM is calculated. There is no expectation on the value of the calculated JC. As a result, the differences of both PCMs are inspected if the PCM is not one.

Q3.1.3 Do generated components abstract from the corresponding parts in the source code models?

By the definition of the component discovery strategy, generated components shall abstract from the source code. **Q3.1.3** is used to check if the execution of the prototypical implementation leads to the expected abstraction.

M3.1.3.1 Generated components compared to the corresponding parts in the source code models (Manual inspection)

In a manual comparison of the code structure and the generated PCM, the level of abstraction is investigated.

G3.2 The prototypical implementation that updates the IM based on the changes in Java code models is updated to use the extended IM.

G3.2 regards the CPRs for the extended IM.

Q3.2.1 Is the extended IM correctly updated after changes in the Java code models?

Q3.2.1 checks the correctness of the CPRs for the extended IM.

M3.2.1.1 Instrumentation points in the extended IM after changes compared to all SEFFs and SEFF elements (IPMS)

Similar to **M2.1.3.2**, changes of SEFFs result in the update of the extended IM so that the IPMS is calculated which shall be zero.

G3.2.1 The proposed approach for the incremental fine-grained SEFF reconstruction further reduces the monitoring overhead compared to not using the approach.

G3.2.1 defines the goal for the incremental fine-grained SEFF reconstruction as an approach for an additional reduction of the monitoring overhead.

Q3.2.1.1 Can the monitoring overhead be reduced compared to not using the incremental fine-grained SEFF reconstruction?

Q3.2.1.1 assesses if the incremental fine-grained SEFF reconstruction can further reduce the monitoring overhead compared to not using this reconstruction approach.

M3.2.1.1.1 Ratio of deactivated to all instrumentation points (Percentage)

With **M3.2.1.1.1**, the reduction of the monitoring overhead in the context of the instrumentation points for the incremental fine-grained SEFF reconstruction is calculated as the ratio of deactivated to all instrumentation points.

M3.2.1.1.2 Ratio of deactivated action to all action instrumentation points (Percentage)

In addition to **M3.2.1.1.1**, **M3.2.1.1.2** measures the reduced monitoring overhead in the context of the action instrumentation points as the ratio of deactivated action to all action instrumentation points.

M3.2.1.1.3 Comparison of the ratios **M3.2.1.1.1** and **M4.3.1** (Difference)

The potential improvement in the monitoring overhead reduction in the context of the instrumentation points is expressed in the calculated difference between the ratios **M3.2.1.1.1** and **M4.3.1**. It is expected that the difference indicates an increased monitoring overhead reduction with the incremental fine-grained SEFF reconstruction.

M3.2.1.1.4 Comparison of the ratios **M3.2.1.1.2** and **M4.3.2** (Difference)

Analogous to **M3.2.1.1.3**, **M3.2.1.1.4** calculates the difference between the ratios **M3.2.1.1.2** and **M4.3.2** in the context of the action instrumentation points. An indication for an improved monitoring overhead reduction with the incremental fine-grained SEFF reconstruction is expected.

M3.2.1.1.5 Difference between the monitoring overhead with and without incremental fine-grained SEFF reconstruction (Time difference)

In contrast to **M3.2.1.1.3** and **M3.2.1.1.4**, **M3.2.1.1.5** measures the difference between the reduced monitoring overhead with and without the incremental fine-grained SEFF reconstruction in the temporal dimension.

M3.2.1.1.6 Percentage by which the monitoring overhead in **M3.2.1.1.5** is reduced (Percentage)

M3.2.1.1.6 calculates the relative value for the time difference in **M3.2.1.1.5**.

Q3.2.1.2 How large is the reduction of the monitoring overhead in general?

After **Q3.2.1.1** checked that the monitoring overhead can be reduced with the incremental fine-grained SEFF reconstruction, **Q3.2.1.2** quantifies the extent of the reduced monitoring overhead.

M3.2.1.2.1 Equals **M3.2.1.1.1** (Percentage)

M3.2.1.1.1 calculates the reduction in the monitoring overhead in the context of the instrumentation points so that its value is reused for **M3.2.1.2.1**.

M3.2.1.2.2 Equals **M3.2.1.1.2** (Percentage)

Similar to **M3.2.1.2.1**, the value of **M3.2.1.1.2** which represents the reduced monitoring overhead in the context of the action instrumentation points is reused.

M3.2.1.2.3 Difference between the monitoring overhead with full instrumentation and with the incremental fine-grained SEFF reconstruction (Time difference)

In addition to **M3.2.1.2.1** and **M3.2.1.2.2**, **M3.2.1.2.3** compares the monitoring overhead with the incremental fine-grained SEFF reconstruction to the monitoring overhead with the full instrumentation by calculating their difference in the temporal dimension.

Q3.2.1.3 Is there an improvement in the estimated PMPs?

Q3.2.1.3 extends on the reduced monitoring overhead to assess if the reduction results in an improvement of the estimated PMPs.

M3.2.1.3.1 Accuracy of the PCM with estimated PMPs without the incremental fine-grained SEFF reconstruction compared to the accuracy of the SEFF with estimated PMPs with the incremental fine-grained SEFF reconstruction (*CompareMetrics*)

For the accuracy of validated PCMs with and without the incremental fine-grained SEFF reconstruction, the *CompareMetrics* function is calculated. It is expected that the validated PCM with the incremental fine-grained SEFF reconstruction is more accurate because of the reduced monitoring overhead.

G4 A prototypical implementation adaptively instruments the source code based on the extended IM and the Java code models and reduces the monitoring overhead.

G4 covers the adaptive instrumentation.

Q4.1 Are all and only the activated instrumentation points of the extended IM correctly instrumented in the instrumented source code?

Q4.1 checks that the activated instrumentation points are correctly instrumented.

M4.1.1 Changed methods in the instrumented source code compared to methods corresponding to a SEFF (Difference)

The instrumented source code is parsed to generate a Java model which is compared to the code model in the V-SUM to detect all changed methods. All of these found methods shall correspond to methods with a corresponding SEFF because only such methods are instrumented. A successful match for the methods provides an indication for a correct instrumentation.

M4.1.2 Number of added statements in the instrumented source code compared to the expected number of added statements (Difference)

In addition to **M4.1.1**, the statements added by the instrumentation are counted, and the resulting number is compared to the expected number of added statements. By knowing the number of added statements per instrumentation point, a lower and upper bound for the expected number of added statements can be approximated so that the actual number of added

statements has to lie within the bounds. In this case, **M4.1.2** indicates a correct instrumentation.

M4.1.3 Comparison of instrumented statements with the instrumentation points in the extended IM (Manual inspection)

While **M4.1.1** and **M4.1.2** are indications for a correct instrumentation, they do not guarantee the correctness. As a result, a manual inspection of randomly chosen instrumentation points ensures that they are correctly instrumented.

Q4.2 Can the instrumented source code be executed to generate monitoring probes?

Q4.2 expands on **Q4.1** to assess if the instrumented source code can be used for the monitoring.

M4.2.1 Successful compilation of the instrumented source code (Yes/No)

The first step for the monitoring is the compilation of the instrumented source code which shall result in the expected answer yes.

M4.2.2 Successful execution of the instrumented source code (Yes/No)

The second step is the actual execution of the instrumented source code for the monitoring. Thus, yes is expected as answer.

M4.2.3 Accuracy of the validated PCM (Accuracy Metrics)

At last, the accuracy for the PCM validated by the monitoring of the instrumented source code is determined using the accuracy metrics.

Q4.3 How large is the reduction of the monitoring overhead?

Q4.3 complements **Q4.1** and **Q4.2** by investigating the reduction of the monitoring overhead.

M4.3.1 Equals **M3.2.1.1.1** (Percentage)

Analogous to **M3.2.1.1.1**, the reduction of the monitoring overhead in the context of the instrumentation points is given as the ratio of deactivated to all instrumentation points.

M4.3.2 Equals **M3.2.1.1.2** (Percentage)

Analogous to **M3.2.1.1.2** and similar to **M4.3.1**, the reduced monitoring overhead is also calculated in the context of action instrumentation points as the ratio of deactivated action to all action instrumentation points.

M4.3.3 Difference between the monitoring overhead with the full and adaptive instrumentation (Time difference)

In addition to **M4.3.1** and **M4.3.2**, the reduction of the monitoring overhead is also determined in the temporal dimension as the difference between the monitoring overhead with the adaptive instrumentation and the monitoring overhead with the full instrumentation.

M4.3.4 Percentage by which the monitoring overhead is reduced (Percentage based on **M4.3.3**)

M4.3.4 expands on **M4.3.3** by calculating the relative value for the reduced monitoring overhead in **M4.3.3**.

5.3. Case Study

Based on the GQM plan, the following requirements for the case study are identified.

REQ1 The case study is a Java- and Microservice-based application.

As the prototypical implementation supports only Java code, and the CPRs are defined for Microservice-based applications, the case study needs to be a Microservice-based application written in Java.

REQ2 The case study has a Git repository with a history consisting of several commits.

To propagate Git commits and their changes to simulate the development of the case study, a history in a Git repository is required.

REQ3 The case study is an open source project and its repository is publicly accesible.

Using an open source project, the corresponding license simplifies the use of the case study for the evaluation. Besides, the publicly accesible repository allows the independent repetition of the evaluation.

REQ4 The commits contain architectural-relevant changes.

Architectural-relevant changes in the commits lead to an update of the PCM. Thus, they can be used to evaluate the CPRs.

REQ5 For a specific commit, there is a manually created PCM instance.

To evaluate the integration of existing code, the automatically generated PCM shall be compared to a reference instance. Therefore, a reference PCM needs to be available which models the case study's architecture at a particular point in time represented by a specific commit.

Based on the requirements, the TeaStore was selected as case study.

The TeaStore provides a Web-based store for tea and related products [65]. It is designed as a test and benchmarking framework for the evaluation of, e. g., performance modeling approaches, run-time auto-scalers, or energy efficiency and power prediction methods. The architecture implemented in Java consists of the six Microservices WebUI, Auth, Image-Provider, Recommender, Persistence, and Registry depicted in Figure 5.1 with their relations. The relations represent the communication paths between the Microservices realized by REST calls. In detail, the Registry service is responsible for the registration and discovery of the other Microservices. While the WebUI provides the user interface, the Auth service enables the authentication of users, the Image-Provider delivers images displayed in the store, and the Recommender service suggests products to the user. All services

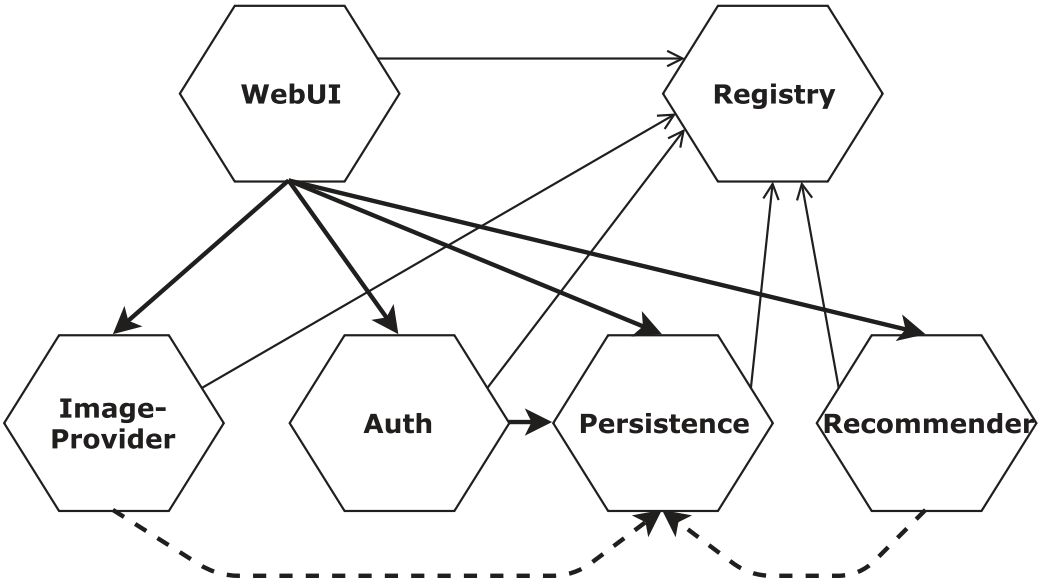


Figure 5.1.: Microservice-based architecture of the TeaStore [65].

except the Registry rely on the Persistence service for the retrieval and storage of data. In addition to the aforementioned Microservices, there is a supporting TraceRepository service for the built-in monitoring with Kieker which collects monitoring information if the monitoring is enabled. Beside the Microservices, the TeaStore also contains the Entities and RegistryClient libraries which include common data structures and functionality for all Microservices [99].

The source code and history of the TeaStore is available on GitHub[1] under the open source license Apache License 2.0 [99]. On 27th August 2021, the 1612 commits include the versions 1.0.0, 1.0.1, 1.0.2, 1.1.0, 1.2.0, 1.2.1, 1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.3.4, 1.3.5, 1.3.6, 1.3.7, 1.3.8, 1.3.9, and 1.4.0. For the evaluation, the commits from version 1.1 to version 1.3.1 are used and split into the intervals [1.1, 1.2] (I), [1.2, 1.2.1] (II), [1.2.1, 1.3] (III), and [1.3, 1.3.1] (IV). Furthermore, there is a manually and independently created PCM [75, 16] for version 1.3.x [99, 16].

Considering the interval (I), between version 1.1 and 1.2, the mandatory use of Checkstyle was introduced leading to a large number of changes: 27 of 50 commits affect 144 Java files with overall 9553 added and 7908 removed lines [99]. Four commits contain five architectural-relevant changes. The first change (I.A) is the removal of an servlet in the Auth service which identified the Auth service. Then, two classes in the Auth service were renamed (I.B). However, the complete file content was adjusted to comply with the Checkstyle configuration so that the changes were recognized as the removal of the old classes and addition of the newly named classes instead of a file renaming. In the context of methods corresponding to SEFFs, the statement of an if clause was embedded into a block for this if clause (I.C), in another method, a statement and loop were deleted while a password check for an if condition was introduced (I.D), and the visibility of two methods was reduced from public to private (I.E). There are no dependency changes. In

[1] https://github.com/DescartesResearch/TeaStore

Table 5.3, the commits of interval (I) with historical information and the occurrence of the architectural-relevant changes are shown.

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
0	(-)	236	26636	0
1	-	0	0	0
2	-	0	0	0
3	(I.A), (I.B)	23	1022	1022
4	-	1	2	1
5	-	0	0	0
6	-	7	969	755
7	-	8	839	716
8	-	62	4385	4138
9	-	4	20	10
10	(I.C)	3	58	53
11	(I.D)	2	21	6
12	-	17	496	429
13	-	21	568	154
14	-	10	335	26
15	-	4	75	58
16	-	4	58	75
17	-	4	75	58
18	-	1	1	1
19	-	1	2	2
20	-	2	196	145
21	-	3	432	389
22	-	1	49	42
23	-	1	3	0
24	-	0	0	0
25	-	2	142	115
26	-	0	0	0
27	-	1	1	1
28	(I.E)	7	174	96
29	-	24	124	110
30	-	1	1	1
31	-	0	0	0
32	-	0	0	0
33	-	0	0	0
34	-	25	111	125
35	-	25	125	111
36	-	0	0	0
37	-	0	0	0

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
38	-	0	0	0
39	-	0	0	0
40	-	0	0	0
41	-	0	0	0
42	-	0	0	0
43	-	0	0	0
44	-	0	0	0
45	-	0	0	0
46	-	0	0	0
47	-	0	0	0
48	-	0	0	0
49	-	0	0	0
50	-	0	0	0

Table 5.3.: Historical information about the commits of interval (I) which ranges from version 1.1 to version 1.2 [99]. The commits are continuously numbered beginning with version 1.1 as commit 0 and ending with version 1.2 as commit 50. In the context of interval (I), the version 1.1 is integrated into Vitruvius so that it contains the addition of the complete source code as architectural-relevant change. For the other commits, the contained architectural-relevant change is explicitly signed. In contrast, - marks no architectural-relevant changes.

The history of interval (II) consists of 20 commits of which 12 commits affect five Java files with overall 141 added lines and one removed line [99]. Three Java files (123 lines in total) were added. The changes include three architectural-relevant changes: (II.A) in the Auth service (II.A1) and WebUI service (II.A2), a new REST endpoint for obtaining the readiness has been added whereby both implementations are identical, (II.B) a method corresponding to a SEFF was extended by one statement, and (II.C), in the TraceRepository service, a servlet has been added which provides functions to control and access log files. The remaining changes are not architectural-relevant, and there are no changes in the dependencies. Table 5.4 displays all commits with historical information including the occurrence of the architectural-relevant changes.

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
0	(-)	1088	28281	0
1	-	0	0	0
2	-	0	0	0
3	-	0	0	0
4	-	1	5	0
5	-	1	5	1

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
6	-	1	2	1
7	-	1	6	0
8	-	1	1	1
9	-	0	0	0
10	(II.B)	1	1	0
11	Reverts (II.B)	2	1	18
12	-	0	0	0
13	(II.B), (II.C)	3	55	1
14	-	1	0	2
15	-	1	5	3
16	-	1	1	1
17	(II.A1)	1	57	0
18	(II.A2)	2	43	14
19	-	0	0	0
20	-	0	0	0

Table 5.4.: Historical information about the commits of interval (II) which ranges from version 1.2 to version 1.2.1 [99]. The commits are continuously numbered beginning with version 1.2 as commit 0 and ending with version 1.2.1 as commit 20. In the context of interval (II), the version 1.2 is integrated into Vitruvius so that it contains the addition of the complete source code as architectural-relevant change. For the other commits, the contained architectural-relevant change is explicitly signed. In contrast, - marks no architectural-relevant changes.

In interval (III), seven of 11 commits affect four Java files with overall 121 added and 134 removed lines while nine Java files with overall 215 added and 227 removed lines are affected by 12 of 100 commits in interval (IV) [99]. Both intervals contain no architectural-relevant changes and no changes in the dependencies. The historical information of interval (III) and (IV) are summarized in Table 5.5 and Table 5.6, respectively.

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
0	(-)	238	28421	0
1	-	0	0	0
2	-	0	0	0
3	-	2	2	2
4	-	1	2	1
5	-	1	1	0
6	-	1	1	1
7	-	1	113	131
8	-	2	4	2

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
9	-	0	0	0
10	-	1	2	1
11	-	0	0	0

Table 5.5.: Historical information about the commits of interval (III) which ranges from version 1.2.1 to version 1.3 [99]. The commits are continuously numbered beginning with version 1.2.1 as commit 0 and ending with version 1.3 as commit 11. In the context of interval (III), the version 1.2.1 is integrated into Vitruvius so that it contains the addition of the complete source code as architectural-relevant change. For the other commits, the contained architectural-relevant change is explicitly signed. In contrast, - marks no architectural-relevant changes.

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
0	(-)	238	28408	0
1	-	0	0	0
2	-	0	0	0
3	-	1	1	1
4	-	1	1	1
5	-	1	1	1
6	-	0	0	0
7	-	0	0	0
8	-	0	0	0
9	-	2	59	33
10	-	0	0	0
11	-	0	0	0
12	-	0	0	0
13	-	0	0	0
14	-	0	0	0
15	-	0	0	0
16	-	0	0	0
17	-	0	0	0
18	-	0	0	0
19	-	0	0	0
20	-	0	0	0
21	-	0	0	0
22	-	0	0	0
23	-	0	0	0
24	-	0	0	0
25	-	0	0	0

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
26	-	0	0	0
27	-	0	0	0
28	-	0	0	0
29	-	0	0	0
30	-	0	0	0
31	-	0	0	0
32	-	0	0	0
33	-	0	0	0
34	-	0	0	0
35	-	0	0	0
36	-	0	0	0
37	-	0	0	0
38	-	0	0	0
39	-	0	0	0
40	-	0	0	0
41	-	0	0	0
42	-	0	0	0
43	-	0	0	0
44	-	0	0	0
45	-	1	1	1
46	-	0	0	0
47	-	0	0	0
48	-	1	1	1
49	-	0	0	0
50	-	1	1	1
51	-	6	153	193
52	-	1	7	6
53	-	0	0	0
54	-	0	0	0
55	-	0	0	0
56	-	0	0	0
57	-	0	0	0
58	-	0	0	0
59	-	0	0	0
60	-	0	0	0
61	-	0	0	0
62	-	0	0	0
63	-	1	1	0
64	-	0	0	0
65	-	0	0	0

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines
66	-	9	227	215
67	-	9	215	227
68	-	0	0	0
69	-	0	0	0
70	-	0	0	0
71	-	0	0	0
72	-	0	0	0
73	-	0	0	0
74	-	0	0	0
75	-	0	0	0
76	-	0	0	0
77	-	0	0	0
78	-	0	0	0
79	-	0	0	0
80	-	0	0	0
81	-	0	0	0
82	-	0	0	0
83	-	0	0	0
84	-	0	0	0
85	-	0	0	0
86	-	0	0	0
87	-	0	0	0
88	-	0	0	0
89	-	0	0	0
90	-	0	0	0
91	-	0	0	0
92	-	0	0	0
93	-	0	0	0
94	-	0	0	0
95	-	0	0	0
96	-	0	0	0
97	-	0	0	0
98	-	0	0	0
99	-	0	0	0
100	-	0	0	0

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines

Table 5.6.: Historical information about the commits of interval (IV) which ranges from version 1.3 to version 1.3.1 [99]. The commits are continuously numbered beginning with version 1.3 as commit 0 and ending with version 1.3.1 as commit 100. In the context of interval (IV), the version 1.3 is integrated into Vitruvius so that it contains the addition of the complete source code as architectural-relevant change. For the other commits, the contained architectural-relevant change is explicitly signed. In contrast, - marks no architectural-relevant changes.

The commits are given by the Git log command in linear, chronological order [34]. In addition, a mapping of the commit numeration to the commit's hash values is provided in Table A.1.

5.4. Experiments

Based on the GQM plan and case study, the following experiments were executed.

5.4.1. Experiment E1

Experiment **E1** aims at the evaluation of the goals **G1**, **G2**, **G3.0**, **G3.2**, and **G4** by executing the combined prototypical implementations for the first step of the CIPM approach with interval (II). Thus, version 1.2 of the TeaStore is integrated into Vitruvius at first. Afterwards, the changes of all commits between version 1.2 and 1.2.1 are propagated to the V-SUM to simulate their development. Here, the V-SUM includes the CPRs between Java and the extended IM and the incremental SEFF reconstruction. In addition, the source code is adaptively instrumented, and the execution times for the change propagation, adaptive instrumentation, and the overall process are measured (**M0.1.2**). At last, the resulting artefacts are evaluated.

For all commits which are successfully propagated, the JC for the Java model in the V-SUM and a newly generated model of the source code is calculated (**M1.1.1** and **M2.1.1.2** for the integrated commit / **M2.1.1** for propagated commits). Furthermore, the PCMs are manually updated with the propagated changes to calculate the JC for the manually and automatically updated PCM (**M3.0.1.1** / **M3.0.2.1**). Regarding the extended IM, the IPMS is calculated (**M2.1.3.2** for the integrated commit / **M3.2.1.1** for propagated commits).

If the instrumentation is performed, the added statements in the instrumented source code are counted beside the determination of the expected number of added statements (**M4.1.2**). Moreover, the instrumented code is parsed into a Java model to compare it to the model in the V-SUM to find all changed methods and to relate them to methods corresponding to a SEFF (**M4.1.1**). If both of the previous indicators signal a possible correct instrumentation, the correct instrumentation of radomly selected instrumentation points is checked (**M4.1.3**). In the context of action and all instrumentation points, the reduced monitoring overhead is calculated (**M4.3.1**, **M4.3.2**).

5.4.2. Experiment E1.1

Experiment **E1.1** is a continuation of **E1** in which the fully and adaptively instrumented code is compiled and packaged with Java 8 (**M4.2.1**). From the resulting artefacts, the classes representing the minimal monitoring library are removed. Afterwards, the artefacts are deployed according to the TeaStore documentation [33], i. e., the war files run inside of an Apache Tomcat 8.5.69 server which uses Java 8 and includes the fully implemented monitoring library while a configured MySQL Community Server 8.0.26 runs externally providing the database.

The TeaStore contains an Apache JMeter test plan (here, called *default*) for load tests [99] which is adjusted to simulate one user and executed with Apache JMeter 4.0 to perform the requests on the instrumented code and to generate monitoring probes (**M4.2.2**). At the same time, the existing pipeline of Monschein runs in order to receive the monitoring probes, to calibrate the PCM, and to determine the monitoring overhead in the temporal dimension from within the pipeline. As Apache JMeter reports the response times for successful requests [2], the monitoring overhead can also be assessed from an external point of view. As a consequence, by executing the fully and adaptively instrumented code, their difference in the monitoring overhead in the temporal dimension is calculated (**M4.3.3**, **M4.3.4**). Additionally, the resulting validated PCM is simulated and compared to monitoring data obtained without executing the validation by calculating the accuracy metrics (**M4.2.3**).

In the calibration pipeline, additional Apache JMeter test plans are provided for the TeaStore [15]. The `20user_cart_big`[2] (here, called *20*) was selected and adjusted to one user for a second monitoring and calibration, i. e., the previously described procedure is repeated with the 20 test plan.

5.4.3. Experiment E1.2

Experiment **E1.2** complements **E1** and **E1.1** by repeating both experiments to evaluate **G3.2.1**. In contrast to **E1** and **E1.1**, the CPRs between the PCM and extended IM and the incremental fine-grained SEFF reconstruction are used. As a result, the reduced monitoring overhead for the incremental fine-grained SEFF reconstruction (**M3.2.1.1.1**, **M3.2.1.1.2**, **M3.2.1.2.1**, **M3.2.1.2.2**, **M3.2.1.2.3**) and the difference in the monitoring overhead with and without the incremental fine-grained SEFF reconstruction (**3.2.1.1.3**, **M3.2.1.1.4**, **M3.2.1.1.5**, **M3.2.1.1.6**) are calculated. Furthermore, the accuracy of the validated PCM with the incremental fine-grained SEFF reconstruction is investigated so that the *CompareMetrics* function for the accuracy metrics of the validated PCMs with and without the incremental fine-grained SEFF reconstruction is calculated (**M3.2.1.3.1**).

[2] `https://github.com/CIPM-tools/CIPM-Pipeline/blob/documentation/cipm.consistency.root/cipm.consistency.tools.evaluation.docker/teastore/cipm-teastore-load/load/20user_cart_big.jmx`

5.4.4. Experiment E2

To increase the confidence in the results of **E1**, it is repeated with the other intervals and the CPRs between the PCM and extended IM. Thus, experiment **E2** covers the repetition of **E1** with interval (I).

5.4.5. Experiment E3

Analogous to **E2**, experiment **E3** repeats **E1** with interval (III).

5.4.6. Experiment E4

Complementing **E1**, **E2**, and **E3**, experiment **E4** is the last repetition of **E1** with interval (IV).

5.4.7. Experiment E5

Experiment **E5** focuses on comparing the propagation of multiple commits with their propagation as one commit. Therefore, for each commit interval, the changes between the first and last commit are propagated as one commit on the integrated first commit. The resulting PCM is compared to the manually and automatically updated PCMs of the propagation of multiple commits within one of the previous experiments (**M3.0.3.1**). In addition, the PCM is compared to the integrated PCM of the next interval (**M2.1.4.1**). An exception for this comparison regards interval (IV) because version 1.3.1 was not integrated before. As a result, version 1.3.1 is integrated to enable the comparison for interval (IV).

5.4.8. Experiment E5.1

Experiment **E5.1** finishes **E5** by comparing the PCM of the integrated version 1.3.1 to the manually and independently created PCM (**M2.1.3.1**, **3.1.3.1**).

5.4.9. Experiment 6

The last experiment **E6** regards **G3.1**. In the considered intervals (I), (II), (III), and (IV), no component is added or removed. Thus, the removal and addition of one existing component is artificially performed. As no other service has a dependency on the WebUI, the WebUI Maven module is completely removed in one commit after version 1.3.1 to simulate its accidental deletion. With the following commit, the removal of the WebUI is reverted leading to its addition as a new component. Both commits are publicly available[3] and propagated on the integrated version 1.3.1. Afterwards, the resulting artefacts are evaluated as in **E1** (including **M3.1.1.2** and **M3.1.2.2**). In **E6**'s context, the PCM for the removal commit is manually updated while the PCM of version 1.3.1 is reused for the second commit. Additionally, the number of removed or added components in the source code and PCMs are counted to calculate their difference (**M3.1.1.1**, **M3.1.2.1**).

[3] `https://github.com/HansMartinA/TeaStore/tree/add-rem-com`

5.4.10. Final Evaluations

After all experiments have been conducted and analyzed, the results are used to assess which goals have been reached so that the dependent metrics **M0.1.1**, **M2.1.1.1**, **M2.1.2.1**, **M3.1.1**, and **M3.2.1** can be evaluated leading to conclusions for **G0**.

5.5. Results and their Analysis

This section contains the results of the experiments. At first, the combined results of **E1**, **E2**, **E3**, and **E4** are presented in subsection 5.5.1 followed by their extensions **E1.1** in subsection 5.5.2 and **E1.2** in subsection 5.5.3. Afterwards, the results of **E5** and **E5.1** are assessed in subsection 5.5.4 and subsection 5.5.5, respectively. After the results of E6 in subsection 5.5.6, the section is concluded with a summary in subsection 5.5.7.

In contrast to the presented CPRs between Java and the PCM, between **E1** and **E5**, all Java types corresponding to `CompositeDataTypes` are recursively visited to create `InnerDeclarations` without differentiating the type's origin. The differentiation was introduced after these experiments have been conducted. **E5.1** and **E6** were carried out with the implemented differentiation.

5.5.1. Results of E1, E2, E3, and E4

In this section, the results for **E1**, **E2**, **E3**, and **E4** are presented.

5.5.1.1. Overview over Propagated Commits

Table 5.7 shows the propagated commits for all intervals in the experiments **E1**, **E2**, **E3**, and **E4**. In preparation for **E5**, it includes the integration of version 1.3.1. In interval (I), 15 of 27 possible commits (55.6%) were propagated while all 12 commits (100%) with changes in Java files in interval (II) were propagated. Five of seven (71.4%) and 11 of 12 (91.7%) possible commits could be propagated in interval (III) and (IV), respectively.

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines	Number Vitruvius changes
Interval (I)					
0	(-)	236	26636	0	2052656
3	(I.A), (I.B)	23	995	995	5678
4	-	1	2	1	219
14	(I.C), (I.D)	100	7415	6011	2744
19	-	4	84	67	1307
20	-	2	180	129	0
21	-	3	396	353	1267
22	-	1	49	42	1348
23	-	1	3	0	0
25	-	2	136	109	0

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines	Number Vitruvius changes
27	-	1	1	1	0
28	(I.E)	7	170	92	49
29	-	24	124	110	2000
30	-	1	1	1	4
34	-	25	111	125	1995
35	-	25	125	111	2004
Interval (II)					
0	(-)	1088	28281	0	2053067
4	-	1	5	0	64
5	-	1	5	1	68
6	-	1	2	1	21
7	-	1	6	0	43
8	-	1	1	1	0
10	(II.B)	1	1	0	25
11	Reverts (II.B)	2	1	18	175
13	(II.B), (II.C)	3	55	1	369
14	-	1	0	2	3
15	-	1	5	3	87
16	-	1	1	1	23
18	(II.A)	2	86	0	214
Interval (III)					
0	(-)	238	28421	0	2053646
4	-	2	3	2	98
5	-	1	1	0	22
6	-	1	1	1	49
7	-	1	114	132	463
10	-	2	5	2	73
Interval (IV)					
0	(-)	238	28408	0	2053403
3	-	1	1	1	14
4	-	1	1	1	7
5	-	1	1	1	6
9	-	2	61	35	1179
45	-	1	1	1	2128
48	-	1	1	1	5
50	-	1	1	1	5
52	-	6	156	195	3998
63	-	1	1	0	4
66	-	9	229	217	6255
67	-	9	219	231	7308

Commit	Architectural-relevant change	Number changed Java files	Number added lines	Number removed lines	Number Vitruvius changes
Version 1.3.1					
0	(-)	236	28396	0	2059715

Table 5.7.: Propagated commits of all intervals and version 1.3.1.

During the execution of the combined prototypical implementations, the time for the model update, i. e., the change extraction and propagation, adaptive instrumentation, and complete process was measured. The values are depicted in Table 5.8. On average, an integration lasted 27.1 minutes while a propagation lasted 3.9 minutes. Considering the number of generated Vitruvius changes for an integration and propagation, the variation in the execution times can be explained. At the same time, the instrumentation has a minimal influence on the execution time. For an integration and propagation, the instrumentation lasted between 0.6 and 0.7 minutes on average.

Commit	Execution time of model update	Execution time of adaptive instrumentation	Overall execution time
Interval (I)			
0	36.1	0.5	36.6
3	6.3	0.7	6.9
4	3.8	-	3.9
14	3.3	0.6	4.0
19	2.7	-	2.7
20	3.8	-	3.8
21	3.1	-	3.1
22	3.5	-	3.5
23	7.2	-	7.2
25	2.8	-	2.8
27	8.2	-	8.2
28	4.0	-	4.1
29	3.4	-	3.4
30	3.3	-	3.3
34	3.4	-	3.4
35	3.3	-	3.3
Interval (III)			
0	23.3	0.6	23.9
4	3.9	-	3.9
5	3.6	-	3.6
6	4.3	-	4.3
7	3.6	-	3.6
10	3.6	-	3.6
Interval (IV)			

Commit	Execution time of model update	Execution time of adaptive instrumentation	Overall execution time
0	23.9	0.5	24.4
3	3.2	-	3.2
4	3.1	-	3.1
5	3.2	-	3.2
9	3.1	-	3.1
45	3.5	-	3.5
48	3.0	-	3.0
50	3.0	-	3.0
52	4.0	-	4.0
63	3.1	-	3.1
66	4.2	-	4.2
67	4.5	-	4.5
Version 1.3.1			
0	22.7	0.9	23.5
Average for integration			
-	26.5	0.6	27.1
Average for propagation			
-	3.8	0.7	3.9

Table 5.8.: Execution times for the intervals (I), (III), and (IV) and for version 1.3.1 in minutes. For interval (II), no execution times were measured.

5.5.1.2. Assessing the Correctness of the Model Update

As shown in Table 5.9, the JC for the compared Java and PCM models is 1.0 in all cases except for the integrated Java models. These cases have the same cause: the `SimilarityChecker` does not compare the array dimensions of parameter types in method declarations. The class model `java.security.MessageDigest` contains two methods with the name `update` and the base parameter type `byte` for the only parameter. The difference between both methods is the difference between the array dimensions of the parameter type. While one parameter type is `byte` and the other one is `byte[]`, the dimensions are not checked and both methods are mismatched with the other one in the code model of the V-SUM so that the addition and removal of an array dimension is reported by EMF Compare. As a consequence, these differences are propagated with the first commit. Although this leads to a minor incorrection in the Java models, it has no influence on the results. The mismatch occurred only in the described situation. If there had been another case, EMF Compare would have found it. Combining the results of the comparisons of the Java and PCM models with the IPMS of zero, it follows that the models were nearly correctly updated.

Commit	JC Java models	JC PCM	IPMS
Interval (I)			
0	0.999997	-	0

Commit	JC Java models	JC PCM	IPMS
3	1.0	1.0	0
4	1.0	1.0	0
14	1.0	1.0	0
19	1.0	1.0	0
20	1.0	1.0	0
21	1.0	1.0	0
22	1.0	1.0	0
23	1.0	1.0	0
25	1.0	1.0	0
27	1.0	1.0	0
28	1.0	1.0	0
29	1.0	1.0	0
30	1.0	1.0	0
34	1.0	1.0	0
35	1.0	1.0	0
Interval (II)			
0	0.999997	-	0
4	1.0	1.0	0
5	1.0	1.0	0
6	1.0	1.0	0
7	1.0	1.0	0
8	1.0	1.0	0
10	1.0	1.0	0
11	1.0	1.0	0
13	1.0	1.0	0
14	1.0	1.0	0
15	1.0	1.0	0
16	1.0	1.0	0
18	1.0	1.0	0
Interval (III)			
0	0.999997	-	0
4	1.0	1.0	0
5	1.0	1.0	0
6	1.0	1.0	0
7	1.0	1.0	0
10	1.0	1.0	0
Interval (IV)			
0	0.999997	-	0
3	1.0	1.0	0
4	1.0	1.0	0
5	1.0	1.0	0
9	1.0	1.0	0

Commit	JC Java models	JC PCM	IPMS
45	1.0	1.0	0
48	1.0	1.0	0
50	1.0	1.0	0
52	1.0	1.0	0
63	1.0	1.0	0
66	1.0	1.0	0
67	1.0	1.0	0
Version 1.3.1			
0	0.999997	-	0

Table 5.9.: Results of the model evaluation.

5.5.1.3. Assessing the Correctness of the Instrumentation

For assessing the correct instrumentation, the number of added statements was counted at first. The values including the lower and upper bounds are shown in Table 5.10. In every case, the values lie within the bounds so that the number of statements suggest the correct instrumentation.

Commit	Lower bound of expected statements	Number of actual statements	Approximated upper bound of expected statements
Interval (I)			
0	491	539	625
3	343	356	387
14	325	330	339
Interval (II)			
0	462	510	596
10	306	309	314
11	306	309	314
13	306	309	314
18	318	322	322
Interval (III)			
0	480	530	618
Interval (IV)			
0	480	530	618
Version 1.3.1			
0	480	530	618

Table 5.10.: Counted statements of the instrumented code in all intervals and version 1.3.1.

Next, the changed methods in the instrumented code were compared to methods with corresponding SEFFs. In Table 5.11, the numbers of unmatched changed methods and unmatched service instrumentation points are displayed. All values are zero so that only methods with corresponding SEFFs have changed. The last indicator which is also exposed

in Table 5.11 is the compilation of the instrumented code. The results suggest that the instrumented code is compilable. In combination with the changed methods and actual number of added statements, the successful compilation is a further indicator for the correct instrumentation.

Commit	Number of unmatched changed methods	Number of unmatched service instrumentation points	Successfully compiled?
Interval (I)			
0	0	0	Yes
3	0	0	Yes
14	0	0	Yes
Interval (II)			
0	0	0	Yes
10	0	0	Yes
11	0	0	Yes
13	0	0	Yes
18	0	0	Yes
Interval (III)			
0	0	0	Yes
Interval (IV)			
0	0	0	Yes
Version 1.3.1			
0	0	0	Yes

Table 5.11.: Comparison of the non-instrumented and instrumented code and compilation result of the instrumented code in all intervals and version 1.3.1.

Although the previous indicators support a potential correct instrumentation, they are not sufficient to confirm the correctness. Therefore, the manual inspection was performed in which it was ensured that the added statements for instrumentation points are correct and contain the right ids from the PCM. In the inspection, the activated action instrumentation points for all propagated commits were checked because of their low number as depicted in Table 5.12. During the inspection of these instrumentation points, the instrumentation of the affected classes which include instrumented service instrumentation points was also checked. For the integrated commits, different instrumentation points were selected. In interval (I), the `Reset` class of the TraceRepository service was checked because it is the only SEFF which contains further actions beside `InternalActions` or `InternalCallActions`. For all intervals, the `UserEndpoint` in the Persistence service was inspected as a baseline. In addition, further investigated classes varied between the Recommender for interval (II), the Registry for interval (III), and the Image-Provider for interval (IV). All manual inspections revealed a correct instrumentation and potential for improving the instrumentation and SEFF reconstruction.

Commit	Number of activated action instrumentation points	Number of instrumentation points
Interval (I)		
0	88	124
3	12	124
14	4	124
Interval (II)		
0	81	115
10	3	115
11	3	115
13	3	115
18	2	119
Interval (III)		
0	83	119
Interval (IV)		
0	83	119
Version 1.3.1		
0	83	119

Table 5.12.: Number of instrumentation points in all intervals and version 1.3.1.

The first potential improvement is found within the `Reset` class and its `deleteFolder` method. The original implementation consists of three statements (expression, if, and expression statement) of which the first and last statement are modeled as one `InternalAction` while the second statement contains an `ExternalCallAction` so that it converts the second statement to a `BranchAction`. This circumstance leads to a correct instrumentation as shown in Listing 5.1. However, the exit statement for the `InternalAction` is placed after the last statement. As a consequence, the monitoring of the `InternalAction` covers the whole method instead of only the first and last statement. Therefore, both the first and last statement should be one `InternalAction` each so that an exit statement is generated after the first statement and an enter statement is added before the last statement.

```
1  package tools.descartes.teastore.kieker.rabbitmq;
2  [...]
3
4  @WebServlet("/reset")
5  public class Reset extends HttpServlet {
6  [...]
7    public void deleteFolder(File folder, String prefix) {
8        [...]
9    }
10
11   public void deleteFolder(File folder) {
12       [...]
13       monitoringController.enterInternalAction("_86j04Ad_Eeyp7eds7yI7tA");
```

```
14        File[] files = folder.listFiles();
15
16        if (files != null) {
17            monitoringController.enterBranch("_86qikAd_Eeyp7eds7yI7tA");
18            [...]
19        }
20
21        folder.delete();
22        monitoringController.exitInternalAction("_86j04Ad_Eeyp7eds7yI7tA");
23        [...]
24    }
25 }
```

Listing 5.1: Excerpt of the instrumented `Reset` class (partially from [99]).

The second potential improvement is concerned with the handling of return statements. As outlined in section 4.6, some cases are considered to avoid non-compiling code, but not all cases are covered. Listing 5.2 is an excerpt from the instrumented `AuthUserActionsRest` class for commit three in interval (I). In the method `login` which consists of one `InternalAction` including all statements, the instrumentation is correct because the enter and exit statement of the `InternalAction` are located before and after the actual method implementation, respectively. Nevertheless, one if statement as example contains a return statement without an exit statement although there should be one.

A similar situation to the `login` method occurred in the `UserEndpoint` in the Persistence service listed in Listing 5.3. The first statement, an if statement, of the `findById` method is reconstructed as an `InternalAction`. The exit statement is placed within the if statement because of the contained return statement. After the if statement, an `InternalCallAction` is instrumented leading to a new enter statement without an additional exit statement for the previous `InternalAction`. Thus, such additional exit statements should be generated while it has to be ensured that further statements follow nested return statements.

Considering the `AuthUserActionsRest` class a second time, its `isLoggedIn` method contains two assignments of the final result to new local variables with a preceding enter and succeeding exit statement before the result is returned. The assignments are a consequence of nested method calls in the return value which are identified as `InternalActions` and `InternalCallActions`. Combined with the mechanism to split return statements, the repeated execution of the mechanism leads to the generation of the assignments. To avoid such situations, the method calls can be separated, or the enter and exit statements can be combined. An example for the combined statements is given in Listing 5.4 for the `register` method of the `RegistryREST`.

```
1 package tools.descartes.teastore.auth.rest;
2 [...]
3
4 @Path("useractions")
5 @Produces({"application/json"})
6 @Consumes({"application/json"})
```

```
7  public class AuthUserActionsRest {
8  [...]
9    @POST
10   @Path("login")
11   public Response login(SessionBlob blob, @QueryParam("name") String name,
         @QueryParam("password") String password) {
12       [...]
13       monitoringController.enterInternalAction("_MtzgAApFEeyde98rBgRSjw");
14       User user;
15       [...]
16       if (user != null) {
17           [...]
18           return Response.status(Response.Status.OK).entity(blob).build();
19       }
20       Response resp = Response.status(Response.Status.OK).entity(blob).build
             ();
21       monitoringController.exitInternalAction("_MtzgAApFEeyde98rBgRSjw");
22       return resp;
23       [...]
24   }
25   [...]
26   @POST
27   @Path("isloggedin")
28   public Response isLoggedIn(SessionBlob blob) {
29       [...]
30       monitoringController.enterInternalAction("_MuxJUApFEeyde98rBgRSjw");
31       Response resp1 = Response.status(Response.Status.OK).entity(new
             ShaSecurityProvider().validate(blob)).build();
32       monitoringController.exitInternalAction("_MuxJUApFEeyde98rBgRSjw");
33       monitoringController.enterInternalAction("_MuxwZApFEeyde98rBgRSjw");
34       Response resp2 = resp1;
35       monitoringController.exitInternalAction("_MuxwZApFEeyde98rBgRSjw");
36       monitoringController.enterInternalAction("_MuyXcApFEeyde98rBgRSjw");
37       Response resp3 = resp2;
38       monitoringController.exitInternalAction("_MuyXcApFEeyde98rBgRSjw");
39       return resp3;
40       [...]
41   }
42 }
```

Listing 5.2: Excerpt of the instrumented `AuthUserActionsRest` class (partially from [99]).

```
1  package tools.descartes.teastore.persistence.rest;
2  [...]
3
```

```
4  @Path("users")
5  public class UserEndpoint extends AbstractCRUDEndpoint<User> {
6  [...]
7    @GET
8    @Path("name/{name}")
9    public Response findById(@PathParam("name") final String name) {
10       [...]
11       monitoringController.enterInternalAction("_0o6tMApQEeycOb9KY39UQw");
12       if (name == null || name.isEmpty()) {
13           Response resp = Response.status(Status.NOT_FOUND).build();
14           monitoringController.exitInternalAction("_0o6tMApQEeycOb9KY39UQw");
15           return resp;
16       }
17       monitoringController.enterInternalAction("_0pAz5gpQEeycOb9KY39UQw");
18       [...]
19   }
20 }
```

Listing 5.3: Excerpt of the instrumented `UserEndpoint` class (partially from [99]).

```
1  package tools.descartes.teastore.registry.rest;
2  [...]
3
4  @Path("services")
5  @Produces({"application/json"})
6  public class RegistryREST {
7    @PUT
8    @Path("{name}/{location}")
9    public Response register(@PathParam("name") final String name, @PathParam("
       location") final String location) {
10       [...]
11       monitoringController.enterInternalAction("_LqueFAskEeyH6q2UaRYLeA");
12       monitoringController.enterInternalAction("_LqvFLAskEeyH6q2UaRYLeA");
13       boolean success = Registry.getRegistryInstance().register(name,
           location);
14       monitoringController.exitInternalAction("_LqvFLAskEeyH6q2UaRYLeA");
15       monitoringController.exitInternalAction("_LqueFAskEeyH6q2UaRYLeA");
16       [...]
17   }
18   [...]
19 }
```

Listing 5.4: Excerpt of the instrumented `RegistryREST` class (partially from [99]).

5.5.1.4. Reduced Monitoring Overhead

For **E1** and **E2** in wich the adaptive instrumentation has been performed, the reduced monitoring overhead in the context of the instrumentation points is listed in Table 5.13. Considering all instrumentation points, the overhead can be reduced by $\frac{2}{3}$ on average. The reduced fine-granular monitoring overhead, i. e., the overhead in the context of the action instrumentation points, ranges between 85.2% and 97.6%. As a result, the monitoring overhead is reduced. Nevertheless, in general, the reduction depends on the changes and how they affect the SEFFs.

Commit	Ratio deactivated to all instrumentation points	Ratio deactivated to all action instrumentation points
Interval (I)		
3	60.5%	85.2%
14	67.7%	95.5%
Interval (II)		
10	67.8%	96.3%
11	67.8%	96.3%
13	67.8%	96.3%
18	68.1%	97.6%

Table 5.13.: Reduced monitoring overhead in experiment **E1** and **E2**. In **E3** and **E4**, no adaptive instrumentation was performed.

5.5.1.5. Summary

In conclusion of the experiments **E1**, **E2**, **E3**, and **E4**, it follows that the results indicate the correct update of the models in the V-SUM and the correct instrumentation. In particular, with the correctly updated Java models and extended IM, the goals **G1**, **G2**, and **G3.2** are achieved.

5.5.2. Results of E1.1

During the execution of **E1.1**, monitoring probes were generated while the monitoring overhead in the pipeline and the response times were measured. As the authors of the TeaStore point out in a study they conducted with the TeaStore to assess challenges in the performance testing of Microservices, performance measurements such as response times are affected by the execution environment and dynamic features of Microservices (e. g., load balancers) so that the results of repeated measurements vary [26]. Therefore, the following monitoring overheads are cautiously investigated, and all calculated reductions are seen as tendencies only.

Table 5.14 displays the monitoring overhead in the temporal dimension within the pipeline. All values are normalized to one hour in order to rule out effects caused by different lengths of the monitoring. The reduced monitoring overhead ranges between -3.4% and 86.2% where the upper and lower bound are special cases. In the case of commit

11, if the absolute difference is considered in the context of the extent of the monitoring overhead, the variance can be neglected and attributed to the influence of the execution environment, i. e., the monitoring overhead depends on the execution. In the case of commit 18 and its change (II.A), none of the test plans contains direct or indirect calls to the added methods so that the overhead comes only from the coarse-grained instrumentation in the adaptively instrumented code while the fully instrumented code adds overhead for the fine-grained instrumentation. This results in a reduction of 86.2% for the default test plan. However, the 20 test plan achieves a reduction of 7.6%. As a consequence, the monitoring overhead depends on the test plan, the changes, and how the changes affect the PCM. All in all, a tendency for the reduction of the monitoring overhead within the pipeline is observable.

Commit	Normalized monitoring overhead of full instrumentation in $\frac{ms}{h}$	Normalized monitoring overhead of adaptive instrumentation in $\frac{ms}{h}$	Absolute difference in $\frac{ms}{h}$	Reduced monitoring overhead
10 (default)	36358.6	28588.0	-7770.6	21.4%
10 (20)	1641.6	1260.7	-380.9	23.2%
11 (default)	19588.5	20058.1	469.6	-2.4%
11 (20)	1118.8	1156.8	38	-3.4%
13 (default)	17529.1	16144.2	-1382.9	7.9%
13 (20)	1703.5	1137.3	-566.2	33.2%
18 (default)	11070.3	1528.7	-9541.6	86.2%
18 (20)	1146.7	1059.1	-87.6	7.6%

Table 5.14.: Reduced monitoring overhead in the temporal dimension in experiment **E1.1**.

In addition to the previously shown monitoring overhead within the pipeline, the response times and their differences are listed in Table 5.15. Here, the reductions of the response times are mixed. In 50% of the cases, there is a reduction towards the adaptive instrumentation while the average response time increases towards the adaptive instrumentation in the other cases. In two of eight cases, no change for the median can be reported, and the median decreases where the average response time rises in another two cases. As a result, the response times exhibit no clear tendency.

Type	Average response time	First quartile of response times	Second quartile of response times	Third quartile of response times
Commit 10 (default)				
Full instrumentation	7.98	5	9	10
Adaptive instrumentation	7.52	5	8	9
Absolute difference	-0.46	9	-1	-1
Relative reduction	5.8%	0%	11.1%	10%

Type	Average response time	First quartile of response times	Second quartile of response times	Third quartile of response times
Commit 10 (20)				
Full instrumentation	59.76	15	23	39
Adaptive instrumentation	264.71	15	27	55
Absolute difference	204.95	0	4	16
Relative reduction	-343.0%	0%	-17.4%	-41.0%
Commit 11 (default)				
Full instrumentation	14.88	5	8	27
Adaptive instrumentation	12.89	5	8	19
Absolute difference	-1.99	0	0	-8
Relative reduction	13.4%	0%	0%	29.6%
Commit 11 (20)				
Full instrumentation	60.92	12	34	44
Adaptive instrumentation	201.44	13	30	50
Absolute difference	140.52	1	-4	6
Relative reduction	-230.7%	-8.3%	11.8%	-13.6%
Commit 13 (default)				
Full instrumentation	14.69	5	8	28
Adaptive instrumentation	15.9	6	9	29
Absolute difference	1.21	1	1	1
Relative reduction	-8.2%	-20%	-12.5%	-3.6%
Commit 13 (20)				
Full instrumentation	75.48	12	35	48
Adaptive instrumentation	61.23	12	35	44
Absolute difference	-14.25	0	0	-4
Relative reduction	18.9%	0%	0%	8.3%
Commit 18 (default)				
Full instrumentation	27.5	5	9	29
Adaptive instrumentation	15.19	5	8	29
Absolute difference	-12.31	0	-1	0
Relative reduction	44.8%	0%	11.1%	0%
Commit 18 (20)				
Full instrumentation	173.09	15	41	63

Type	Average response time	First quartile of response times	Second quartile of response times	Third quartile of response times
Adaptive instrumentation	374.28	14	39	65
Absolute difference	201.19	-1	-2	2
Relative reduction	-116.2%	6.7%	4.9%	-3.2%

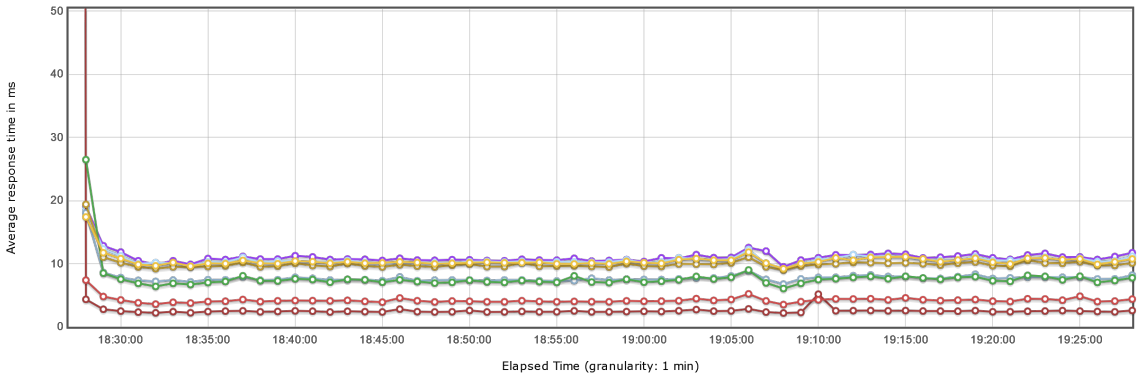
Table 5.15.: Reduction in the response times in experiment **E1.1**. All time values are given in milliseconds.

Moreover, looking at the progression of the average response times during the monitoring, it indicates that the execution environment and test plan influence the response times as they have an effect on the monitoring overhead. For example, the response times for commit 10 and 11 and the default test plan shown in Figure 5.2 and Figure 5.4 are stable and could be used for a comparison. In commit 13 (see Figure 5.6), the average response times for the adaptive instrumentation progress differently compared to the full instrumentation and hinder a comparison for the reduced monitoring overhead. Additionally, the full instrumentation of commit 18 provided instable response times for approximately half of the monitoring while the adaptive instrumentation was stable (see Figure 5.8). Considering the 20 test plan, the average response times are instable for all monitorings as depicted in Figure 5.3, Figure 5.5, Figure 5.7, and Figure 5.9 whereby the adaptively instrumented code has a greater variance in the commits 10 and 11.

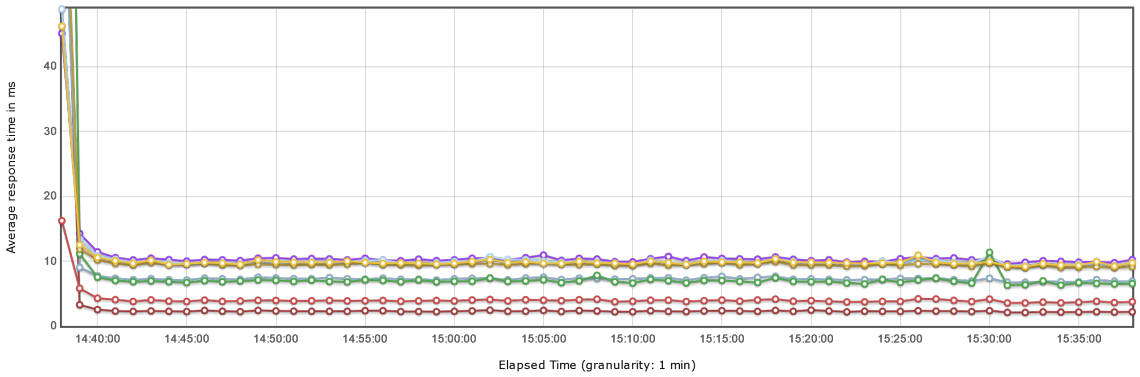
Although monitoring probes were generated, the PCM could not be calibrated. Due to technical and temporal issues, the prototypical implementations of the Dev-time part of the CIPM approach were not combined leaving a gap between the adaptive instrumentation, monitoring, and calibration pipeline. In detail, the system model generation proposed by Monschein [75] was not executed, and the PCM was not simulated because the PCM could not be adapted for the calibration pipeline. Furthermore, a potential conflict can arise in the usage of different versions of the PCM. While the prototypical implementation targets the PCM 5.0 [81], the calibration pipeline depends on version 4.1.0 [15]. However, by evaluating this thesis' approach and generating monitoring probes, and based on the positive evaluation results of the calibration pipeline [75], it is indicated that the TeaStore can be successfully applied on a combined prototypical implementation yielding an accurate and calibrated PCM instance. Nevertheless, such an evaluation is open to be performed including a resolution of the issues.

5.5.3. Results of E1.2

The execution of experiment **E1.2** failed for the first propagated commit 10 in interval (II). Although one `InternalAction` should have changed, no action changed. An exploration of this circumstance revealed that the incremental fine-grained SEFF reconstruction underlies an implicit assumption: when it is executed, there are two states of the changed method: the old state in the V-SUM which will be updated by the new and currently propagated state.

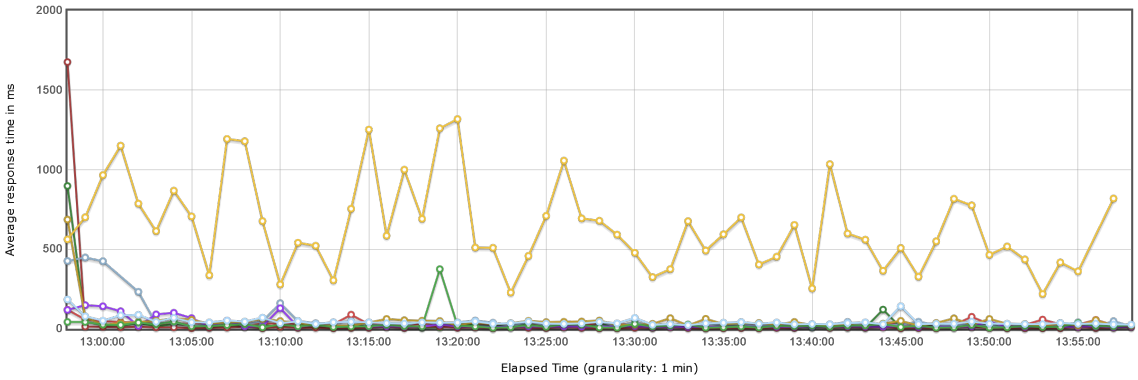


(a) Average response times for the full instrumentation during the monitoring.

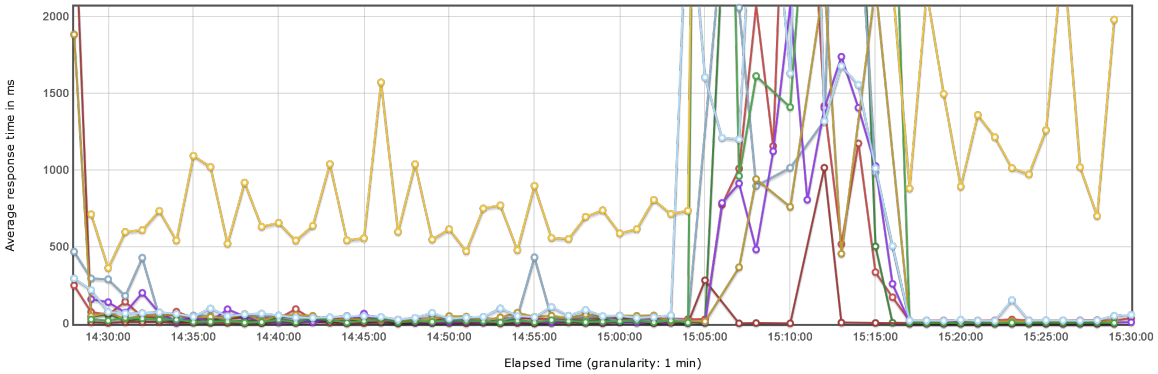


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.2.: Comparison of the average response times for all test cases in the default test plan and for commit 10.

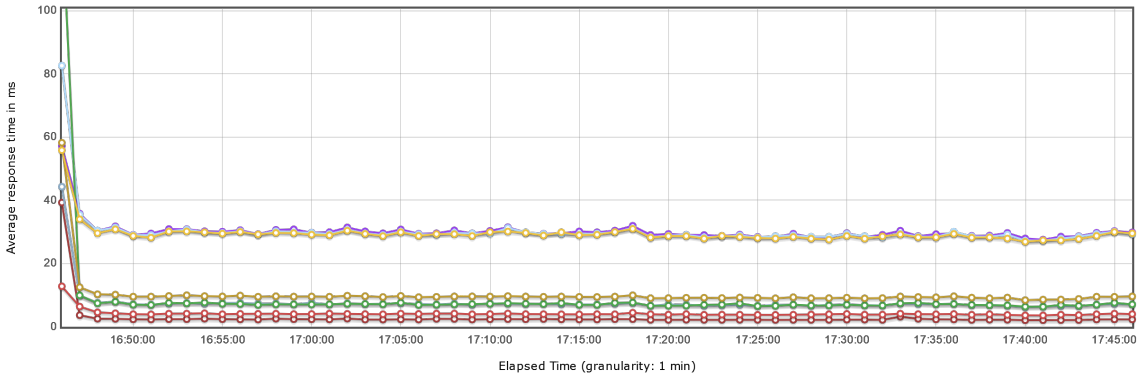


(a) Average response times for the full instrumentation during the monitoring.

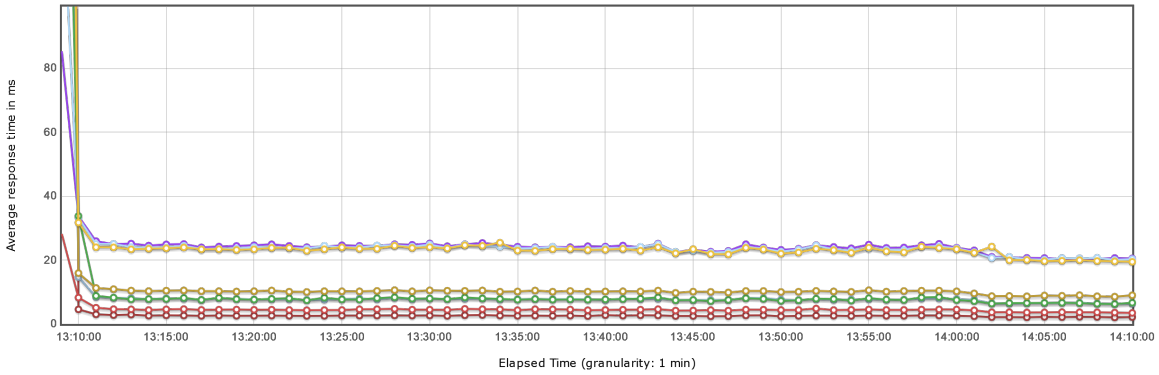


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.3.: Comparison of the average response times for all test cases in the 20 test plan and for commit 10.

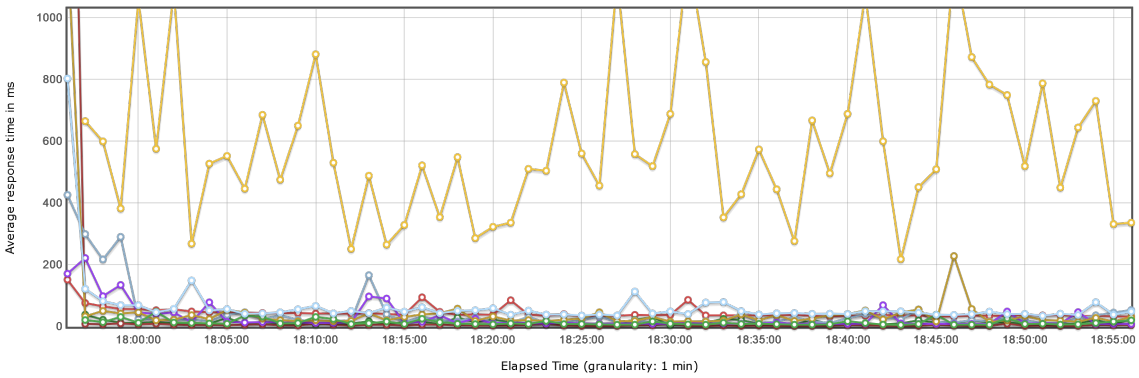


(a) Average response times for the full instrumentation during the monitoring.

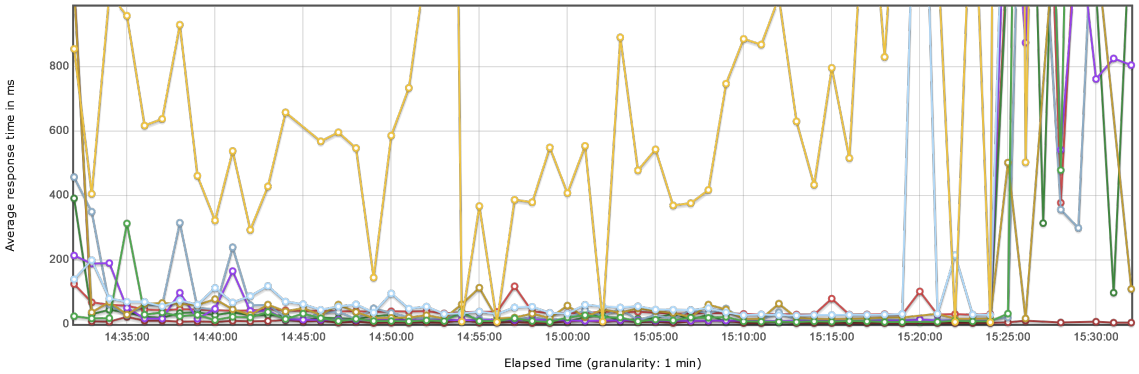


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.4.: Comparison of the average response times for all test cases in the default test plan and for commit 11.

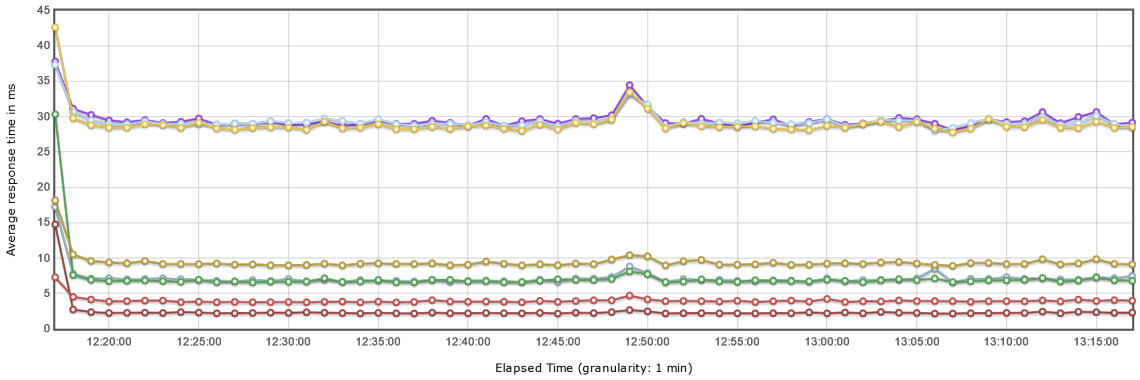


(a) Average response times for the full instrumentation during the monitoring.

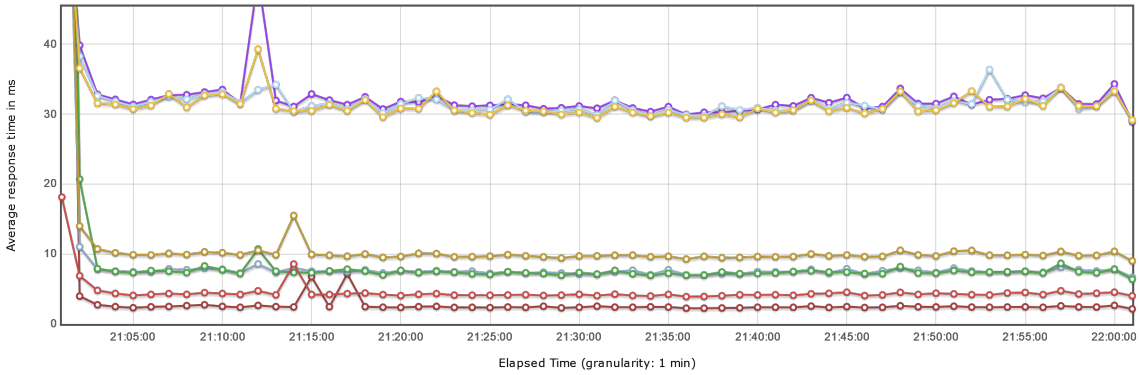


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.5.: Comparison of the average response times for all test cases in the 20 test plan and for commit 11.

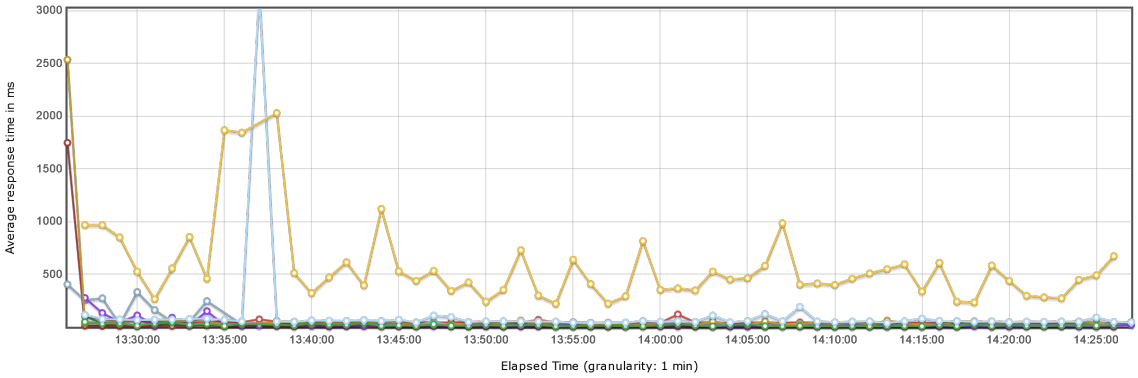


(a) Average response times for the full instrumentation during the monitoring.

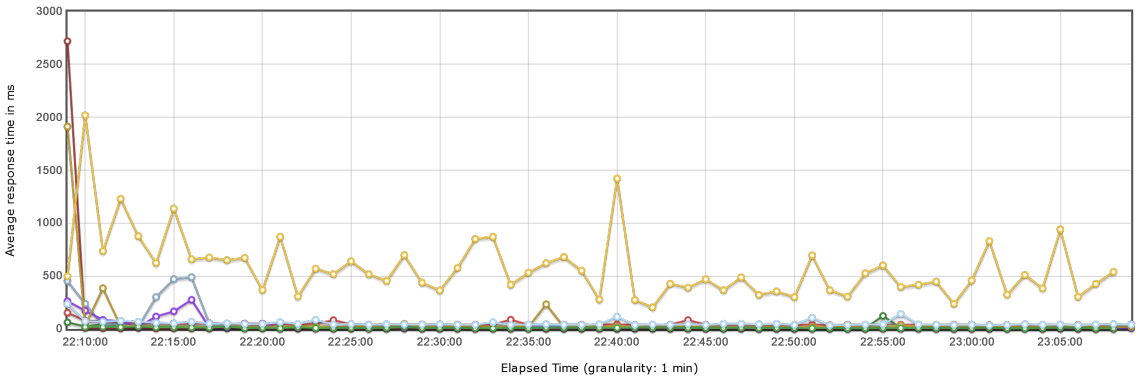


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.6.: Comparison of the average response times for all test cases in the default test plan and for commit 13.

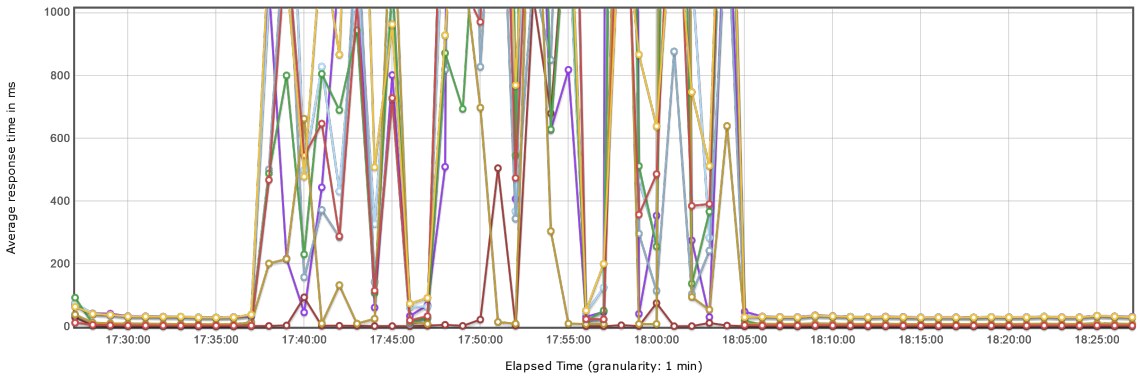


(a) Average response times for the full instrumentation during the monitoring.

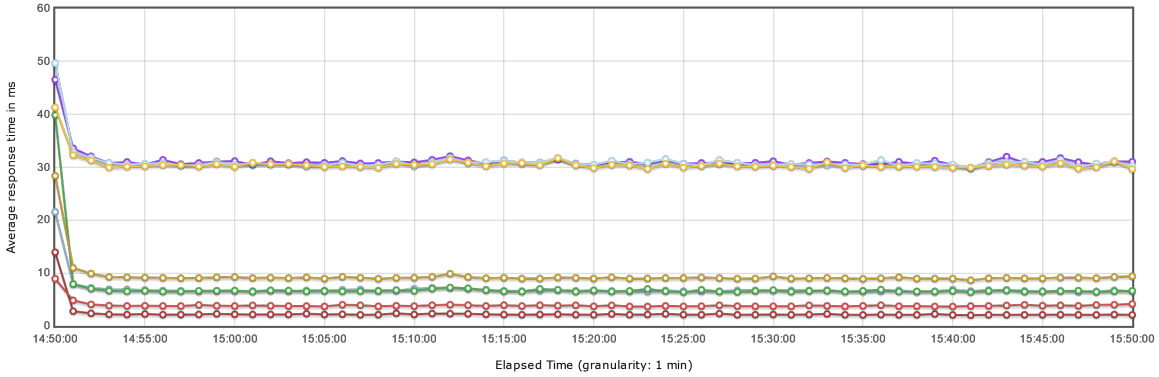


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.7.: Comparison of the average response times for all test cases in the 20 test plan and for commit 13.

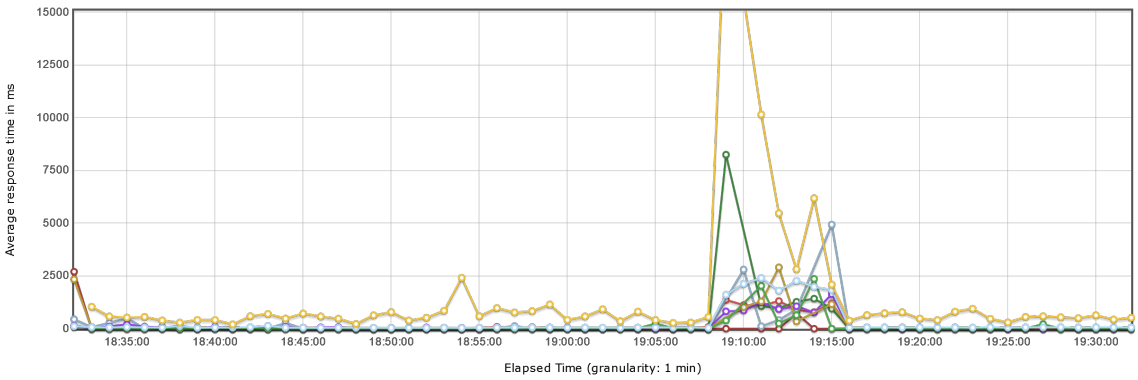


(a) Average response times for the full instrumentation during the monitoring.

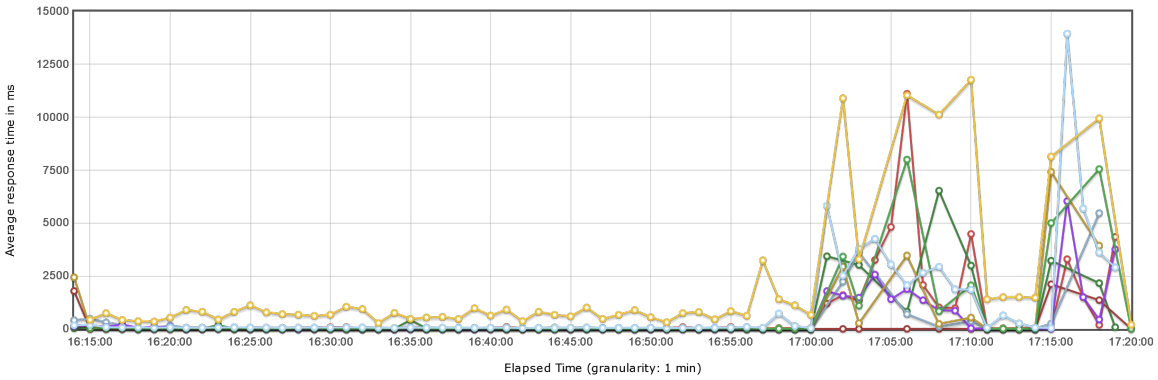


(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.8.: Comparison of the average response times for all test cases in the default test plan and for commit 18.



(a) Average response times for the full instrumentation during the monitoring.



(b) Average response times for the adaptive instrumentation during the monitoring.

Figure 5.9.: Comparison of the average response times for all test cases in the 20 test plan and for commit 18.

However, in the current prototypical implementation, due to the fine-grained changes, a method in the code model in the V-SUM is in exactly one state. Before changes are propagated, a method is in the old state. Afterwards, the method is in the new state. During a change propagation, the state of a method changes while it is not possible to predict when which change causes which state change. As a result, by executing the incremental fine-grained SEFF reconstruction after the actual change propagation triggered by the artificial changes, all changed methods are already in the new state. Additionally, during the change propagation, it is not possible to perform the incremental fine-grained SEFF reconstruction due to unforeseen change sequences. Therefore, an extended approach for the incremental fine-grained SEFF reconstruction is proposed.

1. Encode and store the match result for changed methods.

   After the state-based model comparison, the matching of the statements of changed methods is available forming the comparison of their old and new state. Therefore, the statements are encoded and stored with the matching result because the old and new state cannot be compared in later stages of the change propagation. Furthermore, before the fine-grained change sequence is created, the name of changed methods is set to an empty string and back to the original name to generate additional changes before and after the actual change sequence.

   The encoding of a statement model element is defined as follows: if the parent of the statement is a method declaration, the encoding is the string `0`. Otherwise, the parent of the statement is another statement so that the encoding of the statement is the encoding of its parent statement concatenated with the string `-` and the position of the statement in its parent statement. If the parent statement contains exactly one statement, the position of the statement is `0`. Else, if the parent statement contains a list of statements, the position of the statement equals the position in the list. In case that the parent statement includes multiple statements, the statements are numbered in the order of their occurrence so that the position of a statement equals its number.

2. Store correspondences between statements and actions of the old state.

   By generating and propagating artifical changes for changed methods before the actual change sequence, the corresponding statements for the old SEFF actions can be obtained. They are encoded and stored with their correspondence to the SEFF action. Afterwards, the correspondences between the SEFF actions and their statements are deleted in the correspondence model.

3. Execute the incremental fine-grained SEFF reconstruction.

   After the actual change sequence has been propagated and all changed methods are in their new state, the incremental fine-grained SEFF reconstruction is executed. It reconstructs the new SEFF from the changed method. Then, the new statements are encoded, and the old and new SEFFs are compared by comparing their encoded statements. The econded statements for the old SEFF are available due to step 1 and 2. As the encoding of statements result in strings, two statements are equal if their encoding strings are equal. Based on the comparison of the SEFF actions, the remaining procedure can be executed as it is already implemented.

As a consequence of the failure of **E1.2**, goal **G3.2.1** is not reached.

5.5.4. Results of E5

This section addresses the results of experiment **E5**. Analogous to subsubsection 5.5.1.1, Table 5.16 shows an overview over the propagated changes in experiment **E5**, and Table 5.17 displays the execution times. By propagating all commits of the considered intervals as one commit, the number of Vitruvius changes is greater than the numbers in the previous experiments although this is not necessarily the case for every set of commits.

Interval	Number changed Java files	Number added lines	Number removed lines	Number Vitruvius changes
(I)	144	9358	7713	12770
(II)	19	141	1	607
(III)	4	122	135	606
(IV)	9	219	231	7316

Table 5.16.: Propagated changes as one commit within the intervals.

Interval	Execution time of model update	Execution time of adaptive instrumentation	Overall execution time
(I)	6.2	0.5	6.9
(III)	5.2	-	5.2
(IV)	5.0	-	5.0

Table 5.17.: Execution times for the intervals (I), (III), and (IV) in experiment **E5** in minutes. For interval (II), no execution times were measured.

The evaluation of the updated models in the V-SUM resulted in an IPMS of zero and JC of 1.0 for all Java and PCM models as shown in Table 5.18. The PCMs were only compared to the automatically updated PCMs because there is no difference to the manually updated PCMs as illustrated in subsubsection 5.5.1.2. As a consequence, the models were also correctly updated, and it indicates that there is no difference in the number of propagated commits.

Interval	JC Java models	JC PCM	IPMS
(I)	1.0	1.0	0
(II)	1.0	1.0	0
(III)	1.0	1.0	0
(IV)	1.0	1.0	0

Table 5.18.: Results of the model evaluation in experiment **E5**.

In addition to the aforementioned PCM comparisons, the PCMs were related to integrated PCMs in only one comparison due to the equality between all propagated PCMs. The results are listed in Table 5.19. For all versions except version 1.2, the JC is 1.0 so that these models are equal. In the case of version 1.2, the JC is 0.975. The matching reveals that the propagated PCM contains more data types and `InnerDeclarations` than the integrated model. It is caused by the removal of parameters between version 1.1 and 1.2. The corresponding data types of the parameters are not removed and remain in the PCM whereas they are not used by other parameters. As a result, the data types are not created during the integration of version 1.2 resulting in the observed derivation. While this shows that there can be a difference in the propagation and integration, it is limited to data types in this case which is seen as acceptable.

Version	Propagated interval	Integrated interval	JC PCM
1.2	(I)	(II)	0.975
1.2.1	(II)	(III)	1.0
1.3	(III)	(IV)	1.0
1.3.1	(IV)	-	1.0

Table 5.19.: Results of the PCM comparison for propagated and integrated commits in experiment **E5**.

The adaptive instrumentation is checked as in subsubsection 5.5.1.3. In Table 5.20, the numbers of added statements as the first indicator in context of the lower and upper bounds are listed. For both intervals (I) and (II), the statement count is within the bounds. Table 5.21 contains the results for the second and third indicator. All changed methods in the instrumented code match with methods corresponding to SEFFs. Furthermore, the instrumented code compiles successfully. At last, in the manual inspection, all action instrumenation points and their corresponding affected classes for service instrumentation points were checked. No derivation from the expected instrumentation was found indicating the correct instrumentation.

Interval	Lower bound of expected statements	Number of actual statements	Approximated upper bound of expected statements
(I)	332	347	382
(II)	324	329	336

Table 5.20.: Counted statements of the instrumented code in experiment **E5**.

Interval	Number of unmatched changed methods	Number of unmatched service instrumentation points	Successfully compiled?
(I)	0	0	Yes

Interval	Number of unmatched changed methods	Number of unmatched service instrumentation points	Successfully compiled?
(II)	0	0	Yes
(III)	0	0	Yes
(IV)	0	0	Yes

Table 5.21.: Comparison of the non-instrumented and instrumented code and compilation result of the instrumented code in experiment **E5**.

The reduced monitoring overhead in experiment **E5** is listed in Table 5.22. For interval (II), the reduction is similar to the reduced overhead in the experiments **E1**, **E2**, **E3**, and **E4**. In contrast, interval (I) reduces the monitoring overhead by 56.5% for all and by 80.2% for the action instrumentation points. The values are lower because interval (I) contains more changes compared to interval (II) and single commits.

Interval	Ratio deactivated to all instrumentation points	Ratio deactivated to all action instrumentation points
(I)	56.5%	80.2%
(II)	65.5%	94.0%

Table 5.22.: Reduced monitoring overhead in experiment **E5**.

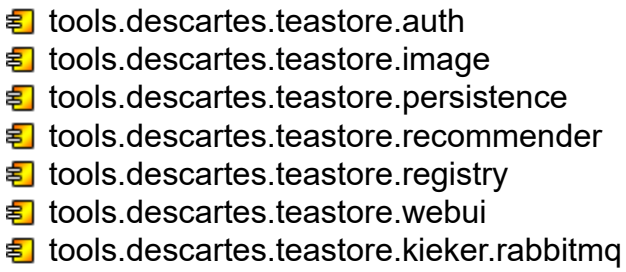
All in all, the accepted difference between the propagation and integration in combination with the correct update of the PCMs during the change propagation results in the achievement of **G3.0**.

5.5.5. Results of E5.1

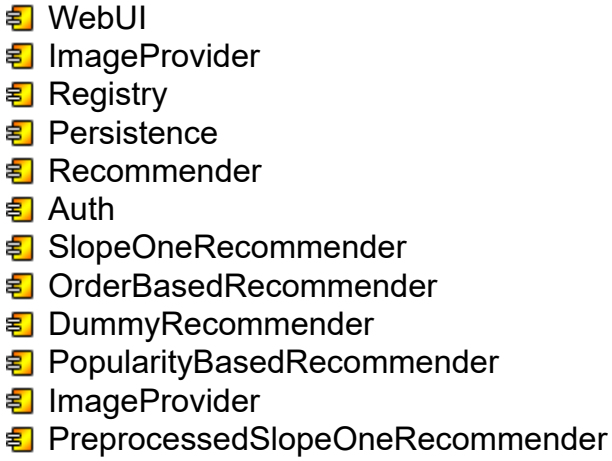
In **E5.1** and its comparison of the integrated version 1.3.1 or the automatically created PCM with the manually created PCM, the calculated JC is $\frac{1}{1234}$ because only the `Repository` model objects match. The comparison of the remaining objects include checks for name equality which is not given resulting in the low JC. For the following comparison, the source of the manually created PCM is [16].

Looking at the components in Figure 5.10, both PCMs contain the same components for every Microservice. In addition, the automatically created PCM contains a component for the TraceRepository service while the manually created PCM contains multiple components for different implementations of the Recommender service. The automatically created PCM does not include the different Recommenders because all implementations are located in the Recommender Maven module [99] so that only one component is created for the Recommender in the automatically created PCM.

Continuing with the interfaces in Figure 5.11, the manually created PCM models the abstract operations required by the TeaStore in the interfaces. In contrast, the automatically created PCM contains the operations enriched with technical details of the source code.

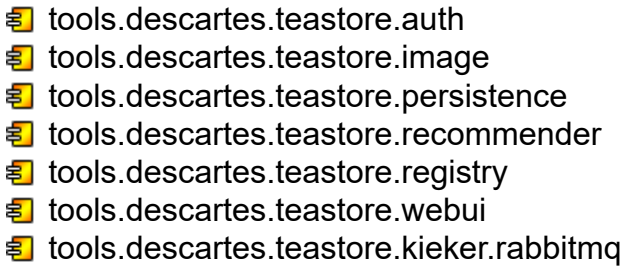


(a) Components of the automatically created PCM.

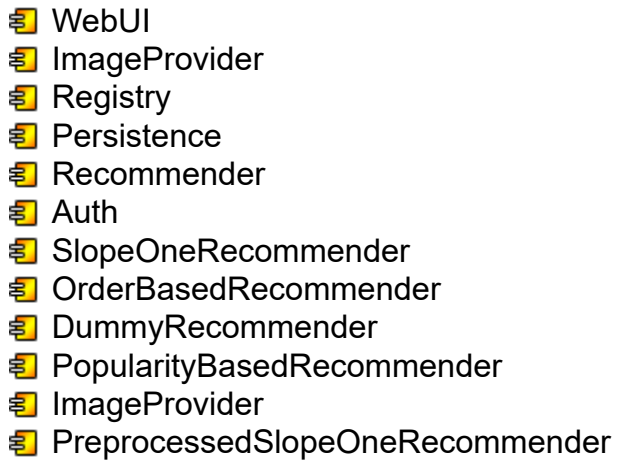


(b) Components of the manually created PCM [16].

Figure 5.10.: Comparison of the components in the automatically and manually created PCM.



(a) Selected interfaces of the automatically created PCM.



(b) Interfaces of the manually created PCM [16].

Figure 5.11.: Comparison of the interfaces in the automatically and manually created PCM.

As example, all servlets delivering the user interface are represented in interfaces, and the `DatabaseGenerationEndpoint` is responsible for the database creation which is not modeled in the manually created PCM. In the context of interfaces, the provided and required roles and SEFFs are compared. Both PCMs provide their interfaces with the components which implement the interface. The manually created PCM contains required roles and models high-level abstract behaviours within the SEFFs with external calls. The automatically PCM misses required roles and external calls. In the source code, the Microservices communicate over REST APIs by sending HTTP requests without direct or explicit method invocations. Thus, the CPRs cannot identify the invoked method and external calls. As a result, required roles are also not detected.

At last, the manually created PCM does not model data types while the automatically created PCM creates them. As example in Figure 5.12, the entities of the TeaStore such as `Orders` or `OrderItems` are modeled. In addition, a list of `OrderItems` is also included.

To summarize, the automatically created PCM creates components which abstract from the source code. Considering the interfaces, the abstraction levels differ. In case of the automatically created PCM, the extent of the interfaces depends on the use case and goal. In this thesis, the CPRs for the PCM focuses on modeling REST APIs. As Krogmann and Langhammer pointed out in previous comparisons between manually and automatically

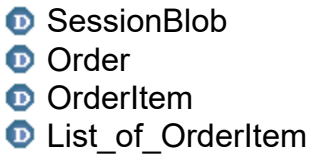


Figure 5.12.: Selected data types in the automatically created PCM.

created PCMs [70, 71], while manually created PCMs abstractly model the code on a high level, automatically created PCMs contain more details and technical information from which a manually created PCM can abstract. At the same time, an automatically created PCM is closer to and aligned with source code. All in all, it follows that the automatically created PCM for version 1.3.1 models the TeaStore except for external calls and required roles.

5.5.6. Results of E6

In Table 5.23, the propagated commits in experiment **E6** are shown. The higher number of Vitruvius changes led to an increased execution time (see Table 5.24).

Commit	Number changed Java files	Number added lines	Number removed lines	Number Vitruvius changes
0	236	28396	0	2059715
1	27	0	2651	14196
2	27	2651	0	17229

Table 5.23.: Propagated commits in experiment **E6**.

Commit	Execution time of model update	Execution time of adaptive instrumentation	Overall execution time
0	21.9	0.8	22.7
1	10.0	-	10.1
2	13.9	0.6	14.5

Table 5.24.: Execution times for the commits in experiment **E6** in minutes.

The results for the model comparisons in Table 5.25 reveal a JC of 0.999997 for the integrated Java model and 1.0 for every other Java model and PCM and an IPMS of zero so that the models were correctly updated. As a consequence, the difference in the number of added or removed components in the source code and PCM equals zero.

Commit	JC Java models	JC PCM	IPMS
0	0.999997	-	0
1	1.0	1.0	0
2	1.0	1.0	0

Table 5.25.: Results of the model evaluation in experiment **E6**.

Checking the correct instrumentation, the number of added statements for the integrated commit lies within the expected bounds. In case of the second commit, the number of added statements is one statement above the approximated upper bound as depicted in Table 5.26. This circumstance is caused by the underestimation of the expected bounds so that the difference is acceptable. Considering the matching of changed methods and the compilation of the instrumented code in Table 5.27 and the manual inspection, no derivations could be found indicating the correct instrumentation.

Commit	Lower bound of expected statements	Number of actual statements	Approximated upper bound of expected statements
0	480	530	618
2	316	319	318

Table 5.26.: Counted statements of the instrumented code in experiment **E6**.

Commit	Number of unmatched changed methods	Number of unmatched service instrumentation points	Successfully compiled?
0	0	0	Yes
2	0	0	Yes

Table 5.27.: Comparison of the non-instrumented and instrumented code and compilation result of the instrumented code in experiment **E6**.

At last, the reduced monitoring overhead for the second commit is 68.9% in the context of all and 98.8% in the context of action instrumentation points.

To summarize **E6**, the removal and addition of the WebUI service resulted in the correct update of the models and instrumentation. Therefore, goal **G3.1** is achieved.

5.5.7. Summary

As already mentioned, the goals **G1**, **G2**, and **G3.2** are achieved in the successful execution of the experiments **E1**, **E2**, **E3**, and **E4**. Analoguously, **G3.0** is reached in **E5**, and **G3.1** in **E6**. As a result, the requirements for **G3** are fulfilled. However, **G3.2.1** is not achieved. Regarding **G4**, the experiments does not include the assessment of the accuracy of a calibrated PCM instance because the calibration was not possible. Nevertheless, as discussed,

the results indicate that a resulting calibrated PCM instance shall be accurate so that the check can be neglected for the achievement of **G4**. Therefore, **G4** is reached.

At last, **G0** is considered. The planned comparison of calibrated PCM instances was not performed. However, the previously satisfied goals suggest that **G0** in a slightly diverging meaning is and can be achieved for the case study.

5.6. Threats to Validity

This section covers potential threats to the internal validity in subsection 5.6.1 and to the external validity in subsection 5.6.2.

5.6.1. Threats to Internal Validity

Internal validity determines how trustworthy the evaluation results are [20].

In this evaluation, a potential threat arises in the check of the updated Java models. For the comparison of the Java models, a new model is parsed from the source code and compared to the code model in the V-SUM by using the same parser and matching rules as in the change propagation. This could lead to the suppression of differences. However, by using the default diffing algorithm of EMF Compare, differences are detected if there is a derivation. In case of the intregated commits, differences were reported. Additionally, a wide range of commits was selected to cover different changes although they do not guarantee the coverage of all cases.

Another threat reside in the check of the correct update of PCMs by comparing the automatically updated PCM to a manually updated PCM because the manual update and comparison of the PCMs is performed by the author of this thesis. The goal of the comparison is a check of the correct update of the last commit's PCM to the current PCM. Therefore, it is sufficient to manually update the last version of the PCM according to the applied CPRs which is performed before the comparison to avoid influences from the automatically updated PCM.

In the context of the PCMs, their integrated versions were not checked. An exception is the comparison of the integrated version 1.3.1 and a manually created PCM. Beside similarities, it reveals that the integrated PCM abstractly models the TeaStore. Combining the comparison with the comparisons of integrated and propagated PCMs and checks of the correct PCM update, all propagated commits and the PCMs can be viewed backwards from the integrated version 1.3.1 to the first integrated version 1.1. Due to the correctly updated PCMs and the equality between integrated and propagated PCMs, it can be concluded that every integrated PCM abstractly models its corresponding TeaStore version.

In the evaluation of the adaptive instrumentation, not all instrumentation points were checked. However, the instrumentation of methods, `BranchActions`, `AbstractLoopActions`, and `ExternalCallActions` does not include edge cases so that the consideration of a sufficient number of cases rise an appropriate level of confidence in the instrumentation. `InternalActions` and `InternalCallActions` require an increased attention because they can change the code in certain cases. Based on the checked instrumentation points, the number of methods, `InternalActions`, and `InternalCallActions` are seen as suffi-

cient. Due to the low number of generated `BranchActions`, `AbstractLoopActions`, and `ExternalCallActions`, they need more checks. All in all, the checks indicate the correct instrumentation, but show potential for improvements.

5.6.2. Threats to External Validity

External validity is concernced with the generalization of the evaluation results [20].

The approach is only evaluated with the TeaStore and aims at Microservice-based applications. Thus, it limits conclusions for other projects and domains and requires further repetitions with adapted CPRs and different case studies.

Furthermore, the TeaStore is designed as a test and benchmarking framework [65]. As a consequence, it limits assertions about the applicability of the approach to real world applications. Therefore, as previously mentioned, the repetitions with different case studies can use open source applications with other design goals or industrial applications.

6. Related Work

This chapter gives an overview over related work split into different subjects. As a result, section 6.1 handles approaches analyzing repositories and their history. section 6.2 reviews approaches for the state-based model comparison. Approaches for reverse engineering software architectures are explained in section 6.3 while section 6.4 contains approaches for integrating existing source code into Vitruvius. At last, section 6.5 considers approaches for instrumenting source code.

6.1. Repository Analysis

In this section, different approaches are presented that involve the analysis of repositories. subsection 6.1.1 presents Screpo, and subsection 6.1.2 presents the approach of Stringfellow et al.

6.1.1. Screpo

Scheidgen et al. built Screpo as a model-based framework on top of EMF, JGit, and Modisco for mining software repositories [92]. Thus, Screpo defines a meta-model for repositories. A model instance represents a revision tree where each revision corresponds to a commit. JGit is used to perform the necessary Git operations. For each changed compilation unit in a revision, a model of the compilation unit is created by Modisco and attached to the revision. Only local references within the compilation unit are resolved. The repository model is stored in a database and allows its analysis. Screpo generates a snapshot model for visited revisions. A snapshot model contains a representation of the whole code base with resolved references between compilation units. Screpo assumes that the snapshot models are sequentially analyzed. As a result, there is only one snapshot model available which gets incrementally updated by replacing the changed compilation units.

Screpo was evaluated with parts of the Eclipse Foundation's repositories [92]. Scheidgen et al. compare the evaluation results with delta-compression. Using a state-based model comparison between two revisions of a compilation unit, their difference is obtained and stored instead of the complete compilation units. The evaluation indicates a higher execution time for the repository model creation due to additional compression. Screpo does not contain compression-based snapshot increments, but it is expected to reduce the execution time for the analysis.

In the CIPM approach, the code model in the V-SUM represents the state of a specific commit while the complete source code repository is modeled in Screpo for mining the repository [92]. Both approaches are similar in using EMF. However, Screpo builds upon Modisco for the Java code model. It explores the possibility to employ a state-based

model comparison for storing the changes between compilation units. The CIPM approach propagates changes obtained by a state-based model comparison.

6.1.2. The Approach of Stringfellow et al.

Stringfellow et al. propose an approach to find potential architectural problems combining three reverse engineering techniques [97]. The approach considers C and C++ code versioned with the Revision Control System (RCS) and defines a component as the set of all files in a directory.

The first technique recovers a component architecture [97]. Relationships between the components are derived from #include statements while the number of #includes determine a relationship's strength.

The second technique generates a change architecture [97]. For the generation, the RCS history data is extracted for every file to group related changes. It is assumed that only one developer makes related changes and checks them in within a certain time interval. Moreover, components with more changed lines of code are considered more change-prone. Based on these assumptions and the extracted data, change metrics are calculated reflecting the assumptions and characterizing the components and relationships. The change metrics are used to build the change architecture by filtering the components and relationships with the highest values.

The third technique recovers a fault architecture [97]. Similar to the change architecture, defect metrics are calculated based on the extracted data from the RCS history. After the application of a threshold, the fault architecture is derived consisting of components and relationships assumed to be the most fault-prone.

At last, the approach of Stringfellow et al. compares all three generated architectures [97]. Fault- or change-prone relationships are expected to become difficult to maintain. If such a relationship has no #include relationship, it is a hint for an increased complexity and architectural drift. The approach was applied on a commercial flight simulation system.

The approach of Stringfellow et al. uses historical data from a RCS repository to recover the change and fault architecture to predict change- and fault-prone components [97] while the CIPM approach uses historical data from a Git repository to incrementally update a general component-based architecture which enables performance predictions.

6.2. State-Based Model Comparison

Approaches including a state-based model comparison are introduced in this section. subsection 6.2.1 contains the approach for semantic lifting, subsection 6.2.2 contains an approach of Kehrer et al., and subsection 6.2.3 contains CoWolf.

6.2.1. Semantic Lifting

Kehrer et al. introduce an approach for semantic lifting [62]. They differentiate between low-level or atomic changes such as setting an attribute value and user-level changes or editing operations. The low-level changes are output by a state-based model comparison.

Then, semantic lifting is defined as the process of forming editing operations out of the low-level changes. The resulting editing operations are disjoint subsets of a set of low-level changes called semantic change sets.

Editing operations are expressed in edit rules which are implemented in Henshin [62]. Henshin is a language and tool environment for transformations within an EMF model based on graph transformation concepts [3]. Therefore, a Henshin rule has a left and a right side describing model patterns. If a Henshin rule is applicable to a model, elements matching the left side are transformed to the pattern of the right side. Furthermore, application conditions specify when a rule should be applied.

In the approach for semantic lifting, change set recognition rules group low-level changes according to an edit rule and allow their recognition [62]. The recognition rules are also Henshin rules and can be automatically generated from the corresponding edit rule, but need manual post-processing. The first step in the recognition process creates potential change sets by applying all recognition rules on low-level changes. In the following postprocessing step, heuristics are applied on the possible overlapping potential change sets to get the semantic change sets representing a minimal number of editing operations. The approach for semantic lifting was evaluated with synthetical and real test cases including UML models indicating a reduction of the number of reported differences.

Semantic lifting aims at providing a high-level representation of differences between two model states for users [62]. However, the state-based change propagation in the CIPM approach automatically updates models without user involvement.

6.2.2. The Approach of Kehrer et al.

Kehrer et al. extended their previously described approach for semantic lifting in order to create executable differences between two models [63]. In their extension, the low-level changes or basic operations are called change actions. It is assumed that every edit operation has an interface with input and output parameters and an implementation. A low-level difference is a data structure for the correspondences between elements of two models and for the directed delta which includes all change actions. As a result, a low-level difference contains the semantic change sets. Similar to the semantic lifting, recognition rules identify potential semantic change sets. Additionally, recognition rules are extended with checks for pre- and postconditions and trace links between edit rule objects and recognition rule objects.

After the semantic change sets have been found, the parameter retrieval phase utilizes the trace links to bind edit operation arguments to the formal parameters of the edit rule [63]. The edit operations' sequence is partwise derived from parameter dependencies. To complete the sequence, the analysis of pairs of edit rules reveals potential dependencies between edit rules which are checked for actual dependencies during the sequence deviation. The resulting order of edit operations form an executable edit script.

The approach of Kehrer et al. was evaluated with the evolution of the Eclipse Graphical Modeling Framework as case study [63]. The results indicate that an increased abstraction level is achieved. Compared to manual reverse engineered edit scripts, the approach can sometimes not detect all edit operations.

The approach of Kehrer et al. provides an executable high-level edit script from low-level changes expressed in Henshin rules [63]. In contrast, fine-grained or low-level changes are propagated in the CIPM approach. Furthermore, they are expressed in the change meta-model of Vitruvius.

6.2.3. CoWolf

Getir et al. describe the CoWolf framework for model evolution and co-evolution [32]. It includes support for

> "state charts, component diagrams and sequence diagrams as architectural models; and discrete time markov chains (DTMC), continuous time markov chains (CTMT), fault trees, layered queuing networks (LQN) as QoS models." [32]

Possible differences between coupled models and two versions of a single model are expressed in Henshin rules. CoWolf starts the co-evolution process with calculating the difference between two versions of one model using semantic lifting. Henshin rules are the result which can be applied for co-evolving the target model. Beside the model (co-)evolution, CoWolf allows the transformation of models into the language of an external solver. Such solvers can analyze attributes of the models, e. g., their performance. CoWolf was evaluated with a case study indicating reduced execution times compared to full transformations.

CoWolf utilizes Henshin rules and semantic lifting for the (co-)evolution of architecture and QoS models [32]. As a result, it does not include source code models and relies on external solvers. The CIPM approach combines Java models with the PCM acting as an architecture-level performance model.

6.3. Reverse Engineering

There are several approaches for reverse engineering software architectures and Microservice-based applications in particular. Therefore, subsection 6.3.1 covers the MiSAR approach, subsection 6.3.2 covers an approach of Rademacher et al., subsection 6.3.3 covers the MicroART approach, subsection 6.3.4 covers an approach of Mayer et al., subsection 6.3.5 covers the ArchiRev method, and subsection 6.3.6 covers an approach of Hassan et al.

6.3.1. The MiSAR Approach

Alshuqayran et al. developed the Micro Service Architecture Recovery (MiSAR) approach [1]. They conducted a study to create a meta-model for Microservice architectures and mapping rules between a Microservice-based application and the meta-model. Thus, the mapping rules can be used to recover an architecture. The study consisted of two manually performed phases: the Recovery Design phase and the Recovery Execution phase. Using one open source application, the meta-model and the mapping rules were generated in the Recovery Design phase. In the Recovery Execution phase, the meta-model and

mapping rules were refined and validated with additional seven open source applications. The resulting meta-model contains different specializations for the Microservices and deployment information.

The MiSAR approach recovers architectures specifically for Microservice-based applications from the complete source code not considering code changes [1]. In contrast, the CIPM approach extracts code changes from a commit to perform an incremental update of a component-based architecture model.

6.3.2. The Approach of Rademacher et al.

Rademacher et al. define a reverse engineering process [88] based on their viewpoint-specific modeling languages [89, 87]. Using the modeling languages, it is possible to express the domain in the Domain Data Modeling Language, Microservices and their interfaces in the Service Modeling Language, deployment and operation of Microservices in the Operation Modeling Language [89], and explicit technology aspects in the Technology Modeling Language [87]. The following reverse engineering process creates models in the corresponding modeling languages [88]. After a preparation phase including the selection of input files, the process creates the domain model, service model, and operations model in this order. At the same time, information about the technology is put into the technology model. The last phases in the process are the technical refinement for finding further technology information and the post-processing. The approach of Rademacher et al. was validated by manually applying the process on a case study.

Similar to the MiSAR approach, the approach of Rademacher et al. statically reconstructs an architecture specific for Microservices [88]. Moreover, it is not implemented while the presented approach is a prototypical implementation to incrementally update a component-based architecture model.

6.3.3. The MicroART Approach

Granchelli et al. present the MicroART approach [43]. It consists of an Architecture Recovery and Architecture Refinement phase whereby the Architecture Recovery is further divided into an Extraction, Abstraction, and Presentation phase. The extraction takes a source repository as input and performs a static analysis on it in which information about the services, their depolyment, and developers are obtained. After the static analysis, a dynamic analysis extracts container information and communication logs. The gathered data from the extraction phase is converted to an architecture expressed in the MicroART-DSL during the abstraction phase. The MicroART-DSL enables the description of Microservice architectures. It combines the Microservices with deployment and developer information and allows the clustering of Microservices. The presentation phase displays the architecture obtained from the abstraction phase. The last phase, the Architecture Refinement, allows the adjustment of the generated architecture in a semi-automated process with software architects in order to eliminate, for instance, infrastructure Microservices. The MicroART approach is prototypically implemented based on the EMF and validated by a case study.

The MicroART approach recovers architectures specific to Microservices and also proposes a process to eliminate infrastructure Microservices [43]. In contrast, the presented approach aims at detecting components within a Microservice-based application without removing a component to incrementally update a component-based architecture model.

6.3.4. The Approach of Mayer et al.

Mayer et al. introduce an approach recovering REST-based Microservice architectures conforming to a specific data model [72]. The data model is centered around services, but differentiates between services, infrastructure, and interactions between service instances and Microservice methods. The actual extraction process consists of a data collection, data aggregation, and data combination phase. The Microservices are instrumented to collect architectural information during runtime. This includes a static information extraction, e. g., to obtain the API, during the deployment of a Microservice, an infrastructure information extraction based on the configuration, and a runtime information extraction. The static and infrastructure information are directly processed in a data management phase updating the architecture. The runtime information, i. e., the logged requests, are aggregated in the data aggregation phase and incorporated into the architecture in the following data management phase. As a result, the approach allows the analysis of the architecture throughout the application's runtime. The data model and reverse engineering process were validated in a study and experiment, respectively.

The approach of Mayer et al. builds a Microservice architecture during an application's runtime, i. e., after its deployment, [72] while the CIPM approach proposes a DevOps pipeline combining information from the runtime and development of an application.

6.3.5. The ArchiRev Method

Pérez-Castillo et al. present the enterprise ARCHItecture REVersed (ArchiRev) method aiming at recovering an ArchiMate model [82]. ArchiMate is a modeling language for Enterprise Architectures (EA) proposing different dimensions in which elements are organized. The ArchiRev method applies several reverse engineering techniques on different artifacts including source code and configuration files to generate models representing specific viewpoints on the EA. The ArchiRev method is implemented in a tool allowing Java and C# code to be selected in the first step (knowledge sources). In the second step (extraction), extraction tasks are executed generating EA models based on the concrete task and reverse engineering technique. The tool, for example, creates application structures and includes static analyzers for the source code with a default set of mappings from compilation unit annotations to ArchiMate elements. The last step of the tool (EA models) visualizes the generated models. The ArchiRev tool was applied on an industrial case study to validate its feasibility.

The ArchiRev method recovers EA expressed in an ArchiMate model [82] while the presented approach concentrates on Microservice-based applications independent of their possible embedding in an EA.

6.3.6. The Approach of Hassan et al.

Hassan et al. consider the architecture reconstruction of web applications [46]. They identify static pages, active pages which are preprocessed before their delivery, web objects, multimedia objects, and databases as components of a web application. Here, web objects are defined as

> "pieces of compiled code which provide a service to the rest of the software system through a defined interface [...] supported by distributed technologies such as [...] EJB [...]." [46]

The semi-automated architecture recovery approach uses extractors on the components of a web application in order to derive facts about the components, relations, and attributes [46]. The facts conform to a schema specific to a domain, e. g., JavaScript. The common concepts of different schemas are summarized in new schemas to build layers of schemas with a rising abstraction level. The approach defines Entity-Level Schemas which describe the relations of program entities and Union Entity Level Schemas which combine different Entity-Level Schemas. The next layer is modelled by Component Level Schemas describing the relations between the components. Architecture-Level Schemas express the relationship between the architectural elements subsystem and component at the highest level of abstraction. To build subsystems, the components are clustered according to heuristics. Therefore, developers can refine the clustering. The generated facts are stored in a database and visualized by an automatic layout tool. The created layout can also be refined by a developer.

In the approach of Hassan et al., different objects, e. g., multimedia objects and databases, are considered as the components of a web application and abstracted over different layers [46]. In the context of the CIPM approach, multimedia objects as example are not considered as components and as a part of the architecture. Therefore, they are ignored during the architecture recovery. Moreover, the source code is abstracted in one layer into the PCM instead over multiple layers.

6.4. Integration of existing source code

This section covers approaches for integrating source code into Vitruvius. The following paragraph handles the Reconstructive Integration Strategy.

The Reconstructive Integration Strategy Langhammer proposes the Reconstructive Integration Strategy (RIS) for integrating a source model into Vitruvius [71]. The RIS simulates the creation of the source model by generating changes serving as the trigger for the consistency preservation process. Target model elements are created according to the CPRs. It is assumed that the order of changes has no effect on the model or CPRs. The simulation precedes an invariant resolving phase which resolves potential conflicts. Syntactical conflicts cause invalid target models and can be avoided by ensuring that invariants of the target meta-models are not violated during the consistency preservation process. Semantical conflicts prevent the creation of target models. In the RIS, the invariant

resolving phase is performed independently of the following traversal phase. During the traversal, all source elements are visited to create the corresponding changes. It starts with the root element and continues with the directly contained elements not requiring other model elements to exist. This step is repeated until all elements have been visited.

The RIS cannot make any guarantee that the simulation creates the source model as a user would have done [71]. This circumstance also applies to the presented approach. By relying on EMF Compare to generate a change sequence, conflicts are automatically omitted, and the model integrity is ensured.

6.5. Adaptive Instrumentation

Within this section, AjaxScope is introduced.

AjaxScope Kiciman et al. present AjaxScope as a platform for the instrumentation of JavaScript code [64]. The approach and its prototypical implementation realize a proxy between a web application and the user. The instrumentation allows performance analysis, runtime analysis, and usability evaluation.

An instrumentation point is an element in the JavaScript AST while a policy node is a basic unit for analysis or instrumentation, i. e., it can, e. g., rewrite an instrumentation point or filter an AST node [64]. Several policy nodes are combined in a sequential pipeline forming an instrumentation policy. Adaptive instrumentation policies are achieved by the introduction of adaptation nodes as special policy nodes. Adaptation nodes either instrument an instrumentation point and stop further processing or passes the instrumentation point to the next policy node. This decision is made by a test. AjaxScope's implementation contains an adaptive drill-down performance profiling policy. It begins with a coarse-grained instrumentation of the whole application to log timestamps. Based on this data, slower code parts can be identified which are instrumented afterwards. This step is repeated to drill down into functions to find the cause of the slowness.

AjaxScope also provides *distribution tests* as special adaptation nodes [64]. Distribution tests spread the instrumentation randomly across the instrumentation points and users. Instrumented code provides measurements for a test function checking the existence of a condition. Therefore, for every instrumentation point, a distribution test is in one of the states *pass* in which the instrumentation point reaches the next policy node, *fail* in which the instrumentation for the point is stopped and the point is not transferred to the next policy node, or *more testing* indicating further measurements are needed and enabling the randomized instrumentation. The prototypical implementation of AjaxScope is evaluated in different use cases including the drill-down performance profiling.

AjaxScope allows the adaptive instrumentation of JavaScript during the runtime distribution of the JavaScript code for various applications [64]. In contrast, the CIPM approach uses the adaptive instrumentation for Java to measure specific attributes of the code execution during its development.

7. Conclusion

The Dev-time part of the CIPM approach extracts changes from a commit to incrementally update a code model in the V-SUM of Vitruvius [73]. It triggers the consistency preservation process for a PCM instance and the extended IM. Afterwards, the source code is adaptively instrumented to monitor changed parts of the code. Then, the taken measurements are used to calibrate the PMPs to receive a valid and accurate PCM instance which allows, e. g., performance predictions.

Parts of the model update and adaptive instrumentation were prototypically implemented in previous work without forming a complete pipeline [14, 21, 75]. In this thesis, the implementations were adapted, updated, extended, combined, and evaluated with the TeaStore. The resulting pipeline checks out a specific commit to perform a component discovery and state-based model comparison with the code model in the V-SUM. The obtained fine-grained change sequence is enriched with a mechanism to detect changed methods and is propagated within the V-SUM. Next, the CPRs update the PCM and transitively propagate the changes on the PCM to the extended IM. Based on the extended IM and the code model in the V-SUM, the adaptive instrumentation is executed.

The evaluation of the approach indicates its correct operation. Nevertheless, it also revealed limitations and issues which form a base for future work. There is a gap between the implemented pipeline and the existing calibration pipeline. By combining both pipelines, the complete Dev-time part of the CIPM approach and the accuracy of such a calibrated PCM can be evaluated. In addition, it allows the repetition and further evaluation with different case studies from potentially different domains. The results and the presented approach increase the usability of the CIPM approach by automating its activities and providing a CI pipeline. Furthermore, minor issues can be resolved. This includes the detection of external calls and required roles in the context of REST APIs, the extension and evaluation of the incremental fine-grained SEFF reconstruction, and improvements in the adaptive instrumentation and SEFF reconstruction.

Bibliography

[1] N. Alshuqayran, N. Ali, and R. Evans. "Towards Micro Service Architecture Recovery: An Empirical Study". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 47–4709. DOI: 10.1109/ICSA.2018.00014.

[2] Apache Software Foundation. *14. Generating Report Dashboard*. Accessed: 13.09.2021. 2021. URL: https://jmeter.apache.org/usermanual/generating-dashboard.html.

[3] Thorsten Arendt et al. "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Model Driven Engineering Languages and Systems*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 121–135. ISBN: 978-3-642-16145-2.

[4] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *Evaluation of Novel Approaches to Software Engineering*. Ed. by Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 206–219. ISBN: 978-3-642-14819-4.

[5] A. Baabad et al. "Software Architecture Degradation in Open Source Software: A Systematic Literature Review". In: *IEEE Access* 8 (2020), pp. 173681–173709. DOI: 10.1109/ACCESS.2020.3024671.

[6] Victor R. Basili. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. Tech. rep. CS-TR-2956, UMIACS-TR-92-96. University of Maryland, Sept. 1992. URL: https://drum.lib.umd.edu/handle/1903/7538.

[7] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software* 82 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066.

[8] Petra Brosch et al. "An Introduction to Model Versioning". 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. 336–398. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_10. URL: https://doi.org/10.1007/978-3-642-30982-3_10.

[9] Pavel Bucek and Santiago Pericas-Geertsen. *JAX-RS: Java API for RESTful Web Services - Version 2.1 Final Release*. Accessed: 13.09.2021. July 13, 2017. URL: https://download.oracle.com/otn-pub/jcp/jaxrs-2_1-final-eval-spec/jaxrs-2_1-final-spec.pdf.

[10] Shing Wai Chan and Ed Burns. *Java Servlet Specification - Version 4.0*. Accessed: 13.09.2021. July 2017. URL: https://download.oracle.com/otn-pub/jcp/servlet-4-final-eval-spec/servlet-4_0_FINAL.pdf.

[11] Shing Wai Chan and Rajiv Mordani. *Java Servlet Specification - Version 3.1*. Accessed: 13.09.2021. Apr. 2013. URL: https://download.oracle.com/otn-pub/jcp/servlet-3_1-fr-eval-spec/servlet-3_1-final.pdf.

[12] *change-based-adaptive-instrumentation*. Apr. 16, 2019. URL: https://github.com/vitruv-tools/change-based-adaptive-instrumentation/tree/4c6d28cfe8d88af039fb3b52851574c

[13] E. J. Chikofsky and J. H. Cross. "Reverse engineering and design recovery: a taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17. DOI: 10.1109/52.43044.

[14] Ilia Chupakhin. *Extrahieren von Code-Änderungen aus einem Commit für kontinuierliche Integration von Leistungsmodellen*. Bachelor's Thesis. Karlsruhe, 2020.

[15] *CIPM-Pipeline*. Accessed: 13.09.2021. Apr. 6, 2021. URL: https://github.com/CIPM-tools/CIPM-Pipeline/tree/0aba63d7ed6764b524af9495380c3a48141220ea.

[16] *CIPM-Pipeline / cipm.consistency.root / cipm.consistency.runtime.pipeline.pcm / src / test / resources / teastore / models /*. Accessed: 04.09.2021. Apr. 17, 2021. URL: https://github.com/CIPM-tools/CIPM-Pipeline/tree/aaaf9d9bdaec1c99e8fcd55725a681d02d699ead/cipm.consistency.root/cipm.consistency.runtime.pipeline.pcm/src/test/resources/teastore/models.

[17] *Class ASTParser*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[18] Contributors to Jakarta RESTful Web Services. *Jakarta RESTful Web Services*. Version 3.0. Accessed: 13.09.2021. Sept. 23, 2020. URL: https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.pdf.

[19] Vittorio Cortellessa, Antinisca Di Marco, and Paola Inverardi. *Model-Based Software Performance Analysis*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 3642136206.

[20] Arlin Cuncic. *Understanding Internal and External Validity*. Accessed: 13.09.2021. July 31, 2021. URL: https://www.verywellmind.com/internal-and-external-validity-4584479.

[21] Noureddine Dahmane. "Adaptive Monitoring for Continuous Performance Model Integration". Master's Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2019.

[22] Lakshitha de Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012). Dynamic Analysis and Testing of Embedded Software, pp. 132–151. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2011.07.036. URL: https://www.sciencedirect.com/science/article/pii/S0164121211002044.

[23] Linda DeMichiel and Bill Shannon. *Java Platform, Enterprise Edition (Java EE) Specification, v7*. Apr. 5, 2013. URL: https://download.oracle.com/otn-pub/jcp/java_ee-7-fr-eval-spec/JavaEE_Platform_Spec.pdf.

[24] *Developer Guide*. URL: https://www.eclipse.org/emf/compare/documentation/latest/developer/developer-guide.html.

[25] S. G. Eick et al. "Does code decay? Assessing the evidence from change management data". English. In: *IEEE Transactions on Software Engineering* 27.1 (2001). Cited By :376, pp. 1–12. URL: www.scopus.com.

[26] Simon Eismann et al. "Microservices: A Performance Tester's Dream or Nightmare?" In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 138–149. ISBN: 9781450369916. DOI: 10.1145/3358960.3379124. URL: https://doi.org/10.1145/3358960.3379124.

[27] Brian Foote and Joseph Yoder. "Big Ball of Mud". In: (Sept. 2003).

[28] Martin Fowler. *Continuous Integration*. May 2006. URL: https://www.martinfowler.com/articles/continuousIntegration.html.

[29] Martin Fowler. *ContinuousDelivery*. Aug. 2014. URL: https://www.martinfowler.com/bliki/ContinuousDelivery.html.

[30] Martin Fowler. *The New Methodology*. Dec. 2005. URL: https://www.martinfowler.com/articles/newMethodology.html.

[31] Martin Fowler and James Lewis. *Microservices - a definition of this new architectural term*. Mar. 2014. URL: https://www.martinfowler.com/articles/microservices.html.

[32] Sinem Getir et al. "CoWolf – A Generic Framework for Multi-view Co-evolution and Evaluation of Models". In: *Theory and Practice of Model Transformations*. Ed. by Dimitris Kolovos and Manuel Wimmer. Cham: Springer International Publishing, 2015, pp. 34–40. ISBN: 978-3-319-21155-8.

[33] *Getting Started*. Accessed: 13.09.2021. Aug. 26, 2019. URL: https://github.com/DescartesResearch/TeaStore/wiki/Getting-Started.

[34] *git-log*. Version 2.33.0. Accessed: 04.09.2021. Aug. 16, 2021. URL: https://git-scm.com/docs/git-log.

[35] M. Godfrey and Eric H. S. Lee. "Secrets from the Monster: Extracting Mozilla's Software Architecture". In: 2000.

[36] Thomas Goldschmidt, Steffen Becker, and Erik Burger. "Towards a Tool-Oriented Taxonomy of View-Based Modelling". In: *Proceedings of the Modellierung 2012*. Ed. by Elmar J. Sinz and Andy Schürr. Vol. P-201. GI-Edition – Lecture Notes in Informatics (LNI). Bamberg: Gesellschaft für Informatik e.V. (GI), Mar. 2012, pp. 59–74. ISBN: 978-3-88579-295-6.

[37] James Gosling et al. *The Java Language Specification, Java SE 15 Edition*. Aug. 2020. URL: https://docs.oracle.com/javase/specs/jls/se15/jls15.pdf.

[38] James Gosling et al. *The Java Language Specification, Java SE 7 Edition*. Feb. 2013. URL: https://docs.oracle.com/javase/specs/jls/se7/jls7.pdf.

[39] James Gosling et al. *The Java Language Specification, Java SE 8 Edition*. Feb. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[40] James Gosling et al. *The Java Language Specification, Java SE 9 Edition*. Aug. 2017. URL: https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf.

[41] James Gosling et al. *The Java Language Specification, Third Edition*. Addison-Wesley, June 2005, p. 688. URL: https://docs.oracle.com/javase/specs/jls/se6/jls3.pdf.

[42] Gradle Inc. *Build Script Basics*. Accessed: 13.09.2021. 2021. URL: https://docs.gradle.org/7.0/userguide/tutorial_using_tasks.html.

[43] G. Granchelli et al. "Towards Recovering the Software Architecture of Microservice-Based Systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 46–53. DOI: 10.1109/ICSAW.2017.48.

[44] Artur Grygorian and Ionut E. Iacob. "A Concise Proof of the Triangle Inequality for the Jaccard Distance". In: *The College Mathematics Journal* 49.5 (2018), pp. 363–365. DOI: 10.1080/07468342.2018.1526020. eprint: https://doi.org/10.1080/07468342.2018.1526020. URL: https://doi.org/10.1080/07468342.2018.1526020.

[45] Jürgen Halstenberg, Bernd Pfitzinger, and Thomas Jestädt. *DevOps - Ein Überblick*. essentials 1. Springer Vieweg, Wiesbaden, 2020. ISBN: 9783658314057.

[46] A. E. Hassan and R. C. Holt. "Architecture recovery of Web applications". In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 2002, pp. 349–359. DOI: 10.1145/581380.581383.

[47] Christoph Heger et al. "Application Performance Management: State of the Art and Challenges for the Future". In: Apr. 2017, pp. 429–432. DOI: 10.1145/3030207.3053674.

[48] 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.

[49] Florian Heidenreich et al. "Derivation and Refinement of Textual Syntax for Models". In: *Model Driven Architecture - Foundations and Applications*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 114–129. ISBN: 978-3-642-02674-4.

[50] Lorin Hochstein and Mikael Lindvall. "Combating architectural degeneration: a survey". In: *Information and Software Technology* 47.10 (2005), pp. 643–656. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2004.11.005. URL: https://www.sciencedirect.com/science/article/pii/S0950584904001740.

[51] John Hutchinson, Jon Whittle, and Mark Rouncefield. "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure". In: *Science of Computer Programming* 89 (2014). Special issue on Success Stories in Model Driven Engineering, pp. 144–161. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2013.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0167642313000786.

[52] IBM. *Übersicht zu Eclipse Modeling Framework (EMF)*. June 2005. URL: https://www.ibm.com/support/knowledgecenter/de/SSQ2R2_9.5.1/org.eclipse.emf.doc/references/overview/EMF.html.

[53] Oracle Inc. *Class GenericServlet*. Accessed: 13.09.2021. 2015. URL: https://docs.oracle.com/javaee/7/api/javax/servlet/GenericServlet.html.

[54] Oracle Inc. *Class HttpServlet*. Accessed: 13.09.2021. 2015. URL: https://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpServlet.html.

[55] Oracle Inc. *Java Language and Virtual Machine Specifications*. 2020. URL: https://docs.oracle.com/javase/specs/.

[56] *Interface ICompilationUnit*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[57] Paul Jaccard. "THE DISTRIBUTION OF THE FLORA IN THE ALPINE ZONE.1". In: *New Phytologist* 11.2 (1912), pp. 37–50. DOI: https://doi.org/10.1111/j.1469-8137.1912.tb05611.x. eprint: https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.1912.tb05611.x. URL: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x.

[58] Jakarta EE Platform Team. *Jakarta EE Platform*. Version 8. Accessed: 13.09.2021. Aug. 26, 2019. URL: https://jakarta.ee/specifications/platform/8/platform-spec-8.pdf.

[59] Jakarta Servlet Team. *Jakarta Servlet Specification*. Version 5.0. Accessed: 13.09.2021. Sept. 7, 2020. URL: https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.pdf.

[60] *JavaParser*. Feb. 22, 2021. URL: https://github.com/javaparser/javaparser/tree/c579b8d1a9cb60db3babbe9384f514690fec13b8.

[61] Zhen Jiang and Ahmed E. Hassan. "A Survey on Load Testing of Large-Scale Software Systems". In: *IEEE Transactions on Software Engineering* 41 (Nov. 2015), pp. 1–1. DOI: 10.1109/TSE.2015.2445340.

[62] T. Kehrer, U. Kelter, and G. Taentzer. "A rule-based approach to the semantic lifting of model differences in the context of model versioning". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 2011, pp. 163–172. DOI: 10.1109/ASE.2011.6100050.

[63] T. Kehrer, U. Kelter, and G. Taentzer. "Consistency-preserving edit scripts in model versioning". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 191–201. DOI: 10.1109/ASE.2013.6693079.

[64] Emre Kiciman and Benjamin Livshits. "AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications". In: *ACM Trans. Web* 4.4 (Sept. 2010). ISSN: 1559-1131. DOI: 10.1145/1841909.1841910. URL: https://doi.org/10.1145/1841909.1841910.

[65] Jóakim von Kistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS '18. Milwaukee, WI, USA, Sept. 2018.

[66] Heiko Klare. *State-Based Propagation in Java Domain*. Accessed: 13.09.2021. Mar. 22, 2021. URL: https://github.com/vitruv-tools/Vitruv-Domains-ComponentBasedSystems/pull/94.

[67] Heiko Klare et al. "Enabling consistency in view-based system development — The Vitruvius approach". In: *Journal of Systems and Software* 171 (2021), p. 110815. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110815. URL: https://www.sciencedirect.com/science/article/pii/S0164121220302144.

[68] Benjamin Klatt. "Consolidation of Customized Product Copies into Software Product Lines". PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology (KIT), Oct. 2014. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043687.

[69] 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://sdqweb.ipd.kit.edu/publications/pdfs/kramer2013b.pdf.

[70] 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.

[71] 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.

[72] B. Mayer and R. Weinreich. "An Approach to Extract the Architecture of Microservice-Based Software Systems". In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 2018, pp. 21–30. DOI: 10.1109/SOSE.2018.00012.

[73] Manar Mazkatli et al. "Incremental Calibration of Architectural Performance Models with Parametric Dependencies". In: *IEEE International Conference on Software Architecture (ICSA 2020)*. 2020. DOI: 10.1109/ICSA47634.2020.00011.

[74] David Monschein. *Enabling Consistency Between Software Artefacts - Usage Documentation*. URL: https://dmonsch.github.io/dModel/.

[75] David Monschein. “Enabling Consistency between Software Artefacts for Software Adaption and Evolution”. Master’s Thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT), 2020.

[76] Object Management Group, Inc. *Object Constraint Language - Version 2.4*. Feb. 2014. URL: https://www.omg.org/spec/OCL/2.4.

[77] Object Management Group, Inc. *OMG Meta Object Facility (MOF) Core Specification - Version 2.5.1*. Oct. 2019. URL: https://www.omg.org/spec/MOF/2.5.1.

[78] Object Management Group, Inc. *OMG Unified Modeling Language (OMG UML) - Version 2.5.1*. Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1.

[79] Object Management Group, Inc. *XML Metadata Interchange (XMI) Specification - Version 2.5.1*. June 2015. URL: https://www.omg.org/spec/XMI/2.5.1/.

[80] *Package org.eclipse.jdt.core.dom*. Accessed: 17.01.2021. June 2020. URL: https://repo1.maven.org/maven2/org/eclipse/jdt/org.eclipse.jdt.doc.isv/3.14.800/org.eclipse.jdt.doc.isv-3.14.800.jar.

[81] *PCM 5.0*. Accessed: 13.09.2021. Aug. 9, 2021. URL: https://sdqweb.ipd.kit.edu/wiki/PCM_5.0.

[82] Ricardo Pérez-Castillo et al. “ArchiRev—Reverse engineering of information systems toward ArchiMate models. An industrial case study”. In: *Journal of Software: Evolution and Process* 33.2 (2021). e2314 JSME-19-0273.R2, e2314. DOI: https://doi.org/10.1002/smr.2314. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2314. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2314.

[83] Santiago Pericas-Geertsen and Marek Potociar. *JAX-RS: Java API for RESTful Web Services - Version 2.0 Final Release*. Accessed: 13.09.2021. May 22, 2013. URL: https://download.oracle.com/otn-pub/jcp/jaxrs-2_0-fr-eval-spec/jsr339-jaxrs-2.0-final-spec.pdf.

[84] Dewayne E. Perry and Alexander L. Wolf. “Foundations for the Study of Software Architecture”. In: *SIGSOFT Softw. Eng. Notes* 17.4 (Oct. 1992), pp. 40–52. ISSN: 0163-5948. DOI: 10.1145/141874.141884. URL: https://doi.org/10.1145/141874.141884.

[85] Jerry Preissler and Oliver Tigges. *Docker - Perfekte Verpackung von Microservices*. Accessed: 13.09.2021. Nov. 9, 2015. URL: https://www.innoq.com/de/articles/2015/11/docker-perfekte-verpackung-fuer-micro-services/.

[86] *PrICoBE*. Jan. 2021. URL: https://sdqweb.ipd.kit.edu/wiki/PrICoBE.

[87] F. Rademacher, S. Sachweh, and A. Zündorf. “Aspect-Oriented Modeling of Technology Heterogeneity in Microservice Architecture”. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019, pp. 21–30. DOI: 10.1109/ICSA.2019.00011.

[88] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. "A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems". In: *Enterprise, Business-Process and Information Systems Modeling*. Ed. by Selmin Nurcan et al. Cham: Springer International Publishing, 2020, pp. 311–326. ISBN: 978-3-030-49418-6.

[89] Florian Rademacher et al. "Graphical and Textual Model-Driven Microservice Development". In: *Microservices: Science and Engineering*. Ed. by Antonio Bucchiarone et al. Cham: Springer International Publishing, 2020, pp. 147–179. ISBN: 978-3-030-31646-4. DOI: 10.1007/978-3-030-31646-4_7. URL: https://doi.org/10.1007/978-3-030-31646-4_7.

[90] Red Hat, Inc. *Was ist eine REST-API und was ist REST (Representational State Transfer)?* Accessed: 13.09.2021. URL: https://www.redhat.com/de/topics/api/what-is-a-rest-api.

[91] Chris Richardson. *Introduction to Microservices*. May 2015. URL: https://www.nginx.com/blog/introduction-to-microservices/.

[92] Markus Scheidgen, Martin Smidt, and Joachim Fischer. "Creating and Analyzing Source Code Repository Models". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2017. Porto, Portugal: SCITEPRESS - Science and Technology Publications, Lda, 2017, pp. 329–336. ISBN: 9789897582103. DOI: 10.5220/0006127303290336. URL: https://doi.org/10.5220/0006127303290336.

[93] Ken Schwaber and Jeff Sutherland. *The Scrum Guide - The Definitive Guide to Scrum: The Rules of the Game*. Nov. 2020. URL: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf.

[94] *SPLevo*. Accessed: 13.09.2021. June 10, 2016. URL: https://github.com/kopl/SPLevo/tree/761370b8c3f29a6c2379395bef86777cf30b2232.

[95] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. ISBN: 9783211811061.

[96] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management*. Ed. by Jorn Bettin. 1st ed. Heidelberg: dpunkt-Verl., 2005. ISBN: 9783898643108 ; 3898643107. URL: http://www.gbv.de/dms/hbz/toc/ht014305211.pdf ; http://d-nb.info/972281509/04 ; http://zbmath.org/?q=an:1092.68027 ; http://digitale-objekte.hbz-nrw.de/webclient/DeliveryManager?pid=1862230&custom_att_2=simple_viewer ; https://www.voelter.de/books.html.

[97] C. Stringfellow et al. "Comparison of software architecture reverse engineering methods". In: *Information and Software Technology* 48.7 (2006), pp. 484–497. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2005.05.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584905000844.

[98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201745720.

[99] *TeaStore*. Accessed: 04.09.2021. Aug. 26, 2021. URL: https://github.com/DescartesResearch/TeaStore/tree/8c36e8616db4a3173cdfe3537b39d71d830c2840.

[100] The Apache Software Foundation. *Introduction to the POM*. Accessed: 13.09.2021. Sept. 11, 2021. URL: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html.

[101] Jilles van Gurp and Jan Bosch. "Design erosion: problems and causes". In: *Journal of Systems and Software* 61.2 (2002), pp. 105–119. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(01)00152-2. URL: https://www.sciencedirect.com/science/article/pii/S0164121201001522.

[102] *Vitruv*. Accessed: 13.09.2021. July 14, 2021. URL: https://github.com/vitruv-tools/Vitruv/tree/e717464171040eda6da7ed704a04e6096fa4e17d.

[103] *Vitruv-Domains-ComponentBasedSystems*. Accessed: 13.09.2021. July 14, 2021. URL: https://github.com/vitruv-tools/Vitruv-Domains-ComponentBasedSystems/tree/80e6225904ec42772cfb5657137b9fd258062b41.

[104] Sonya Voneva et al. "Optimizing Parametric Dependencies for Incremental Performance Model Extraction". In: *Software Architecture - 14th European Conference, ECSA 2020 Tracks and Workshops- The 6th International Workshop on Quality-Aware DevOps (QUDOS 2020) co-located with ECSA 2020*. 2020.

[105] *What is EMF Compare?* 2019. URL: https://www.eclipse.org/emf/compare/overview.html.

A. Appendix

A.1. Acronyms

AST abstract syntax tree

AbPP Architecture-based Performance Prediction

ADL Architecture Description Language

CMOF Complete MOF

CPR consistency preservation rule

CI Continuous Integration

CIPM Continuous Integration of Performance Models

DSL domain-specific language

EMF Eclipse Modeling Framework

EJB Enterprise Java Beans

EMOF Essential MOF

XP Extreme Programming

GQM Goal Question Metric

IMM Instrumentation Meta-Model

IM Instrumentation Model

IPMS Instrumentation Point Matching Score

JC Jaccard similarity coefficient

JDT Java Development Tools

JaMoPP Java Model Parser and Printer

MOF Meta Object Facility

MDSD model-driven software development

OCL Object Constraint Language

OMG Object Management Group, Inc.

OSM Orthographic Software Modeling

PCM Palladio Component Model

PMP performance model parameter

POJO Plain Old Java Object

RDSEFF `ResourceDemandingServiceEffectSpecifications`

REST Representational State Transfer

SEFF Service Effect Specification

SUM Single Underlying Model

JLS 3 *The Java Language Specification - Third Edition*

JLS 7 *The Java Language Specification - Java SE 7 Edition*

JLS 15 *The Java Language Specification - Java SE 15 Edition*

UML Unified Modeling Language

Vitruvius VIew-cenTRic engineering Using a VIrtual Underlying Single model

V-SUM Virtual Single Underlying Model

XMI XML Metadata Interchange

A.2. Mapping: Commit Numeration to Hash Values

Commit	Hash Value
Interval (I)	
0	77733d9c6ab6680c6cc460c631cd408a588a595c
1	9c5e7aa3bfdec546740f0dc0d1a3befc36195e8e
2	585e7e9c501506580cfda0c97e5f645ac849b46e
3	38c81c974a06a2781b19395c02905ba5f604b6c0
4	115b89c518d1f281c3132ffc8d6c3e1f6293a3f5
5	362ca4028d7407156d04e8f347e90ebd1fe26004
6	439caf08f087a2093324f7f03fa8eda3eda4e25a
7	3aaa59b5b8449b214884eb24efb7e54276563c76
8	6ecfd88ecd86f4634bc95541a9fa5e5cdaf742a4
9	3291291cc07ca420b4c6792b1d134f08b3b0c947

Commit	Hash Value
10	18ef9411e4d54db780b3d47a5fee306fb4ccd8a6
11	bd9062b294545290c265acc81ba7e6f329a6a5b1
12	907cff311827f51035788544ec6bbb27abe3b9a0
13	f8752ca8b2b3c4ebc94c7e7cb4d6cfcd34d395a3
14	c834417fbd78230dc0354c22a696bbfcd3423a7b
15	492100017f535341ed047cc8e2306654ab2becc2
16	ec94531e55463c41b6a069b24205f22c42948261
17	2fe02ea07a6f702946ca66f023d35ab6fafba91a
18	baf77779976d6265745894c31a303ace226694f4
19	1cd2c8c4bac100519b5cabafc1c83a7cd7737f49
20	79e2e244ddd014fbb9df86310b7e1e97427da196
21	ec88ca71c11bcd3699ff2e965af6cea75d0ef735
22	da98b2d0056e782448f2e0022ceb0ce54228fd67
23	554ae7f76cadf85b6dd743e97d706987226fc4f0
24	447db7f7708721fcbe8b15d93a4b166d717861d6
25	ae97a5e149b7dd4bd56bbc4e0eb71ec12ac655d3
26	c7304932eef264eadbc6ef64cf7a5b46c1f8bf7a
27	9b0dbc68e0a20d4264f0d3d7b7dccc3c17d7d76c
28	16a26135a9a756ccb066e6b963808dbba1d898e9
29	2f97af33dea1e223e346c0e6995a71cd0cdb6d33
30	26c5a920ba08f980eb9f85b156a1accacb8a4fd8
31	22c3cc238c2ce06967dce9b83fa6e2dc7a72f3be
32	413db162b63d0d55f213439ef33553c935784726
33	039df89be3fdd63ba6557835561610120303 5fee
34	f59c83cefaaf4185b3b6cce28c3a1f8260fa8265
35	0b7e58a62f3592ba8dcb0a7b981a35cacb87a38d
36	86b977d579d0bbc46826c17cbdb8251da2bcc816
37	46c8a8155df2977caebd695a5b70ae8f7eced731
38	5da47577aee96e56ea6c3349eda1f1e63e7498d9
39	b639e0d6c6aaf23d845bbd16970fb1aeac1ee99c
40	b8779e32e91534556c23a8d1f8aecae4040fddea
41	c5d937a356f5c6a0695dd9a1956d4a4137f54ec0
42	2b96fddc1edc319bac947162e8d16e710cd2bd3e
43	af1e452c86aabd33a3e4e1a70b003931eb5850ad
44	845a610cc2dc662f4fe205e61c5e5d5cb68c29b6
45	0704e1a80882dc6f161dec85f81b656bd80f73a0
46	1b49c068aa280d74095a15dff24eb3a32bb61a26
47	0d6ba6828abf3821e1d67a45e3d12cbf33fe0c31
48	a5b3f297f6c8b936aa49025599ed61f49b8ed79c
49	504aba04279ea7f6bc4f4e62a969d8f427059a03
50	53c6efa1dca64a87e536d8c5a3dcc3c12ad933b5
Interval (II)	

Commit	Hash Value
0	53c6efa1dca64a87e536d8c5a3dcc3c12ad933b5
1	a8381b6a338e2a60db6dc5b2cd71f20444f47e0d
2	4a74699f22885d8c69d579de412d5d36c2129eca
3	3ea814eeeebc90f9408d74e8c8080b0f5ce5b538
4	4289b157b3fbf02fd4cc20d4d56c5becd844883c
5	1b63f2f9ce8a5bd9c69be3fba745281d61cd3c95
6	56a53ff7d4ab5da7ab7525bd2520234d630a94b4
7	5b3921260ed32009b3dcb5ffaf1cb95d6ef2dc03
8	787a23be2345d4b3bb18fa07bde8af84e70f10ee
9	64105610d36abe100951e6225e6788e0b89d9fa0
10	7d5b672094b48d3efe8d4c2d442a41dd0ae6b1bb
11	d10092e7899a2c6b614702d1b35db859f2d7195b
12	1514911c45a3cf28dcccd9a3db671593911badb3
13	6fd7c262a1cc9745b62a89c51d8e8ef3b30185be
14	62c292941ccbcc85267889e9ffc629b5f8072159
15	f695f5ae05a1f660c17680d525ec64f7579dc9f9
16	6dc58b7cd95f8606a6bac1f1ff75a0120a9a88a5
17	6462ae1fd676f6ad21c8c603b6c5a87b3363d89d
18	d159c35d09d69e9cf364f7ec861b6d8ce9a7fa67
19	0b8fca9398f69ec929e3eacb6569b4a5f8eb058d
20	f8f13f4390f80d3dc8adb0a6167938a688ddb45e
Interval (III)	
0	f8f13f4390f80d3dc8adb0a6167938a688ddb45e
1	cf59ce3c99561fb1b83a4785305c119c720d1e73
2	624982650875d0dca231e1f8c2d76a559ffed571
3	df9eef824f74533c351036fc24b9321024d78fe0
4	d5a4464fb9eec467750b48321be89367df5263b0
5	bc322fd25c3ea2ab23c7ee45140702febb32a1f9
6	583050817139e82eb6ef99b649b2c2f09d504128
7	2e0264cedbefda1529a206e68a101764c79326ce
8	315fe19fbdd14ab96686840431035c6ba3d6d472
9	afa32739012ad86e13b4e6d76c12bc6e341d0120
10	77103bdd7d3c8c239943e24b79424315c2924f59
11	745469e55fad8a801a92b0be96dc009acbe7e3fb
Interval (VI)	
0	745469e55fad8a801a92b0be96dc009acbe7e3fb
1	900cb91e895a801ba518939b6a472d9bfc0adf49
2	13256fb5ac64f6d5a7346817d5d2b38bb963baef
3	23b0a843735dd5d6b9174266e309624ceb861ec5
4	816b98c9c5a210d41f5cd70724fd06c0d2d48833
5	1bbab532eed113d31803150bf29ef0c8b26ea556
6	e0996dcf7fa9fe7aa1a34752ff73a2300691eabe

Commit	Hash Value
7	49cb2dd58e109d6402bd8f4c3fd2bc2c352cb4f1
8	cd67bbd5497222ecdde45acd8d6e3baa9feddcc6
9	bfd2b36394e9453e7020c7906de07e667776ccac
10	ad6c8557c81f0dc0601508c49ff7e0af6fe9e0fa
11	0d32426425692444ef33e26fe9a99364e88780ab
12	6843c00be99c0666195ac577177fbf956e7e332c
13	23e5f18ef1f27ed0ab36915a11d46c4945dbd03b
14	b42d44b84e49fd1039db68d391fc28d03deff2af
15	d9e1cdca5adbde0cbe28c5d15bf298fbb4815319
16	882c0481929c7f628042c9f5103165bedbe76b6f
17	7d5449f27518baaaa77f37401c82e3376efcb810
18	834d84d033a81aa6e5934a7a46f65c2a8a2f7e57
19	e1d134bd4f27754a94e5a3e96831d5bd50709a80
20	3dc265add6d18f5a2080e8b1def919d87877bfe5
21	d6f80444b18a49a75b805edd80ac8ef128d0a560
22	b0ecb45238772f06db1e11e8e7baaa72f48cdd96
23	b59562c3ea236fbd59d8361397fc7676bda256a1
24	bb6a24e0082436660055f1fd9ea11f4bafdbcf1f
25	8e182111dce10d0f84fa8b775b5a5f72c1c00b14
26	8b9c3c4b61135bbe7a2fa340913aa95472200396
27	d2b50bf17c9ea4f6d48f2511f351820c63c0053a
28	40fc2388eadd1a0194c765abc450ff10f6156dae
29	f46215c1ee9a2839e644dd69c8ca099626a2861f
30	1973d02a7a15565b368a50f865588f2e67751e75
31	00f9142a19103f063fa7da48c29d8978d5a4352d
32	3d040638f35d890808b23a846b1cc61426f96249
33	a96dd569448292fd5dd83e28eb0b303e3a0c5562
34	e9dadd5e84753e4ac832d12eaf376ef51507c903
35	2cca63aad1db3e92af334fca79d786386d5ea3d5
36	b0d62482cbd113ad1d61c374baf37031c4c5d867
37	85372b809d883c1efba4288f484e093fe49d5f61
38	815f4f64ff3d73d1211e44b6d099be4248266136
39	ffcbe96c8a2a307f96db1da4fde1d1a2ad80999c
40	0f19bc5dea0b73b623e378104abf7d73de453481
41	1da27c65d47b38d6ea3cc10bb247ff7e3c69c17f
42	d278bb55ea23f40325edfa6244a6abd6ce7a9e4e
43	014712685ae4033aa125bce7fcadc4c39ccb7c24
44	4c48631b69b9597c7cc83e16b596a4fa72bb58f2
45	57df9652d55643b0af5d4a097aabf4138b8ac1cb
46	1cbc324e1640a207032922858a9e56ea92aba779
47	bdcbb8ce1e28c8f9b708c79fe97f1d1bee4ced59
48	0bf9aa78024adc2631950ea0004b5c561d2bf5e5

Commit	Hash Value
49	ea62e166bc38c1ddbf76c155072aac1c5ee56c89
50	2d422b9abdd27f810eee48aa25d30b21233ef06c
51	653528e387a4fb764e48c9422ab218c65f87082f
52	0ce0cababe2590eff6d03c15f9eb3c977fdf4914
53	df597ccdd0e16f651f0113e04d014320573fee76
54	03c1f1c3a179dde5c3cc33bcf0edf60a0b49ea52
55	032167ddc872887fa1eb4c1fe2df8209a0bd7b76
56	19662c0afc9e8005917e8eba42d9b59592d0feeb
57	4cd9e06869ab671bee1d11b7abc3eebf9b582f35
58	a7727800a9879c96596e56d320420dc4f1bd8a2c
59	3e17bfbb8b39a1eba260abb9085969fb30fdb9d1
60	e0ffa6a1d477c1e1cc65f8eeb1c3bcabb6f6f5da
61	cf75936549548faa52e37c098aa16c24dde8a067
62	6d334d025b2133a92fee8d645b078f109e65f5cf
63	806e5cc946f7995256d0da40492a2050c74e5ada
64	12efccf95b372d0df94075e7797ea4b96c9b5c88
65	79330c1f9c9efc2118b018805f4c629d6096b9b9
66	9f0ec365de1cb438904530f2b90e65d9d296d6ef
67	05ade1575d5fb2276ed4e2ac0f272dbd0533b141
68	c4860ee3392cfb00e83a824b1e9c7b991e10c318
69	6d222bf82dab2e192ab7820f3d5619875b80bd31
70	8f9a8eeb2ed6f1486b5286e8dfed61ab279566ca
71	b58219ac535670f3e63700289abe8a6ce0d7dcf0
72	790452f95a9d51803afd26b36788957a0cca2c4d
73	3c674e12622a585faa39334eeca3daff98fddc80
74	637ec65a89eef33aabee8ca031d283f0e2db02b9
75	0344db1d3621df6bb5a6787973f4d9db599b480b
76	b7c4e9df95668509f7058d2fa32157229c795fc9
77	c43f0cc67b88a1f5b982c380cc6dc5b52686f471
78	87c32b70ff19df2f32f899cd659741ae60e9ac6b
79	07106148f2cedb12c01525b1586127eadefef986
80	0ffdf356e82b2e26f93a828ed4f378c505bd68f2
81	bdde10b80d195d08ed7e3617107de006210d753a
82	10f049d663e523873f27e6a794635572a975e2a5
83	a349ddeb32c58d49376cf8d165908312b5a3405b
84	e29360bd1698a6cd2f1f28a9aa6282276f582e58
85	02877a14aeaaaa9cc1365dafafeb85d8f3387e9b
86	045b1a917ac70fec559a2548e14977acc09dd76a
87	98570ca8ac398dc4d1f6e944b895bccb39726867
88	df33017194634db70c4e786b64cc1d6b386fe676
89	a543c026e9563f50ff6371912d470ec30f261ae3
90	65971f539d55fff3444a02bc31f6d3d9b4f26813

Commit	Hash Value
91	9a7ba550cd47ba4b7d9a652814d337f2f1f6d79b
92	6f70c3feb0640bf9d8691acdcf2dccae1710cbf0
93	febeaa537041ef7b23e479ec958586c2154e9ab4
94	d16000d2a74a299d9722c840434bedf109c04198
95	30675d11c9267eb2aee3d04e73b28a13a4eab8df
96	1a11ca0e48d59f665d30a6303ed49d14cafa13c5
97	a5f12f3bbb8a1c58c257ec0b49667cac9fca94a3
98	2c25727b02222308b21399bcd8266f732bbbe41d
99	49e2437abf782b3a618fe5ed155a2c8469557e47
100	de69e957597d20d4be17fc7db2a0aa2fb3a414f7
Experiment E6	
0	de69e957597d20d4be17fc7db2a0aa2fb3a414f7
1	8c3945ea1c83fc5cb31515818eaa48df03f694d9
2	015be98b282a047f1d905a26a5437446f13041ce

Table A.1.: Mapping of the used commit numeration within the thesis to the commit's hash values [99].