Karlsruhe Institute of Technology

Continuous Integration of Performance Models for Lua-Based Sensor Applications

Master Thesis of

Lukas Burgey

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. Martin Armbruster

8. August 2022 – 14. April 2023

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, 14.4.2023

. . . .

(Lukas Burgey)

Abstract

The availability of architecture-level models of software can aid the development of the software by preventing architecture degradation and easing maintenance of the software. Further, architecture-based performance prediction (AbPP) can be used to diagnose performance issues during the implementation of software. Architecture-level performance models, like the Palladio Component Model (PCM), can realise both these benefits. Previously, manual intervention was required to create and repeatedly update such models during the development of software. This tedious process hinders the wider adoption of such procedures in practice. Approaches that can automatically create, update and calibrate instances of e.g. the PCM during the development of software are required to tap these potential benefits of AbPP.

The Continuous Integration of Performance Models (CIPM) approach can be employed to automatically make a calibrated architecture-level performance model, the PCM, available during the development of software. The approach leverages the Vitruvius framework, the Co-Evolution approach, and adaptive instrumentation to achieve the automatic updating of the PCM. A prototypical implementation of the CIPM approach targets microservice-based web applications implemented in the Java programming language. No implementations for other programming languages exist and the process of adapting the CIPM approach to support another programming language has previously not been explored.

In this thesis we present an approach to adapting the CIPM approach to support Lua-based sensor applications. We contribute a code model for the Lua language based on an Xtext grammar. The code model is integrated into the CIPM approach by implementing Consistency Preservation Rules of the Vitruvius framework. We developed a partial prototypical implementation of the adapted approach. The implementation of the adaptive instrumentation which is required to complete the implementation of the approach was omitted.

The prototypical implementation was evaluated using real-world Lua-based sensor applications from the SICK AppSpace ecosystem. While the evaluation demonstrates the feasibility of the adapted approach, it also reveals minor technical issues with the prototypical implementation. Future work is required to complete the prototypical implementation and address the remaining technical issues and limitations.

Zusammenfassung

Die Verfügbarkeit von Softwaremodellen auf Architekturebene kann die Entwicklung von Software unterstützen, indem sie eine Verschlechterung der Architektur verhindert und die Wartung der Software erleichtert. Außerdem kann die architekturbasierte Leistungsvorhersage (engl. architecture-based performance prediction (AbPP)) verwendet werden, um Leistungsprobleme während der Implementierung von Software zu diagnostizieren. Leistungsmodelle auf Architekturebene, wie das Palladio Komponenten Modell (engl. Palladio Component Model (PCM)), können diese beiden Vorteile realisieren. Bisher war ein manueller Eingriff erforderlich um solche Modelle während der Softwareentwicklung zu erstellen und wiederholt zu aktualisieren. Dieser langwierige Prozess behindert die breitere Anwendung solcher Verfahren in der Praxis. Ansätze die Instanzen solcher Modelle während der Softwareentwicklung automatisch erstellen, aktualisieren und kalibrieren können sind erforderlich, um die potenziellen Vorteile von architekturbasierter Leistungsvorhersage zu realisieren.

Der Ansatz der kontinuierlichen Integration von Leistungsmodellen (engl. Continuous Integration of Performance Models (CIPM)) kann eingesetzt werden, um ein kalibriertes Leistungsmodell auf Architekturebene, das PCM, automatisch während der Softwareentwicklung zur Verfügung zu stellen. Der Ansatz nutzt das Vitruvius-Framework, den darauf aufbauenden Co-Evolution-Ansatz und adaptiven Instrumentierung, um die automatische Aktualisierung des PCM zu erreichen. Es existiert eine prototypische Implementierung des CIPM-Ansatzes für auf Microservices basierten Webanwendungen, die in der Programmiersprache Java implementiert sind. Es gibt keine Implementierungen für andere Programmiersprachen, und der Anpassungsprozess des CIPM-Ansatzes um eine andere Programmiersprache zu unterstützen, wurde bisher nicht erforscht.

Wir präsentieren in dieser Arbeit einen Ansatz zur Anpassung des CIPM-Ansatzes zur Unterstützung von Lua-basierten Sensoranwendungen. Wir entwickeln ein Codemodell für die Lua-Programmiersprache, das auf einer Xtext-Grammatik basiert. Das Codemodell wird in den CIPM-Ansatz integriert, indem so genannte Konsistenzerhaltungsregeln des Vitruvius-Frameworks implementiert wurden. Wir haben eine partielle prototypische Implementierung des angepassten Ansatzes erstellt. Aus Zeitgründen mussten wir die Implementierung der adaptiven Instrumentierung weggelassen, welche für die vollständige Implementierung des Ansatzes erforderlich ist. Die prototypische Implementierung wurde mit realen Lua-basierten Sensoranwendungen aus dem SICK-AppSpace-Ökosystem evaluiert. Die Evaluation zeigt die machbarkeit des angepassten Ansatzes. Zudem deckte die Evaluation kleinere technische Probleme mit der prototypischen Implementierung auf. Weiterer Forschungsbedarf besteht, um die prototypische Implementierung zu vervollständigen, die verbleibenden technischen Probleme und Einschränkungen zu beheben.

Contents

List of Figures

List of Tables

1. Introduction

Managing the performance of software and especially the degradation of software performance is a challenging endeavour. The development of software can benefit significantly from systematic performance management approaches. Both a-priori (e.g. architecture-based performance prediction (AbPP)) and a-posteriori (e.g. performance testing) approaches to performance management exist. AbPP usually has no need to implement a performance prototype and is therefore usable for software that is developed according to the procedures of agile software development, which has become popular in the community [5].

Smart and programmable sensor hardware, the behaviour of which can be customized using software, has become available. In contrast to commodity hardware there sensor nodes have limited capabilities. Further, the deployed hardware of such sensors is more heterogeneous. Regarding the development of sensor applications, AbPP can therefore aid answering the following questions.

- How would the relocation of sensor data processing into the cloud affect the extra-functional aspects of the application?
- Is it worthwhile to do pre-filtering of sensor output on a less capable sensor node?
- What performance is expected from using more affordable hardware for the compute nodes of the network?

Architecture-level performance models, like the Palladio Component Model (PCM), can be used for AbPP [6]. In addition the availability of architecture-level models of software can aid the development of the software by preventing architecture degradation and easing maintenance of the software. Previously, manual intervention was required to create and repeatedly update models as the PCM during the development of software. This lengthy and tedious process has prevented the adoption of such techniques. Approaches that can automatically create, update and calibrate instances of e.g. the PCM during the development of software are required to tap these potential benefits of AbPP.

The Continuous Integration of Performance Models (CIPM) approach can be employed to automatically make a calibrated architecture-level performance model, the PCM, available during the development of software [31, 6]. In addition to the PCM the CIPM approach uses a model of the source code of the application and an instrumentation model. The approach uses the Vitruvius framework to keep the the three models consistent with each other. When the source code changes the code model of the Vitruvius framework is updated and changes to it are propagated to the other models. To limit the monitoring overhead adaptive instrumentation is used: Only changed parts of the software are instrumented to calibrate the PCM after changes were made to the source code. This is achieved using the instrumentation model. The approach is implemented in the form of two prototypical implementations: The first implementation contributed the monitoring of the instrumentation points of the instrumentation model and the subsequent calibration of the PCM [33]. The second prototypical implementation of the CIPM implements the rest of the approach in form of a continuous integration pipeline: It targets microservice-based web applications implemented in the Java programming language. No implementations for other programming languages exist and the process of adapting the CIPM approach to support another programming language has not been previously explored.

In this thesis we present an approach to extending the CIPM approach with support for Lua-based sensor applications. We contribute a code model for the Lua language based on an Xtext grammar [50]. For this code model a matching is implemented, so changes to the code model can be derived by the approach. The code model is integrated into the CIPM approach by implementing Consistency Preservation Rules (CPRs) of the Vitruvius framework. We developed a prototypical implementation of the adapted approach. Because of the time constraints of this thesis we could not implement the adaptive instrumentation which is required to complete the implementation of the approach. For the evaluation of the prototypical implementation the SICK AppSpace ecosystem of sensor applications implemented using the Lua programming language is used. We evaluated the prototypical implementation using both a minimal running application composed of sample applications of the SICK AppSpace and a real-world sensor applications for the detection and sorting of objects using their color. We demonstrate the feasibility of the adapted approach using an evaluation based on a Goal Question Metric (GQM) plan. Still, minor technical issues with the prototypical implementation remain and could be addressed by future work. Further, future work is required to complete the prototypical implementation and to address limitations of the approach, like the omission of modelling of types.

The structure of this thesis is as follows. The foundations are laid out in chapter 2. An overview over the approach structured according to the PRICoBE principle can be found in chapter 3 [35]. The CIPM approach for Lua-based sensor applications and its prototypical implementation are presented in chapter 4. The evaluation of the prototypical implementation which is based on a GQM plan is discussed in chapter 5. Related work is presented in chapter 6. Finally, this thesis is concluded and possible future work is outlined in chapter 7.

2. Foundations

This chapter outlines the foundations of the presented approach.

The popularity of agile software development is the foundational motivation behind the presented approach. Agile software development is introduced in section 2.1. The basic concepts of software modeling and Model-Driven Software Development (MDSD) are introduced in section 2.2. The Eclipse Modeling framework, which serves as the technological foundation for the modeling activities of the approach is presented in subsection 2.2.2. Xtext, a project for textual modeling is introduced in section 2.3.

The Vitrivius approach to keeping software models consistent is presented in section 2.4. The PCM, a meta-model for amongst other things the architecture and performance software models is presented in section 2.5. Building further on Vitrivius, the Co-Evolution approach is introduced in section 2.6. The Continuous Integration of Performance Models approach, which the presented approach is based upon is presented it section 2.7. The presented approach extends the CIPM approach with support for applications written in the Lua programming language, which is outlined in section 2.8. The approach we will present in this thesis is evaluated using Lua applications from the SICK AppSpace ecosystem, which is introduced in section 2.9.

2.1. Agile Software Development

Agile software development was popularized in the 2001 *Agile Manifesto* [5]. The manifestos principles aim for software development with shorter release cycles even down to a couple of weeks [5]. The principles further include "[Welcoming] changing requirements"[5]. Changed requirements usually lead to changes of the software architecture. Redesigning software architecture is a challenging endeavour, especially in a short time frame. Applying agile software development in practice is aided by further practices, which will be outlined below.

Continuous Integration (CI) is the practice of frequently integrating the developer changes. The term was first used by Grady Booch in 1991 [7]. A typical frequency for the integration is daily integration. The practice is used to keep the development team in synchronisation, which is required when the software is incrementally developed like with agile software development.

Continuous Delivery (CD) extends the Continuous Integration with delivering the integrated software regularly. This is usually achieved through pipelines, which are often called CI/CD pipelines. Changes by the developers are checked into a Version Control System (VCS), like Git [14]. When the changes are pushed to the server the pipeline can automatically build the software, run the test suite, deliver build artifacts or even deploy the software in a live server. The automation aids the developers with the frequent releases of agile software development.

2.2. Model-Driven Software Development

This section will introduce basic concepts of software modelling. The Model-Driven Architecture (MDA) is introduced in subsection 2.2.1. Methods for finding the differences between models can be found in subsection 4.5.5.

Model-Driven Software Development is a term for techniques to develop software using or driven-by models.

Models have three properties according to Stachowiaks general model theory [42]: They are always models or representations of *something*, the mapping property. The reduction property defines that models in general do not reproduce every aspect of the thing they are representing. Finally, according to the pragmatic property models may be ambiguously assigned to their originals and they fulfill their replacement function only for certain subjects, time frames and mental or actual operations. In the case of models that describe other models we speak of meta models, or models *about* models.

The goals of model driven software development are to decrease the development time of software and to increase the quality of the software [43]. By separating different concerns of software during its implementation, the maintainability and adaptability should be eased. In addition code redundancy can be decreased and code reuse should be can be increased.

2.2.1. Model-Driven Architecture

The Object Management Group (OMG) is a standards consortium in the field of the computer industry [15]. MDA is a certain procedure for the model-driven software development which is standardized by the OMG [8]. The goal of MDA is the interoperability and portability of models and tools. MDA bundles other OMG standards, such as the Meta-Object Facility (MOF), a meta meta model [32] that defines how meta models within MDA are structured. This assures that the models and meta models or *modelling languages* of MDA are interoperable. Other standards of MDA include XML Metadata Interchange (XMI), an file format for the exchange and persistence of the models based on Extensible Markup Language (XML). XML is a text file format also described as a *metamarkup language* [16], which means that XML files do not contain a fixed set of tags but instead one can flexibly define the available tags through extension.

2.2.2. Eclipse Modeling Framework

The Eclipse Integrated development environment (IDE) is an extensive software development platform [11]. Eclipse is based on a plugin architecture, which makes a large amount of eclipses functionality extensible and customizable. While eclipse is especially popular with programming languages of the Java ecosystem, it supports a very large amount of programming languages and many customized versions of the eclipse IDE for specific use cases are available. One of these are the *Eclipse Modeling Tools*, which is intended for use with projects using practices like the previously introduced MDSD.

The modeling tools are build on an extensive collection of software for MDSD, the Eclipse Modeling Framework. Eclipse Modeling Framework (EMF) project provides facilities for modeling various kinds of data and information [12]. Central to the EMF project is the *Ecore* meta meta model, which is an implementation of OMGs Essential Meta-Object Facility (EMOF), a subset of the previously introduced MOF. Viewers and editors are available for the Ecore based models. E.g. ecore model can be edited in the eclipse IDE. Persistent information is modelled using Ecore Resources. Ecore Resources can be serialized and deserialized using the previously introduced XMI format. The EMF includes facilities for Java code generation for its on meta models. This allows the programmatic traversal and manipulation of the models in Java applications. Further, EMF has validation facilities that can be used to automatically determine if a model is fitting its meta model.

The sub project EMF Compare can be used to find the differences of or merge two versions of a models. EMF Compare allows both two-way and three way (two models with a common ancestor) model merging. Methods for deriving changes in models using EMF Compare are described in the next section.

2.2.3. Change Derivation of Models

An important aspect of model-based software engineering (MBSE) the evolution of the software models. The differences or changes to a software models are often used in existing approaches, like the Vitruvius approach, as a basis for further action [48].

Models may be changed incrementally by developers or architects, which of course is trivial, as one can just record the atomic changes the operator makes. A more challenging case are models that are generated automatically based on their original, an example being a source code model that is generated by parsing a version of the source code. A changed original may cause non atomic changes to the model. In this case state based change resolution approaches can be used to find the difference between the old and the new version of the model.

As previously introduced EMF Compare can be used for finding the differences between two Ecore models [13]. In addition EMF compare can be used to merge two Ecore models into a single model. The project provides default implementations for match engines, that match individual elements of the models with each other. Depending on the used model it may be necessary to implement a model specific match engine for EMF Compare to be able to derive the differences between the models correctly.

To obtain a change sequence between an old and a new version of a model one first determines the differences of the two model versions. These differences are then applied to the old model version, while changes to the underlying are recorded by using EMF adapters. The recorded changes to the old version are the change sequence from the old model version to the new. This approach is used by the Vitruvius approach [21].

2.3. Xtext Framework

Xtext is a framework for textual modeling [50]. As such it can be used to develop Domain-Specific Languages (DSLs), Domain-Specific Modelling Languages (DSMLs) and even general purpose programming languages [50]. Xtext uses Ecore-based models to integrate the textual models with the wider EMF ecosystem. The project can operate in two modes: If an Ecore model already exists and a textual model shall be defined, one must implement an Xtext grammar, that is later used parse and create the textual models. If no Ecore model exists, one defines the textual model by again implementing an Xtext grammar. Xtext can then infer an Ecore model from the production rules of the grammar.

Xtext is integrated into the Eclipse IDE and a language or textual model implemented with Xtext has automatic support in Eclipse. This includes syntax highlighting, validation of code and automatic code generation. Xtext uses ANTLR 3 to generate a parser for the implemented language [34, 1]. The parser can parse files of the language into an model and persist them as Ecore resources [51]. To facilitate this Xtext generates an Ecore language meta-model for the programming language from the Xtext grammar. Consequently the models which are obtained by the parser can be used by any project using Ecore models [12]. This allows the handling of the parsed code model by projects like Vitruvius, which we will discuss later in this section.

2.4. Vitruvius: View-Centric Engineering using a Virtual Single Underlying Model

Vitruvius is a framework for view based software development [21]. In the context of MBSE software is usually described using multiple models, which describe a certain aspect of the software. Keeping the various models consistent is a challenging endeavour. The data of the underlying models is available through views which can aggregate the information spread to multiple models.

Orthographic Software Modeling (OSM) is a view-based modeling approach that models different aspects or dimensions of software using different views [3]. They employ a Single Underlying Model (SUM) to contain all information about the modelled entities [4]. In addition the derive separate view models for different perspectives on the overall model. Vitruvius extends this notion to a Virtual Single Underlying Model (V-SUM) [28]. In order for the larger V-SUM to keep its validity, the various models within the Vitruv V-SUM must be kept consistent, in case on of its models is changed. The changes in one model must be "semantically" also made in the other models. For this the Vitruvius approach keeps using so-called CPRs, which we will discuss later in this section.

To aid the consistency preservation, Vitruvius uses a correspondence model to associate model elements of different models with each other. For example a function element of a code model may correspond to its representation in an architectural model.

So called CPRs are used to keep multiple models of a Vitruvius V-SUM consistent. The rules are defined using two DSLs: the *reactions* language and formerly the *mappings* language. For the purpose of this thesis we focus on the reactions language.

Using the reaction language it is possible to define how changes in one single model of the V-SUM can be "transferred" to other models of the V-SUM. An example would be how changes in the code model of a software in development can be transferred to an architectural model. The reactions language allows the use of two kinds of statements: Reactions itself and routines. Reactions have a trigger, which specifies to what changes in the first model changes need to be made in the second model. Examples for triggers, are the creation or deletion of model elements, and attribute changes of model elements. In addition reactions contain instructions that are executed when the reaction triggers. These instructions are implemented using Xtend a language that transpiles to the Java language. From these instructions routines can be called.

Routines are extended functions for the manipulation and traversal of the Virtual Single Underlying Model (VSUM). They have a match block that can be used to retrieve corresponding model elements of the function arguments. If a model element has no correspondence the routine may not be executed. Further routines can be used to create model elements in the second model. Again arbitrary Xtend code can be used to manipulate the models so consistency is preserved.

2.5. Palladio Component Model

This section introduces the Palladio Component Model (PCM), which is employed in the CIPM approach as meta-model as the architecural performance model. PCM is a meta-model for the performance prediction of component based software architectures [6]. The PCM was conceived as part of the Palladio approach [6].

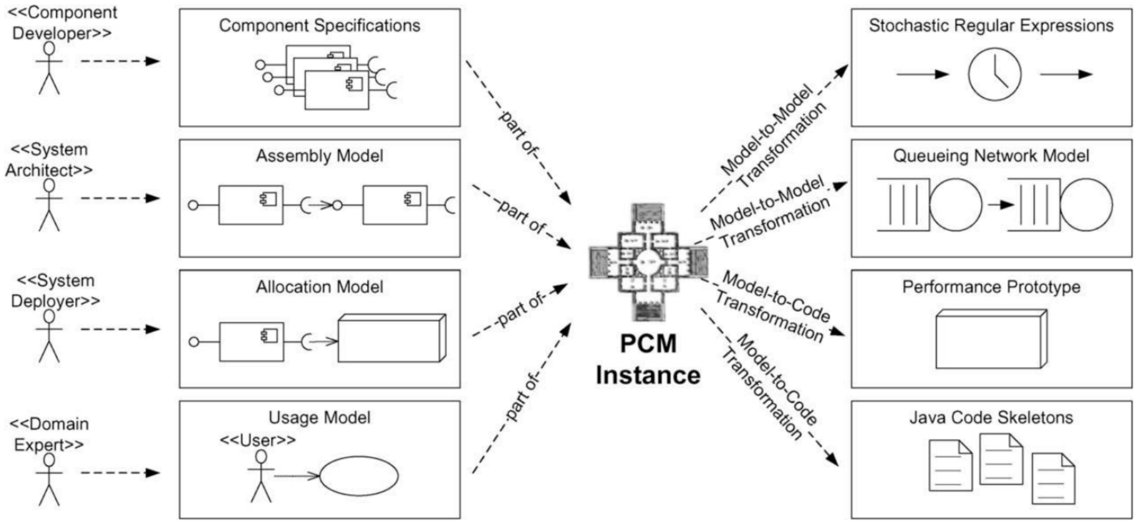


Figure 2.1.: Overview of the Palladio approach[6].

An overview over the complete palladio process can be seen in Figure 2.1. The actors of four different roles (component developer, system architect, system deployer and domain experts) can contribute to the development of a corresponding model instance. The component specifications are modelled using Repository models. An overview over the

Repository meta model is depicted in Figure 2.2. The central class of the meta model is the Repository. It is composed of an arbitrary amount of Interfaces and BasicComponents. The functionality provided by components through its Application Programmable Interface (API) are referred to as services. Interfaces in turn are composed of Signatures, which describe the functions or services of an interface. They include the service name and contain further model the parameters a function takes (omitted in the figure). The BasicComponents are composed of Service Effect Specification, that model the behaviour and resource demand of the services provided by the component. The structure of Service Effect Specifications (SEFFs) is further discussed in subsection 2.5.1.

2.5.1. Service Effect Specifications

SEFFs or more precisely Resource Demanding SEFF (RDSEFF) model the behaviour of component services of the PCM. An example pair of source code and the corresponding RDSEFF can be seen in Figure 2.3.

SEFFs have an associated signature and contain multiple actions, their *step behaviour*. The step behaviour starts with a StartAction and ends with a StopAction. The control flow of the function between the StartAction and StopAction is modelled using ForkActions and LoopActions and BranchActions. Other behaviour besides control flow is modelled using the following Actions.

- ExternalCallActions are used to express invocations of services of other components. They reference the signature of the called service.
- InternalCallActions model calls to functionality of a component that is not modelled as a service. Because of this InternalCallActions contain additional step behaviour for this functionality.
- InternalActions represent any remaining resource demanding behaviour.

The actions can have resource demanding (RD) behaviour, e.g., an action could take 1000 CPU cycles to process. The resource demanding behaviour can be modelled as constants, probability distributions, and functions of random variables [6]. Using the resource demands of the actions of a service the its actual resource demand can subsequently be simulated. This enables the analysis of qualitative aspects of the software like performance measures. A prominent performance measure in this context is response time.

2.5.2. SEFF Reconstruction

This subsection will introduce previous approaches to reverse engineer SEFFs.

2.5.2.1. Static SEFF Reconstruction

Krogmann presented an approach to static reverse engineering of RDSEFFs [25]. The approach uses static control flow analysis to reconstruct a SEFF for a software method.

Initially all relevant calls by a given definition are identified and marked. Starting from all the marked calls, all control flow statements, like loops and branches are transitively marked beginning with the marked call and terminating at the root of the reconstructed method. Using all the marked calls and statements the abstract control flow of the method can be reconstructed and a RDSEFF with this control flow can be created for the method.

2.5.2.2. Change-Driven SEFF Reconstruction

Langhammer extended the previously presented static SEFF reconstruction with an approach to a change-driven incremental SEFF reconstruction [28]. This approach is further integrated with the Co-Evolution approach. Exported methods have a corresponding SEFF through the Vitruv correspondence model. If changes are made to methods that have a corresponding SEFF, a new SEFF is reconstructed for the method and replaces the old SEFF in the correspondence.

Langhammer maps the marked calls to abstract actions of the PCM. External calls are mapped to ExternalCallActions, library calls and internal calls are mapped to InternalActions. For this mapping specific external/library/etc. call finders are implemented [28]. Every method gets StartAction and a StopAction, before and after the reconstructed control flow. For external calls the called signatures and required role are determined by mapping-specific finders.

A potential down side of the approach is that the old SEFF of a changed method is not reused.

2.5.2.3. Fine-Grained Incremental SEFF Reconstruction

Dahmane proposed an approach for incremental fine-grained SEFF reconstruction [9]. The process is referred to as incremental as the old SEFF is reused.

If changes in a method corresponding to a SEFF are detected, a new SEFF for the method is reconstructed using the previously described method. The new SEFF is matched with the old SEFF. Elements of the seff are considered equal if they reference the same source code elements. Unmatched elements of the old SEFF indicated deletions in the method, which consequently are deleted from the SEFF. Unmatched elements of the new SEFF are changes to the method. These will be included in the resulting SEFF.

The evaluation of the approach showed that loop actions could not be correctly matched because comparing the statements of a loop action was not possible.

2.6. The Co-Evolution Approach

Langhammer presented the Co-Evolution approach, which can be employed to automatically keep consistency between source code and a PCM instance [27]. The Co-Evolution approach was prototypically implemented for Java and is embedded in the Vitruvius approach. The Java Model Parser Printer (JaMoPP) is used to parse Java source code into an Ecore-based Code Model (CM) [17]. In addition, JaMoPP can print a CM as source code. Langhammer contributed multiple technology-specific CPRs. For example CPRs that can keep the consistency between Java source code based on Enterprise Java Beans and a PCM instance were contributed.

2.7. Continuous Integration of Performance Models

The CIPM approach aims to make up-to-date architectural software models, like the PCM, available during the complete development and operation of software [31]. The approach presents a complete model-based DevOps pipeline, which makes the adoption of the approach in the context of agile software development (see section 2.1) feasible.

Generating an architectural performance model from scratch and calibrating it with performance model parameters (PMPs) can be prohibitively time consuming. The approach employs two mechanisms to achieve more efficient updating and calibration of the architectural performance models:

- Incremental model updating inspired by and reusing parts of the Co-Evolution approach as presented in section 2.6.
- Adaptive instrumentation (see subsection 2.7.4)

An example of the performance model update workflow is presented in subsection 2.7.2. Details regarding the prototypical implementation of the CIPM approach can be found in subsection 2.7.5.

2.7.1. Commit-Based Continuous Integration of Performance Models

Armbruster presented an approach to commit-based Continuous Integration of Performance Models [2]. They addressed open issues with the approach and presented an complete commit integration pipeline by extending the prototypical implementation of the CIPM approach. This prototypical implementation is further discussed in subsection 2.7.5 as it is the basis for the prototypical implementation of the approach presented in this thesis.

Armbruster further evaluated the CIPM approach using TeaStore, a microservice based Java web application designed for benchmarking and testing [49]. They found that the approach operated correctly, albeit with several limitations.

2.7.2. Model Update Process

The process of updating an architectural performance model is depicted in Figure 2.4. Initially a software developer makes changes to the software, commits them and pushes them to a repository. The DevOps pipeline triggers the depicted update process. A parser parses the source code into a CM. A change sequence is derived by comparing the new CM with the code model that resides in the Vitruv VSUM. This change sequence is propagated into the VSUM, which also triggers CPRs of the architectural model. The implemented CPRs react to the changes and further propagate the changes to the architectural performance model, e.g., an instance of the PCM. The CPRs further populate the Instrumentation Model (IM) with information which parts of the code were changed and consequently need to be instrumented now. Using the updated CM and IM, the source code is adaptively instrumented (see again subsection 2.7.4). The instrumented code is executed and monitored. This yields PMPs which are fed back to the architectural performance model. Examples for such PMPs are branch probabilities, resource demands of actions and loop iteration numbers [31]. The performance model is now calibrated and can be used for AbPP of the software.

2.7.3. Instrumentation Meta-Model

The Ecore-based Instrumentation Meta-Model (IMM) was originally presented by Dahmane [9]. An instrumentation model as defined by the IMM describes, which PCM services of an application must be instrumented and monitored so all the needed PMPs for the calibration of the respective PCM. As the original IMM operated an the service level, changes within the behaviour of a service would cause the complete behaviour of the service to get instrumented. This characteristic is sub-optimal in terms of monitoring overhead.

To combat this, Monschein extended the IMM [33]. In addition to the modelled information of the original IMM, the extended IMM also describes, which actions of a PCM service were changed. This increase in granularity, permits a more precise instrumentation. Changes within a service behaviour will only result in the instrumentation of the changed parts of the service (based on the information of the extended IMM).

The extended IMM is depicted in Figure 2.5. The central element of the extended IMM is the instrumentation model. The model has service instrumentation points, which in turn have of action instrumentation points. Both instrumentation point types have references to their corresponding service or action. In addition the activity state of an instrumentation point is tracked using a `active` variable. Further, the action instrumentation points have an instrumentation type, which indicates what type of PCM abstract is instrumented.

2.7.4. Adaptive Instrumentation

Source code is instrumented by inserting instrumentation statements into the original source code. The instrumentation statements account for e.g. the timing of the execution. They measure how much time the program did spent in a routine. Further statements can account for branch probabilities or loop iteration counts.

The instrumentation statements introduce a significant overhead into the execution of the program. Instrumenting the whole program is therefore undesirable.

Adaptive instrumentation can conserve execution overhead for software that is in development and changes regularly. Instead of instrumenting the complete application after each change, only the changed parts of the software are instrumented. This reduces the instrumentation overhead significantly.

Initially all parts of the software which are relevant to the AbPP are instrumented to obtain al needed Only changed parts of the software (which are relevant for the AbPP) are instrumented, as up-to-date PMPs still exist for the unchanged parts of the software. Instrumentation points can be deactivated once they were monitored for a sufficiently long time period.

2.7.5. Prototypical Implementation

The prototypical implementation is split into two separate parts: The monitoring and calibration was contributed by David Monschein. Further, a prototypical implementation of the commit-based CIPM approach exists for microservice-based software written in the Java programming language [2]. The prototypical implementation leverages parts of the implementation of the Co-Evolution approach which we previously introduced section 2.6. It uses the PCM as and architectural and performance model, the extended IMM to achieve adaptive instrumentation. Further, JaMoPP, the "Java Model Parser and Printer" is used to create a code model of for the source code. This prototypical approach was was further evaluated in the context of Java based microservices in [2].

2.8. The Lua Programming Language

The Lua programming language was first introduced in 1993. Lua is a dynamically typed interpreted and imperative scripting language. The language is described as a "tiny and simple language" [18]. The usage profile of Lua was previously described by the terms "Extensibility", "Simplicity", "Efficiency" and "Portability" [18]. Lua is designed to easily interface with other software written in e.g. C/C++. Lua provides a flexible table data type, in addition to the usual elementary types like boolean, string and numbers. Functions are first order types and can be assigned to variables and be contained by tables.

Lua has a standard library of functions that are always implicitly in the scope of a program. Examples for such functions are `print` which prints a string of characters to the command line and mathematical functions that do certain calculations.

Annotation based approaches exist for extending Lua with with named table types. These approaches are based on a server implementing the Language Server Protocol (LSP) [30, 29].

The Lua language uses so-called syntactic sugar, which are alternative ways to syntactically formulate certain semantics. An example for such syntactic sugar can be seen in Figure 2.6 Both statements declare an identical function `foo` that prints a variable.

2.9. SICK AppSpace Ecosystem

The SICK AppSpace is a commercial ecosystem of sensor applications [39]. SICK provides a wide array of sensor nodes like RFID and LASER scanners. A selection of the sensors can be programmed using Lua applications, see section 2.8. These applications are referred to as SensorApps or just *apps*. Non-programmable sensors can be integrated into the system sensors using Sensor Integration Machines (SIMs). SIMs can communicate with sensors and other systems using a variety of network protocols.

The apps are executed by the *AppEngine* component, which runs on the programmable sensors and SIMs [37, 41]. The AppEngine further provides infrastructure for the SensorApps: low-level interfacing with sensor hardware, network communication with other devices amongst others. SensorApps and AppEngine can provide a so-called common reusable objects wired by names (CROWNs), which is essentially an API. CROWNs can be injected into other SensorApps via dependency injection, which is facilitated by the corresponding AppEngine. Using the CROWNs more substantial applications can be composed of multiple simple sensor apps. CROWNs are provided by SICK SensorApps by *serving* them. A function of the AppEngine is used to register a function with the AppEngine, so it is available to the other apps.

Complete SensorApps are available through the SICK AppPool [38]. Depending on the requirements SensorApps are portable e.g. between different models of SIMs.

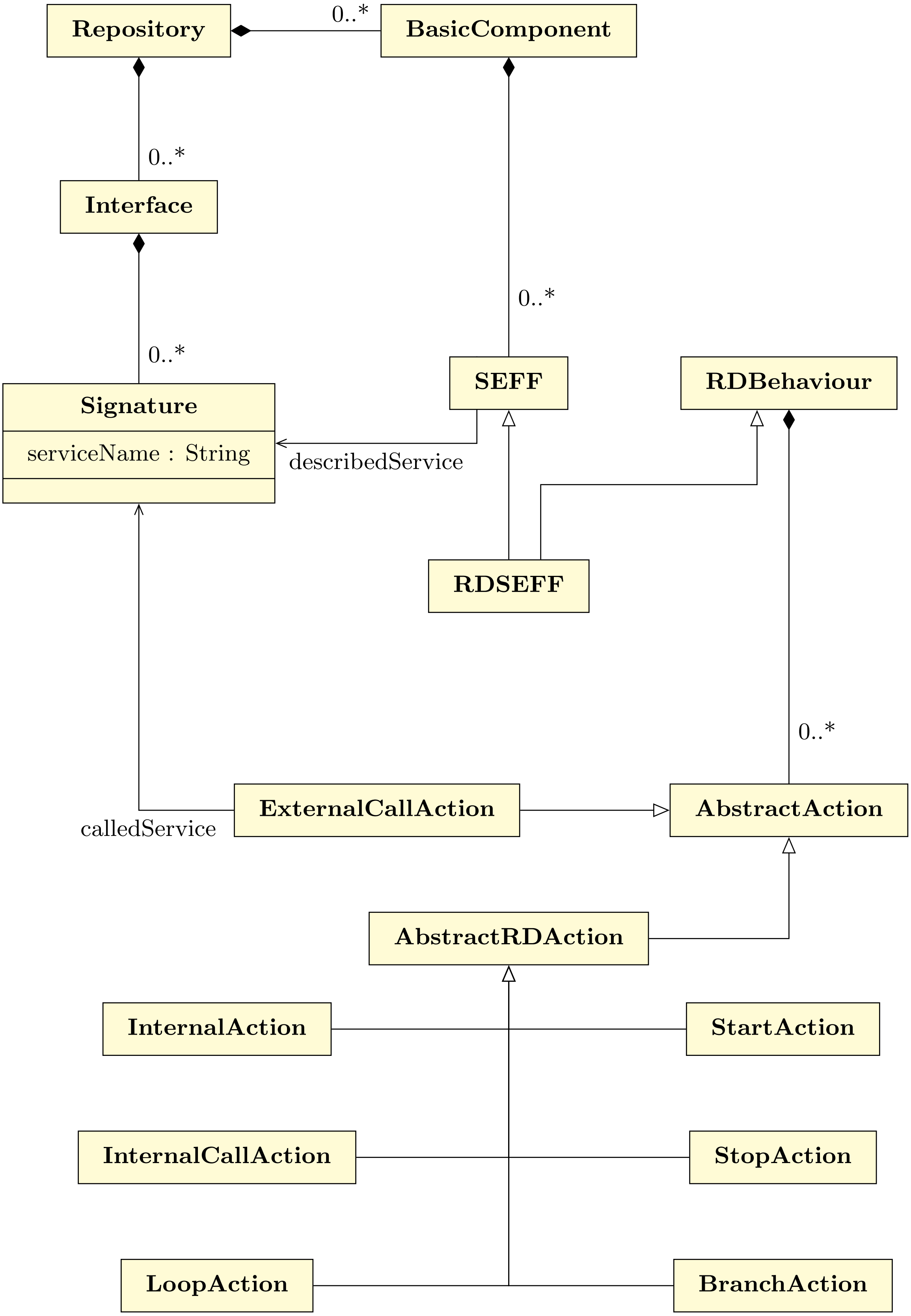
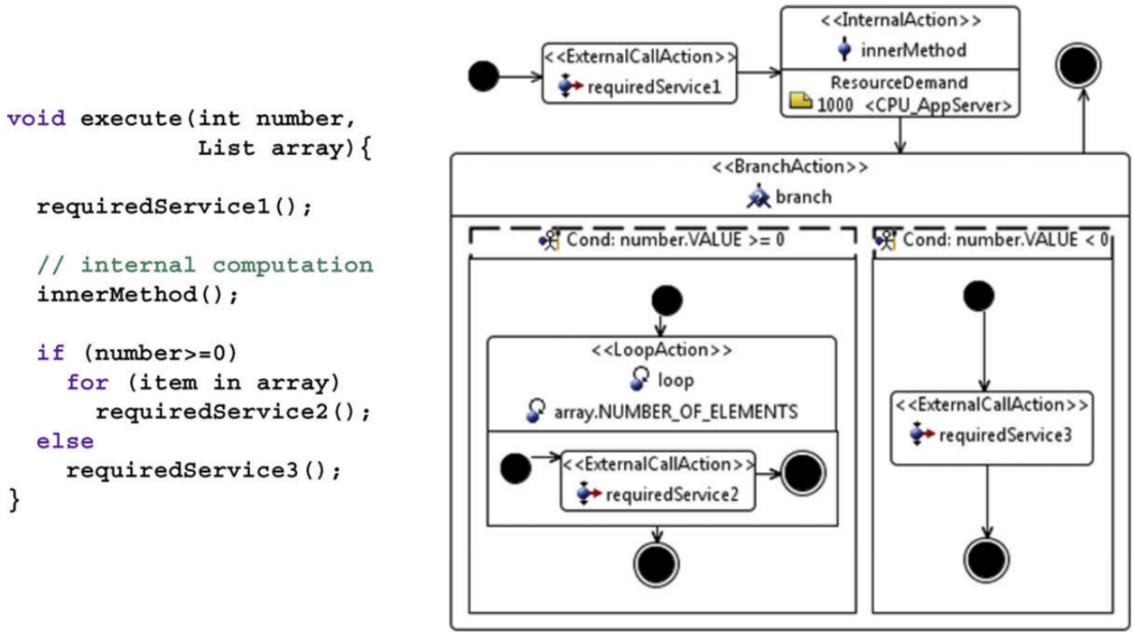


Figure 2.2.: An overview of the PCM Repository meta model based on [6].
SEFF = Service Effect Specification; RD = ResourceDemanding



```
void execute(int number,
             List array){

  requiredService1();

  // internal computation
  innerMethod();

  if (number>=0)
    for (item in array)
      requiredService2();
  else
    requiredService3();
}
```

Figure 2.3.: A pair of source code and the corresponding RDSEFF

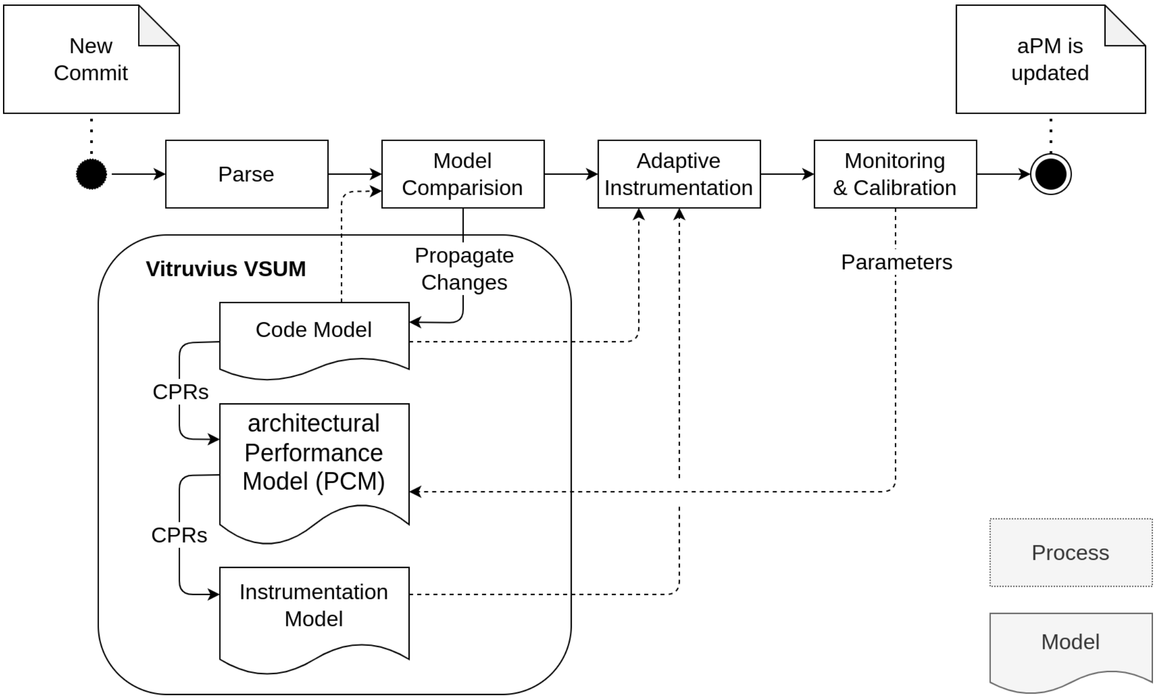


Figure 2.4.: An overview of the model update process of the CIPM approach [31]

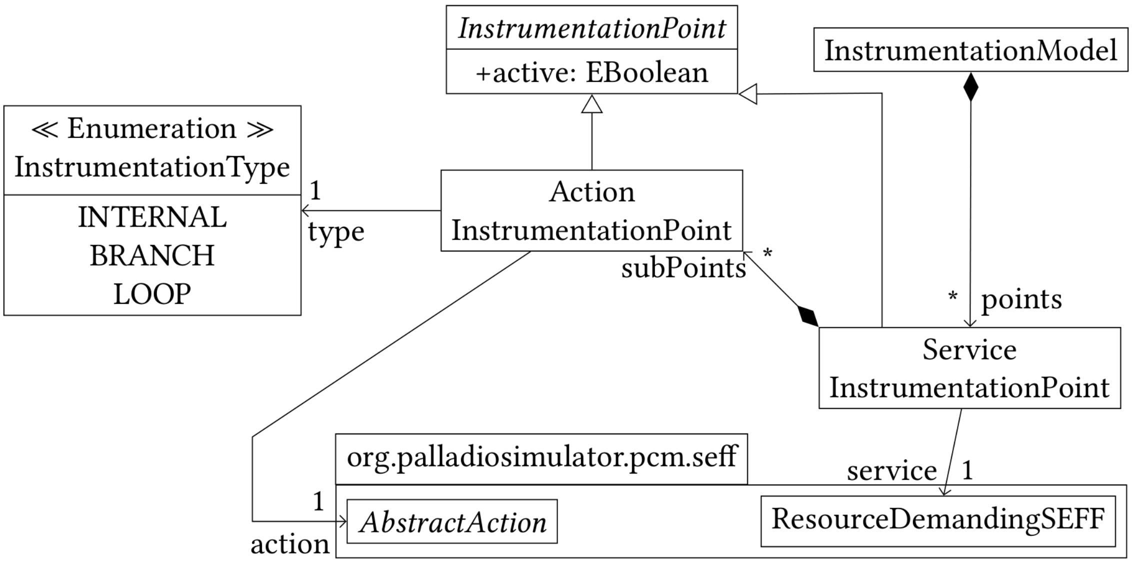


Figure 2.5.: The extended Instrumentation Meta-Model as presented by Monschein [33]

```
-- variable-based function instantiation
foo = function(bar) print(bar) end

-- this is equivalent to the previous statement
function foo(bar)
  print(bar)
end
```

Figure 2.6.: Two equivalent Lua statements

3. Approach Overview

This chapter presents an overview over the motivation for our approach. It is structured according to the PRICoBE principle (for Problems, Research Questions, Idea, Contributions, Benefits and Evaluation) [35]. The initial problem statements can be found in section 3.1 and the research questions we try to answer in section 3.2. The idea which inspired our approach is introduced in section 3.3. Furthermore, the contributions and benefits expected from the presented approach are listed in section 3.4 and section 3.5. The the problems, research questions and contributions correspond to other items of the same number. E.g. **P1.2** corresponds to **RQ1.2**.

3.1. Problems

The presented approach is embedded in the CIPM approach. The CIPM approach is prototypically implemented for the Java programming language. It is designed for software which adheres to a component based architecture. For example, previous works in the context of CIPM used microservices as software components.

The generalisability of the CIPM approach is still under investigation.

P1 The CIPM approach is solely designed and evaluated for use with the Java programming language.

- **P1.1** Both the source code parser and the CM are language specific and can therefore only be used for the Java programming language.
  - **P1.1.1** The method of change derivation used in the CIPM approach is code model specific.
- **P1.2** The CPRs used in the CIPM approach are specific to the Java CM.
  - **P1.2.1** The SEFF reconstruction method used in the CIPM approach is designed for the Java CM.
  - **P1.2.2** No approach for reusing SEFFs from the previous version exists.
- **P1.3** The method of source code instrumentation is specific to the Java programming language.

P2 The concept of software component detection currently only supports microservices.

3.2. Research Questions

This section lists research questions which correspond to the problems of the previous section 3.1.

RQ1 What conceptual changes are needed so the CIPM approach supports a programming language like the Lua programming language?

- **RQ1.1** How can a CM that is usable with the CIPM approach be obtained for Lua applications?
  - **RQ1.1.1** How can the change sequence between versions of a Lua CM be derived?
- **RQ1.2** How can CPRs be adapted to support the different CM? Which CPRs from the CIPM approach can be reused verbatim?
  - **RQ1.2.1** How can SEFFs be constructed functions of a Lua application?
  - **RQ1.2.2** How can SEFFs from the previous version be updated to match the new code model?

RQ2 How can the software components of Lua applications be discovered for use in the CIPM approach?

3.3. Idea

The idea of the presented approach is adapting the CIPM approach for use with Lua applications. The SICK AppSpace ecosystem (see section 2.9) will serve as a case study to evaluate and demonstrate the process of adapting the CIPM approach. The SICK AppSpace ecosystem uses Lua applications to implement and customize the behaviour of sensor nodes, SIMs and the cloud. In order to apply the CIPM approach to this different environment a significant portion of the prototypical implementation of the CIPM approach need to be adapted in an the adapted CIPM approach. Suitable parts of the existing approach should be reused and missing components must be implemented from scratch. Especially a code model for the Lua applications is created and integrated into the CIPM approach, which extends the applicability of the approach. The the adapted CIPM approach will be evaluated using real-world and close-to-real-world example applications from the SICK AppSpace ecosystem.

3.4. Contributions

The following contributions are planned for the thesis. The contributions listed in this section correspond to the problems from section 3.1.

C1 The prototypical implementation of the CIPM approach is adapted to support Lua applications from the SICK AppSpace ecosystem. The adapted prototypical implementation is evaluated.

- **C1.1** A Lua CM for the integration with the CIPM approach can be automatically created.
  - **C1.1.1** The change sequence between two version of the Lua CM can be derived. This change sequence can be used by the Vitrivius framework (see section 2.4) to propagate the changes into its VSUM.

- **C1.2** The CPRs used in the prototypical implementation are adapted to the new Lua CM.
  - **C1.2.1** A SEFF reconstruction method is implemented for the new Lua CM.
  - **C1.2.2** SEFFs from previous propagations are reused to achieve SEFF *updating*

C2 The component discovery is extended to support Lua sensor applications from the SICK AppSpace ecosystem.

3.5. Benefits

The following benefits are expected from the contributions listed in the previous section.

B1 The CIPM approach can now be applied to a wider range of use cases including Lua applications.

- **B1.1** A code model for Lua applications can be generated. The code model is integrated into the CIPM approach.
  - **B1.1.1** Changes in the CM can be propagated into the VSUM of Vitruv, which makes the code model usable with the Vitrivius approach.
- **B1.2** Changes to the Lua CM can be correctly propagated to a corresponding PCM and instrumentation model instance.
  - **B1.2.1** SEFF can be reconstructed for functions of Lua applications.
  - **B1.2.2** Only the changed statements of Lua application have to be instrumented and monitored again, because of fine-grained incremental SEFF *updating*,

B2 The CIPM approach is applicable to more software as more kinds of software components can be discovered.

4. The CIPM Approach for Lua-Based Sensor Applications

This chapter will describe the CIPM approach for Lua-based sensor applications and its prototypical implementation. We will refer to the presented approach as the adapted CIPM approach.

The CIPM approach as presented in section 2.7 was adapted to extend the applicability of the approach. The adapted approach extends the the CIPM approach approach to support Lua applications of the SICK AppSpace ecosystem.

The approach relies on a code model for the Lua applications as described in section 4.1. We define the consistency rules between instances of the Lua code model and the PCM in section 4.2. Further consistency rules between the code model and the instrumentation model are described in section 4.4. Finally, the prototypical implementation is presented in section 4.5.

4.1. Lua Code Model

The CIPM approach was extended by integrating an Xtext-based code model for the Lua programming language. As outlined in section 2.3 the Xtext can be used to generate a meta-model given an Xtext grammar of a language. As a starting point we extended an Xtext grammar from the Melange project [45]. Further details on how we obtain an instance of the Lua code meta-model presented here can be found in subsection 4.5.3.

The Melange grammar can be used to parse a single source code file into an EMF resource. The root element of this resource is a `Chunk`, the Lua representation of the contents of a Lua file.

The propagation of models into the Vitruv VSUM uses single Resources. We therefore combine all the individual file content models into one model, which then can be propagated into the VSUM. This enables the creation of a code model Resource that can be used with the CIPM approach.

We therefore extended the Melange grammar with additional rules. These rules are not used during the parsing of the individual source code files. Instead, Xtext generates classes for them in the meta-model. These synthetic classes are then instantiated by a parsing post process, that combines all the parsed chunks of the individual files. This results in a single model for the complete Lua application that can be propagated into the Vitruv VSUM.

An overview over the code meta-model can be seen in Figure 4.1. Classes contributed by use are depicted in blue.

The new central entity of the code meta-model is the Application. It models the source code of a complete Lua application. As such it may contain a number of Apps. Each App represents a single SICK AppSpace App. Further, each App has a name, which corresponds to name of the modelled app. The Chunk class from the original grammar did not represent the information about which file was modelled. We therefore wrapped the Chunk class into a SourceFile class. Each App may contain any number of SourceFiles.

Each Chunk contains one Block, which contains any number of Statements. The Statement class has many subclasses, of which only three subclasses are depicted in this overview. The StatementFunctionDeclaration represent the declaration of a Lua function. It has contains a Function, which represents the actual body of the function. StatementFor represents a for loop, while StatementIfElse represents conditional expressions.

Further, the BlockWrapper class was introduced as a super class for all the meta-model classes that contain a Block of statements. It is subclassed amongst others by the Function, StatementFor and StatementIfElse classes, which each contain a Block containing further Statements.

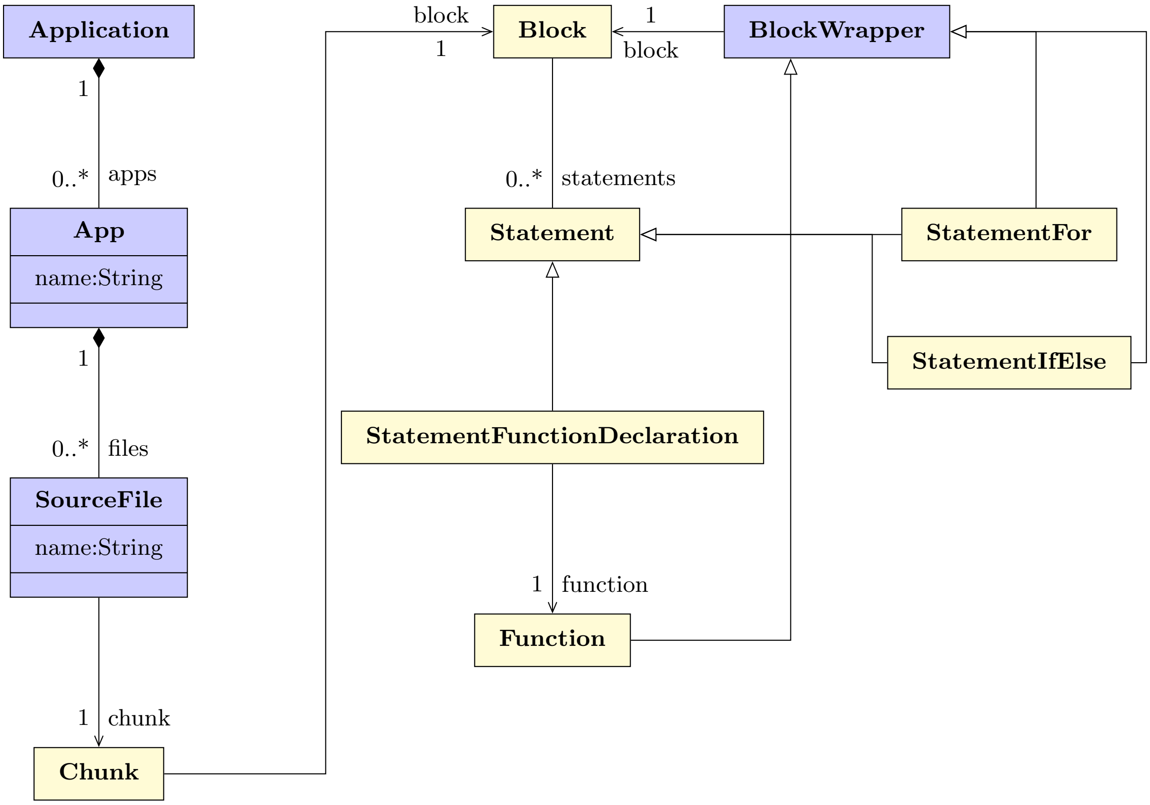


Figure 4.1.: A partial overview over the code meta-model. Classes depicted in blue were introduced into the original meta-model.

4.2. Consistency between the Lua Code Model and the PCM Repository Model

This section defines the consistency the code model and the PCM repository model.

Code model and PCM repository model are consistent if the following rules hold:

- For each `App` a PCM `BasicComponent` with the same name exists.
- For each served function declaration and its contents the following constraints must hold: If an object does fit a description on the left column of Table 4.1, it must have a corresponding element with type as indicated by the right column in the PCM repository model. If the object is in addition a named object, both corresponding objects must have the same name.

Lua Code Meta-Model	**PCM Repository Meta-Model**
`StatementFunctionDeclaration`	`OperationSignature`
Root `Block` the function	`ResourceDemandingSEFF`
Other `Blocks`	`ResourceDemandingBehaviour`
`StatementFor`	`LoopAction`
`StatementWhile`	`LoopAction`
`StatementIfThenElse`	`BranchAction`
`Statement` containing an internal function call	`InternalAction`
`Statement` containing function call to the standard library	`InternalAction`
`Statement` containing a function call to a CROWN which is not provided by a component	`InternalAction`
`Statement` containing an external function	`ExternalCallAction`

Table 4.1.: Class mapping between the code meta-model and the Repository meta-model of the PCM.

- Contiguous statements containing only internal function calls are represented by a single InternalAction. Dahmane showed that the monitoring overhead can be greater than the actually monitored resource demand [9]. This constraint reduces the monitoring overhead of the approach.

Modelling of the SICK AppSpace Ecosystem In the PCM repository model components may provide services to other components, and further has a interfaces containing the signatures of the provided services. Apps of the SICK AppSpace ecosystem also can provide functions to other apps. Each app has a manifest which describes the functions provided by the app, and further includes information regarding e.g. the parameters of a provided function.

Because of these similarities we decided to model SICK AppSpace apps as components in the PCM repository model. The functions served by an SICK AppSpace app are regarded as the provided services of a PCM component. As such, they all are part of the interface of the component.

Initially we considered also modelling further entities of the SICK AppSpace ecosystem as PCM components:

The Lua standard library could potentially be modelled as a component. This would require the the reconstruction of all the signatures of the standard library. In addition the Lua standard library is partially implemented as C libraries, which prevents the reconstruction of the internal functionality of the library functions. Because of this, the benefit of modelling the Lua standard library is limited. We therefore decided against modelling the Lua standard library as a component, because we felt that the modelling effort would be justified by the slight increase in modelling granularity.

Further, the app engine of the SICK AppSpace ecosystem could be modelled as a component. The app engine provides platform functionality to the apps which are deployed on the app engine. This functionality is available via Lua function calls, but is again not implemented in Lua. We therefore again decided against modelling the app engine as a component.

Limitation: Modelling of Types Because of the dynamically typed nature of Lua is challenging to determine the types for e.g. parameters of a function. We therefore do not consider the types of Lua variables, etc. in our approach. Instead we introduce a Lua *any type* into the PCM repository model as a composite type. The `OperationSignatures` reconstructed for the PCM repository model use this *any type* to model the parameters and return types of the signature.

Future work is required to extend the presented approach with a more complete support for reconstructing the types of parameters and return values. Three possibilities for determining the types of lua variables present themselves: By using a language server protocol implementation like the Lua language server [29, 30], one could query the language server to resolve the types of given language objects. This approach relies on the availability of type annotations in the source code.

Another approach could be to parse these type annotations directly within the approach. This could be achieved by extending the Lua code model of this approach with support for the parsing of the type annotations.

In the case of the SICK AppSpace apps, another approach could be the parsing of the app manifests [38]. The manifests contain information about the exported functions of an app. In particular their parameter and return types.

4.3. Fine-Grained Incremental SEFF Update

The CIPM approach currently does not contain an approach to the incremental fine-grained SEFF reconstruction. While an approach was proposed in previous work, it is not implemented and evaluated [2]. This approach is based on recreating the SEFF of a changed function and merging the old and the new SEFFs to obtain an updated SEFF.

We propose a different approach to the incremental fine-grained update of SEFFs. Instead of regenerating and merging SEFFs, we update existing SEFFs during the propagation of changes to the code model.

The actions which make up the step behaviour of a SEFF are reconstructed per statement. Statements are either added, removed or modified. Depending on the respective matching used to derive the changes in the code model, it is possible that a statement is either changed or removed and then added again in a modified form.

InternalAction Fusing Two actions are contiguous when their corresponding statements are contiguous in the statements of their Block or if the statements are only separated by statements without any actions. When two InternalActions would be contiguous after the insertion of an action into the SEFF step behaviour, we fuse the two actions. This means that one of the two actions is selected. All the statements corresponding to the other action are put into correspondence with the selected action. Then the other action is removed. The selected action now represents the resource demand of both former actions.

Added Statements For an added statement, we reconstruct actions based on the function call expressions the statement contains. Function calls to other apps are mapped to ExternalCallActions. Other function calls, such as function calls to the same app, calls to the Lua standard library and calls to CROWNs are mapped to InternalActions. A correspondence from the statement to all the reconstructed actions is added. Then all actions are inserted into the step behaviour of the parent SEFF one-by-one.

We calculate the insertion index for the action by finding the index of the statement in the current block and comparing it to the indices of the statements corresponding to the actions of the step behaviour.

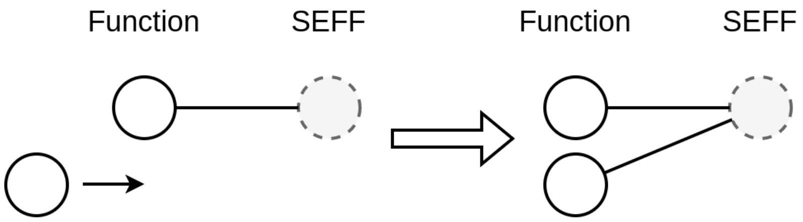


Figure 4.2.: Illustration of the fusion of the InternalActions of two statements which contain only internal calls.

If an InternalAction is inserted into the step behaviour and the preceding or succeeding action in the step behaviour is also an InternalAction, and further the current statement only has internal function calls, we add a correspondence from the current statement to the preceding InternalAction and drop all the reconstructed InternalActions. This situation is illustrated in Figure 4.2. On the left the step behaviour of the SEFF and the Block of a function are depicted. The block already contains a statement with an internal call that corresponds to an InternalAction in the SEFF.

A statement with an internal call is then inserted into the Block of the function. Initially an InternalAction is reconstructed for the new internal call, but during the insertion procedure it is *fused* with the already existing InternalAction. After the insertion the statements with internal calls remain and both correspond to the single InternalAction.

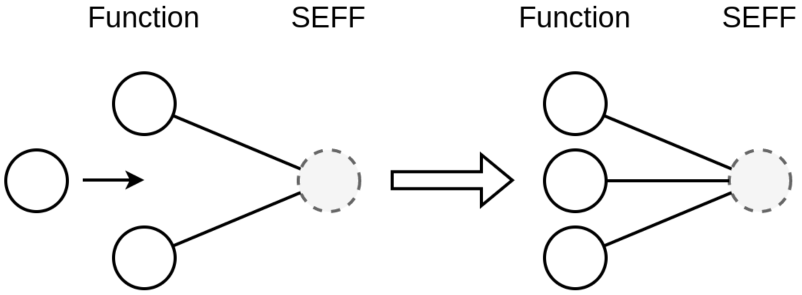


Figure 4.3.: An internal statement is inserted between statements corresponding to the same fused InternalAction. It is therefore also fused to this InternalAction

If an InternalAction is inserted into the step behaviour and both a preceding and a subsequent statement are corresponding to the same InternalAction (because of action fusing) we add a correspondence between the current statement and the fused action and drop the action that would have been inserted.

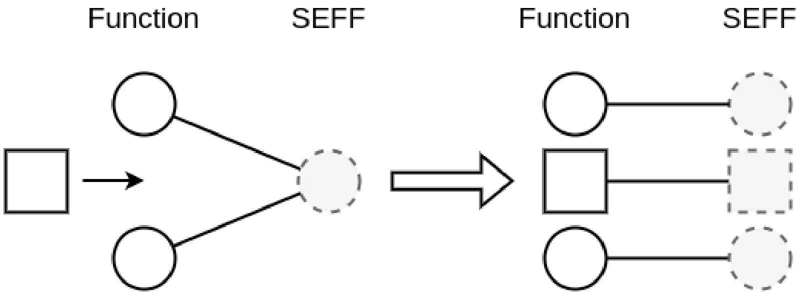


Figure 4.4.: Illustration of an action insertion that causes the splitting of an internal action.

If a statement containing an ExternalAction is inserted into the step behaviour and both a preceding and a subsequent statement are corresponding to the same InternalAction (because of action fusing) we delete the fused action insert the ExternalAction into the step behaviour and reconstruct actions for all the statements of the now deleted InternalAction

Removed Statements When a statement is removed we remove its correspondences to its actions. If these actions have no further correspondences to other statements, they are deleted.

If after the deletion of an actions two internal actions are contiguous in the step behaviour of the SEFF, they are fused as previously described.

Changed Statements When a statements is change, we determine all corresponding actions of the statement. Then all corresponding statements for all these actions are determined. All the actions are removed from the step behaviour and then we reconstruct the actions of the statements and insert them as described under *Added Statements.*

4.4. Consistency between the Lua Code Model and the Instrumentation Model

The change propagation to the instrumentation model is adapted, because of our new approach to the fine-grained incremental SEFF update. Service Instrumentation Points (SIPs) corresponding to SEFFs and Action Instrumentation Points (AIPs) corresponding to actions are created and removed as in the original CIPM approach.

When changes are made to a served function, its corresponding SEFF is updated, not reconstructed. Therefore special attention has to be put on the fused internal actions (see the previous section). If the correspondences of a fused action change because more statements were "fused" into the action the resource demand that is represented by the InternalAction changes. This requires that the AIP that corresponds to this InternalAction is activated if it was previously deactivated. This causes the represented resource demand of the action to be instrumented again, which is required to recalibrate the performance model.

4.5. Prototypical Implementation of the CIPM Approach for Lua-Based Sensor Applications

This section will present the prototypical implementation of the adapted CIPM Approach. The previous prototypical implementation upon which the presented prototypical implementation is based is discussed in subsection 4.5.1. Changes to this previous prototypical implementation are described in subsection 4.5.2. The process of obtaining a code model for a SICK AppSpace application is discussed in subsection 4.5.3. Further, the implementation of the detection of apps of such an application is outlined in subsection 4.5.4. The implementation of the change derivation is described in subsection 4.5.5. We implement the previously presented consistency preservation rules using the reactions from Vitruvius. The implemented reactions are described in subsection 4.5.6 and subsection 4.5.7 respectively.

An overview over the state of the prototypical implementation is depicted in Figure 4.5. We implemented a new code model for Lua. Lua source code can be parsed into code model instances using a generated parser. Code model instances can be compared and changes can be derived as described in subsection 4.5.5. The code model is integrated into the Vitruvius VSUM. Changes to the code model can be propagated to the PCM and the instrumentation model of the VSUM using newly contributed CPRs. The already existing CPRs from the PCM to the instrumentation model were adapted slightly. Because of the

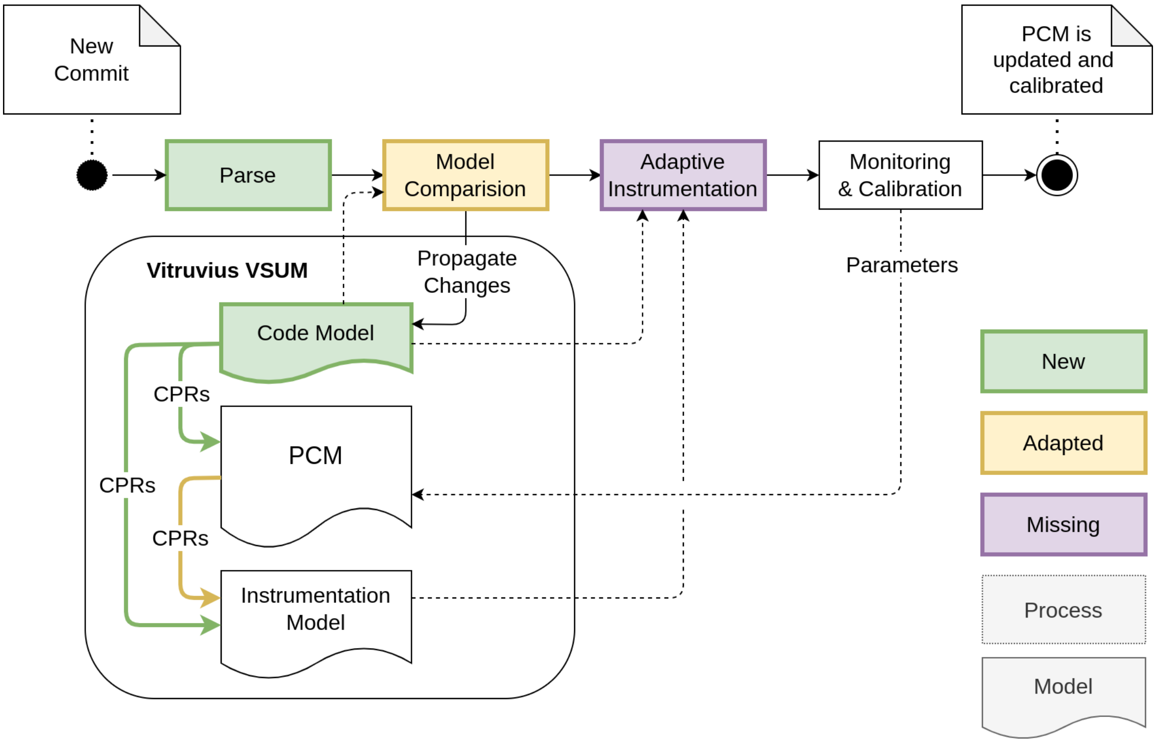


Figure 4.5.: An overview over the workflow of the approach annotated with information regarding the state of the prototypical implementation regarding the respective process.

time constraints of this thesis we could not implement the adaptive instrumentation of the Lua source code. The monitoring and calibration phase was implemented by previous work [33].

4.5.1. Prototypical Implementation of the Commit-Based CIPM Approach

The CIPM approach was previously prototypically implemented for microservice-based Java web applications [2]. The prototypical implementation of the original the CIPM approach used JaMoPP as a code model [17]. It further leverage's the Vitruvius framework to keep the models of the software consistent, see section 2.4 [48]. In particular, the Vitruv framework version 2 is used.

The used models are the JaMoPP as the source code model, the PCM as architecture and performance model of the software (see section 2.5) and the extended instrumentation (meta-)model (see subsection 2.7.3).

4.5.2. Changes to the Previous Implementation

As previously stated, the prototypical implementation of the adapted CIPM approach was based on the prototypical implementation of the original CIPM approach.

The prototypical implementation is changed in order to match the different approaches of the adapted CIPM approach. The adapted prototypical implementation now uses version

3 of the Vitruvius framework, which requires the use of a view-based API to propagate changes into the Vitruv VSUM. In addition the architecture of the implementation was reworked, to separate concerns of the implementation, which we hope will ease maintenance of the implementation. Further, steps were taken to make the architecture of the implementation more generic. The goal is that other code models can be integrated into to the implementation without the need to adapt the complete implementation. We therefore extracted logic which is note specific to the code model into generic classes.

An overview over the adapted commit propagation architecture is depicted in Figure 4.6 The newly introduced generic class `CommitIntegrationState` is the central element of the commit integration process. The class encapsulates all the models that are part of the approach in the form of model facades. Facades exist for the PCM, the extend instrumentation model and further a generic code model. The `CommitIntegrationState` allows the creation and loading of copies of the current state. In addition to the individual models, a facade to the Vitruvius VSUM is part of the commit integration state.

The commit integration state is in association with a `CommitIntegration`, which encapsulates all information necessary, to instantiate a `CommitIntegrationState`. This includes the git repository that contains the propagated source code; the change propagation specifications, which define how changes are propagated in the Vitruv VSUM; and component detection strategies, that are used by the code model facade.

The actual commit propagation logic is implemented in a generic `CommitIntegrationController`.

The `AppSpaceCommitIntegrationController` extends the generic `CommitIntegrationController`, implements the `CommitIntegration` interface. Further, the `AppSpaceCommitIntegrationController` is not generic, but specific to the `LuaCodeModelFacade`. By implementing the `CommitIntegration` interface, any test class has the possibility to modify way the commit integration state is initialized.

4.5.3. Lua Code Model

The first step of the adapted CIPM approach is obtaining a code model for a given version of a software. We use the Xtext project for the code meta-model and parser generation [50]. As a starting point a Xtext grammar from the melange project was used [45].

The grammar, which was solely published as an example to demonstrate the melange process, was insufficient for our intentions to integrating the code model into the CIPM approach. This is due to the grammar focussing on the syntactical reproduction of the source code. The grammar was configured to use backtracking as it used non left-factored statements.

Further we intended to implement reference resolution at the Xtext grammar level, which was also not implemented by the melange grammar. To fully benefit from the reference resolution mechanism provided by Xtext, we implemented a scope provider for the grammar, so references are resolved within the correct scope of the Lua program.

We disabled parser backtracking, as we found that the backtracking made debugging the grammar more difficult and unpredictable. Consequently we left-factored all non-left factored rules.

The SEFF reconstruction which is used in the CPRs is aided by the ability to resolve function calls to their declarations.

We introduced a `Referenceable` type into the code meta-model using multiple inheritance. This type bundles all productions rules that produce elements that are referenceable by name, which includes e.g. function declarations and variable assignments. Other elements of the code model can reference elements of the Referenceable type. This is used to implement e.g. variable name expressions. The dynamically typed nature of Lua did not permit us to easily add type scoping to the reference resolution.

An instance of our new code meta-model is created by the following steps:

1. Parse all Lua files in all the apps that are part of the project. This yields an EMF resource containing a Lua Chunk (which is part of the defined code meta-model) for each parsed file.

2. Apps are detected as described in subsection 4.5.4. In addition a synthetic `App` is created for mocked function declarations, that are created in the next step.

3. All proxies in the code model are eliminated by replacing them with references to mock function declarations in the synthetic mock app. The calls which are mocked include calls to the Lua standard library (which are implicitly in scope of Lua applications), calls to CROWNs (see section 2.9) of other components and the device itself.

4. Calls from one app to another have been mocked by the previous process. These mocks are resolved by a post process, which replaces the reference to the mock with a reference to the correct function declaration in the providing app.

5. For each `Chunk` a containing `SourceFile` object is created.

6. The `SourceFiles` are added to their respective `App`.

7. All `Apps` are copied into an `Application` object.

8. The resulting `Application` is saved in a single resource.

The resource which was created by this process can be propagated into Vitruv VSUM and be used further for the CIPM approach. Because of the time constraints of this thesis, the obtained code model has the following limitations:

Limitation: Incomplete Support for Lua Syntax Sugar As described in section 2.8, Lua is scripting language with a very flexible syntax including a significant amount of *syntactic sugar*. The same semantics can be expressed in multiple ways which makes the implementation of a grammar that captures all these semantics complex and time consuming. An example of such ambiguous syntax can be seen in Figure 2.6. Both examples declare the same function. We implemented the first version of the expression, as the latter would require the correct resolution of variables throughout the code model. Because of the time constraints of this thesis a subset of these Lua semantics was implemented. We are confident that a more complete grammar could be implemented given more time.

4.5.4. Detection of the SICK AppSpace Apps

Analogous to the approach to detect components of microservice based applications presented by Armbruster, we detect SICK AppSpace apps by locating all the app manifests contained in the target repository [2]. We use the app name as the component name for the modelling. All source files within the scripts directory of an app, are parsed, wrapped into a named chunk (see subsection 4.5.3) and included with the corresponding `Application`.

4.5.5. Hierarchical State-Based Change Resolution

As described in subsection 2.2.3 we need to derive changes in an automatically generated model, the code model. For this we leverage EMF compare, which can compare and merge Ecore based models [13]. EMF compare requires a match engine for its operation. The DefaultMatchEngine provided by EMF compare was not sufficient for matching versions of the code model.

Instead we reused the hierarchical match engine approach from SPLevo which was used by Armbruster for the matching of a Java code model [22]. This hierarchical approach compares elements of the models which are on the same level in the containment hierarchy. E.g. first the root elements of both models are compared and subsequently the contents of the root elements, provided there was a match between the root elements.

The hierarchical match engine relies on an equality checker for the individual elements. We implement a custom equality check for a subset of the types of the code meta-model. For all remaining types we fall back to an edition distance based equality checker, which is provided as part of EMF compare. The edition distance based equality checker compares elements by calculating an edition distance, from one element to the other based upon the elements features and contents. If the edition distance is below a certain threshold, the elements are considered equal.

Most of the custom equality checks similarly comparisons of the elements names and contents. A notable exception is the comparison of two *variable expressions*. A variable expression has a reference to an referenceable element, which corresponds to the assignment of the variable. Comparing variable expressions based on the name of the variable is not sufficient. If new assignments to the same variable within a more local scope to the variable expression are introduced in the code model, the name based comparison would determine the variable expression equal in both models. Instead the variable expression must now reference the assignment of the more local scope now, because of the shadowing semantics of the scoping. Therefore, in this case and similar cases where references are involved, we also compare the location of the referenced elements, to detect the replacement of the referenced object.

4.5.6. CPRs from the Lua Code Model

To keep the Lua code model consistent with the PCM repository model, we use the reactions DSL of Vitruv.

We gather information about the apps that are propagated during the change propagation, by scanning the code model.

Essential for the approach is to determine which functions are exported to other apps or *served*. Instead of additionally parsing the manifests of an app to determine if a function is served and therefore available to other app, we instead scan the source code for the function calls that *serve* the actual function declaration.

During the scanning of an `Application` we gather the following information:

- Which function declarations are served, including which function call is serving them.
- Other apps called by an app, to determine the `OperationRequiredRole` for these calls.
- For each served function declaration, we track which actions are calling the function declaration.
- Within each served function we find `Blocks` containing architecturally relevant calls These will be used to determine if actions must be reconstructed for a statement.

Vitruv as of version 3 differentiates between subtypes of correspondences. Correspondences created in reactions are `ReactionsCorrespondences`, while others are `ManualCorrespondences`. This change required modifications to the model initialization of the PCM repository model and the instrumentation model. Previously the root elements of these models were inserted into the model during the initialization and then set into correspondence with their corresponding Literal. This is required to locate the models root elements in the reactions. Because of the aforementioned differentiation, correspondences created outside of the reactions language are not visible inside of them. To work around this we created initialization reactions, that create these correspondences when the corresponding root element is inserted into the model.

Regarding the propagation of changes to the Lua code model the following reactions were implemented:

Creation of `Apps` When an `App` is created a BasicComponent for it is created in the PCM repository model. In addition, an empty OperationInterface and an OperationProvidingRole are created.

Creation of `StatementFunctionDeclarations` For created StatementFunctionDeclarations we determine if the declaration is server. If it is an OperationSignature is created for the declaration and further inserted into the OperationInterface of the containing App.

Creation of `Blocks` For created Blocks we determine if the block is marked for action reconstruction in the previously gather information. If it is and it further is the root block of its containing StatementFunctionDeclaration a ResourceDemandingSEFF is created for the block. If the block is not the root block a ResourceDemandingBehaviour is created instead.

Creation of `Statements` To determine if actions must be reconstructed for an added Statement, we check if its containing Block is marked for action reconstruction. If the statement is a control flow statement and a BlockWrapper as previously introduced, we further check if the block contained by the Statement requires action reconstruction.

If the statement is no control flow statement, we find all function call expressions in the statement and classify them: Function calls to other Apps are classified as external calls and an ExternalCallAction for the function call is created. All other function calls are currently classified as internal calls and hence an InternalAction is created for these calls.

After all actions for the added statement have been reconstructed, they are inserted into the step behaviour of the ResourceDemandingBehaviour or ResourceDemandingSEFF corresponding to the block that contains the added statement.

Changing of `Statements` It is not possible to formulate a trigger for an reaction that triggers if any content of an object is changed. Instead the reaction language provides an `anychange` trigger, that is triggered for all changes. We use this trigger and filter the changes in order to obtain changes to the contents of statements.

If changes inside of a statements are detected, we remove all corresponding actions of the statement from the PCM repository model. We reuse routines for the creation of actions for a statement to create the new actions for the changed statement.

Creation of *serve* calls Creation of function call expressions which *serve* a function declaration.

As previously described OperationSignatures, ResourceDemandingSEFF, etc. are created when served Blocks are added. This approach does not work if an already function existing that was previously not served becomes now served. For an added serve call, we locate its served function declaration and then follow the same steps as if it had just been created as described above.

It is noteworthy that it is possible that for a commit that adds a new function declaration and corresponding serve call, that the change containing the creation of the serve call is processed first. This creates the complete seff immediately. All subsequent reactions to e.g. the added StatementFunctionDeclaration will not create a duplicate OperationSignature but instead, do nothing because every routines checks if the correspondences that are created by it are already existing.

Removal of objects When objects are removed from the Lua code model, the implemented reactions remove the corresponding objects from the PCM repository model. Actions are removed from the step behaviour of a SEFF, by removing them and further fixing the predecessor and successor fields of the adjacent actions.

An overview over the correspondences created by the reactions can be seen in Table 4.2. Objects of the Lua code meta-model are listed on the left and their corresponding counterpart of the PCM repository model on the right.

Additional correspondences within the Lua Meta-Model:

Lua Code Meta-Model	**PCM Repository Meta-Model**
`App`	`BasicComponent`
`App`	`OperationInterface`
`App`	`OperationProvidedRole`
`Block`	`ResourceDemandingSEFF`
`Block`	`ResourceDemandingBehaviour`
`StatementFunctionDeclaration`	`OperationSignature`
Control Flow `Statements` (`StatementFor, StatementWhile, StatementIfThenElse`) which contain no architecturally relevant function calls	`InternalAction`
`StatementFor`	`LoopAction`
`StatementWhile`	`LoopAction`
`StatementIfThenElse`	`BranchAction`
`Statement` containing a function call within the same component	`InternalAction`
`Statement` containing a function call to the standard library	`InternalAction`
`Statement` containing a function call to a CROWN which is not provided by a component	`InternalAction`
`Statement` containing a function call to another component	`ExternalCallAction`
Other `Statements`	-

Table 4.2.: Correspondences between the code model and the PCM repository model as created by the CPRs of the prototypical implementation.

- A function call that serves a function declaration corresponds to this served function declaration.
- The Repository literal of the PCM corresponds to the Repository object of the PCM repository model.
- The Repository literal of the PCM further corresponds to the used `LuaAnyType`
- `ExternalCallAction` additionally correspond to the `OperationRequiredRole` which were created for the call actions.

4.5.7. CPRs to the IMM

This section describes how the consistency rules as defined in section 4.4 are preserved during the commit integration.

Armbruster implemented CPRs from the PCM to the instrumentation model using the reaction language of Vitruv [2, 21]. We were able to reuse nearly all of these reactions without modification.

The reactions from the Lua code model to the code model were extended to activate AIPs for which the represented resource demand was changed by a seff update

The reactions language allows the specification of multiple source and target models. Reactions are triggered when changes are made in the source models and then propagated to the target models. The reactions for the Lua code model therefore now target the PCM and the instrumentation model.

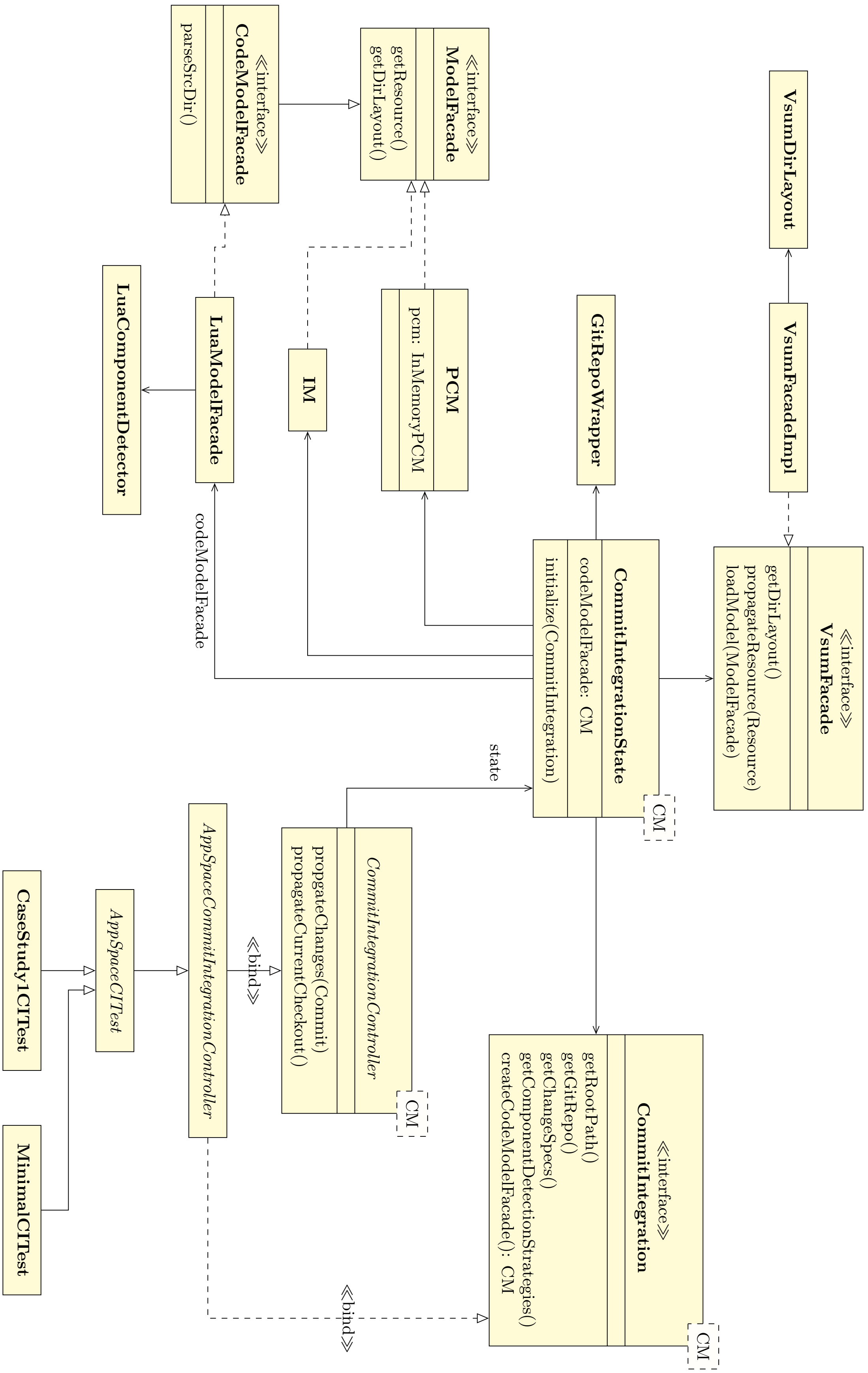


Figure 4.6.: The adapted commit integration architecture of the prototypical implementation.

5. Evaluation

This chapter discusses the evaluation of the prototypical implementation presented in the previous chapter. The evaluation follows the GQM paradigm [47]: The goals of the thesis are defined in a conceptual manner. For each goal, one or more questions are posed. These questions must be answered to determine if the corresponding goal is met. Metrics are selected for each of these questions. By measuring these metrics, the questions may be answered. Answering the questions will aid in coming to a conclusion regarding the goals of the thesis.

The detailed GQM plan is listed in section 5.2. The requirements and application concepts for the case studies are discussed in section 5.3. The actual case study applications are presented in section 5.4. The metrics for the GQM are acquired using two experiments. The results of these experiments are presented and discussed in section 5.5. An overall discussion of the evaluation and the approach can be found in section 5.6. Finally, the threats to the validity of this evaluation are discussed in section 5.7.

5.1. Metrics

This section defines metrics which are used in the evaluation of the the adapted CIPM approach.

5.1.1. Jaccard Coefficient

Originally defined by Jaccard as the "coefficient of community" [19], the Jaccard Coefficient or Jaccard Similarity Coefficient of two sets is defined as their intersection over their union of the sets, see Equation 5.1.

$$JC(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

The coefficient measures the similarity of two sets, with two identical sets resulting in a Jaccard Coefficient of 1 and two sets without any intersection resulting in a Jaccard Coefficient of 0.

Analogous to Monschein, we use the Jaccard Coefficient to measure the similarity of models [33]. Comparing two models is done by matching elements of the models against each other. This relies on a similarity measure for individual model elements. The similarity of two model elements can be determined by comparing their type, features, position in the model and contents. For models with a common ancestor, it is possible to also take the identifiers of model elements into account.

To calculate the Jaccard Coefficient between two models, we define that the elements of a model are its set. Further, we define that model elements that were successfully matched are part of the intersection of the two sets. The union of the two sets contains the intersection and all unmatched elements. It follows that the informative value of the Jaccard Coefficient depends on the quality or stringency of the matching that is used to match the two models.

5.1.2. F-Score

The F-Score or F-Measure is a measure for the accuracy of binary classifications [10]. Binary classification can observe the following results: true positive (TP) classification, true negative (TN) classification, false positive (FP) classification and false negative (FN) classification.

The F-Score is often used measure the quality of a retrieval process that selects data from a large pool of information. In this case the F-Score relies on a relevance definition, that which prescribes which data of the pool are relevant and should therefore be retrieved.

The F-Score of classification results is derived from the recall and precision of the classification results, which are defined as:

$$\text{Precision: } P = \frac{TP}{TP + FP} \tag{5.2}$$

$$\text{Recall: } R = \frac{TP}{TP + FN} \tag{5.3}$$

Precision is a measure for the quality of the retrieved entities, while Recall is a measure for the completeness of the retrieval. Because both precision and recall are not conclusive measures regarding the classification, the F-Score tries to balance both measures by using the harmonic mean of precision and recall[10]. The general F-Score or F_β and the commonly used F_1 form are displayed in Equation 5.4 and Equation 5.5 respectively.

$$F_\beta = \frac{(1 + \beta^2)PR}{\beta^2 P + R} \tag{5.4}$$

$$F_1 = \frac{2PR}{P + R} = \frac{2TP}{2TP + FP + FN} \tag{5.5}$$

This thesis uses the F_1 form of the F-Score and we will subsequently refer to this form as the F-Score.

We use the F-Score to measure the consistency of two models that are kept by processes of the presented approach. Given models A and B that are consistent if the existence of a model object $a \in A$ implies the existence of a corresponding element $b \in B$. We say that cases were we find such corresponding a and b are the true positives. Cases were we find an a without a corresponding b are the false negatives. On the contrary, cases were we find an b without a corresponding a are the false negatives. In cases where there are no relevant model elements in either models, the F-Score is not defined.

5.1.3. Total Instrumentation Point Activation

This section defines the Total Instrumentation Point Activation (TIPA), a measure for how many instrumentation point were active during the propagation of a sequence of commits using the the adapted CIPM approach.

Given a sequence of commits C the TIPA of the sequence is defined by

$$\begin{aligned} AIP(c) &:= \text{The set of active instrumentation points after propagation of c} \\ IP(c) &:= \text{The set of instrumentation points after propagation of c} \end{aligned}$$

$$TIPA(C) = \frac{\sum_{c \in C} |AIP(c)|}{\sum_{c \in C} |IP(c)|} \tag{5.6}$$

The TIPA has values from 0 to 1 with 1 meaning all instrumentation points were active after all propagations and 0 indicating that no instrumentation points were ever active.

TIPA allows the coarse grained analysis of how the overhead of monitoring active instrumentation points was reduced by the approach.

5.2. GQM Plan

This section presents the GQM plan that was used for the evaluation of the approach. The goals and subgoals correspond to the contributions from section 3.4 as indicated by their indices.

G1 Adapt, extend and evaluate the CIPM approach and the prototypical implementation with support for Lua applications.

- **Q1** Does the prototypical implementation support Lua code?
  - **M1** Satisfaction of the subgoals **G1.1** and **G1.2**.
- **Q2** How much does the the adapted CIPM approach reduce the monitoring overhead of a Lua application in terms of activated instrumentation points?
  - **M2** [**TIPA**] Active instrumentation points, see subsection 5.1.3.

    We can calculate this metric by aggregating the results of all propagations of the commit history of an applications. For each propagation we can determine the total number of AIPs which existed after the propagation. In addition we can calculate how many of these AIPs were activated.

    By accumulating these values for all the propagations we can calculate the TIPA as defined in subsection 5.1.3.
- **Q3** What is the execution time of the change propagation?
  - **M3** [**Execution Time**] Both the change propagation time for a commit and the change propagation time per propagated change can be calculated.

G1.1 Lua source code of applications can be parsed and a code model for use with the CIPM approach can be generated. Further, changes within the code model can be derived using a state based approach.

Q4 For how many commits of an application can a correct CM of the Lua source code be generated?

M4 [**Textual Differences**] Similarity of:

- Original source code of a Lua application and commit c: $O(c)$
- The result of parsing $O(c)$ into a code model and printing it

The compared texts may differ, but not in a semantically relevant manner.

The original source code of an application can be acquired during the parsing step of the approach. Having parsed a source file into an EMF Resource, we print the Resource as source code using the printing functionality provided by Xtext. This yields us two text files for comparison: The original source file and the parsed and printed file. The two files are compared character by character. If they are equal, then both files are considered "identical". If not both files are put through the same filtering process: Extraneous contents such as blank lines, comments are stripped from the files, as these are not reproduced well by the printing facilities from Xtext. If the filtered texts now match the files are considered similar, otherwise they are considered dissimilar.

Q5 How many changes to the CM can be propagated correctly to the Vitruvius VSUM?

M5 [**Jaccard Coefficient**] Similarity of:

- The parsed code model $P(c)$ of a commit c.
- The VSUM code model after the commit c was propagated into the VSUM.

These two models are compared to determine if the code model of the VSUM was updated correctly. Both models are matched against each other using the hierarchical matching approach described in subsection 4.5.5. Based on the matched and unmatched elements of both models the Jaccard Coefficient as defined in subsection 5.1.1 can be calculated.

G1.2 Architectural changes to the Lua CM can be propagated to the PCM and the IM using the adapted CPRs.

Q6 Are architectural changes to the CM propagated correctly?

M6 Satisfaction of the subgoals **G1.2.1** and **G1.2.2**.

Q7 How many CPRs are executed during the propagation of the commits of a Lua application?

M7 [**Coverage Percentage**]

To our knowledge no approach to calculating the coverage of the execution of Vitruv reactions exists. We can approximate the coverage of the reactions by gathering the coverage data of the generated Java code for the implemented reactions.

G1.2.1 Architectural changes to the Lua CM can be propagated to the PCM using adapted CPRs.

Q8 How many architectural changes to the CM correctly propagated to the repository model of the PCM?

M8.1 [**Jaccard Coefficient**] Similarity of:

- The PCM Repository model of the VSUM after the propagation of a sequence of commits $C := \{c_0, \ldots, c_i\}$: $R(C)$
- A manually created and validated PCM Repository model for the Lua application and commit c_i: $MR(c_i)$.

M8.2 [**Jaccard Coefficient**] Similarity of:

- The PCM Repository model of the VSUM after the propagation of a sequence of commits $C := \{c_0, \ldots, c_i\}$: $R(C)$.
- The PCM Repository model of the VSUM after the propagation the of commit c_i: $R(\{c_i\})$.

To compare the two models a matching is required. We reuse an existing comparison matching implementation for the PCM repository model which is not based on identifiers.

G1.2.2 The prototypical implementation can propagate architectural changes to the instrumentation model of the VSUM. Further, it is possible to implement adaptive instrumentation using this instrumentation model.

Q9 How many changes to the PCM repository model are propagated correctly to the IM of the VSUM?

M9.1 [**F-Score**] Matching of SEFFs and SIPs of the two models.

This metric can be calculated for a propagation by iterating through all SIPs and SEFFs of the resulting instrumentation model and PCM repository model contained in the VSUM. For each SIP in the instrumentation model we check if its referenced SEFF exists in the PCM repository model. If the referenced SEFF is found the SIP and SEFF are considered "matched", if not the SIP is considered "unmatched". In addition we iterate through all the SEFFs found in the PCM repository model and check if they have matched a SIP previously. If this is not the case the SEFF is considered *unmatched* Using Table 5.1 the F-Score as defined in subsection 5.1.2 can be calculated.

M9.2 [**F-Score**] Matching of abstract actions and AIPs of the two models.

This metric can be calculated using an analogous matching scheme to the one used for the previous metric and Table 5.2.

TP	Matching SEFF and SIP exist
FP	SIP without matching SEFF exists
FN	SEFF without matching SIP exists

Table 5.1.: The classification of matching results regarding service instrumentation for the calculation of an F-Score.

TP	Matching abstract action and AIP exist
FP	AIP without matching abstract action exists
FN	Abstract action without matching AIP exists

Table 5.2.: The classification of matching results regarding action instrumentation for the calculation of an F-Score.

Q10 How many instrumentation points of the IM are correctly activated and deactivated and is it consequently possible to implement adaptive instrumentation using the instrumentation model?

M10 [**F-Score**] Matching of abstract actions that were changed since the last instrumentation and active AIPs of the two models.

A matching scheme similar to the scheme used for the service and action instrumentation in the previous two metrics can be used to calculate this metric for a propagation. We determine which actions changed by matching the PCM repository model with its version from the previous propagation. An identifier based matching can be used for the matching, as the newer model is a descendant from its previous model, and consequently uses the same identifiers for unchanged actions. Unmatched actions of the newer repository model are considered the changed actions and require a corresponding active AIP. Analogous to the previously used matching schemes we search for matching changed actions and active AIPs.

The F-Score of this matching can then be calculated using Table 5.3 and Equation 5.5. If no changed actions are found, no F-Score can be calculated because the F-Score is not defined for retrievals without any true positive cases.

As we did not actually implement the instrumentation and monitoring of Lua applications we simulate the complete instrumentation and monitoring of the application by deactivating all the AIPs before the subsequent propagation.

G2 The adapted prototypical implementation can automatically detect Apps of the SICK AppSpace ecosystem.

Q11 How many apps can the adapted prototypical implementation discover correctly?

M11.1 [**Manual Inspection**] $\frac{\text{\# discovered apps}}{\text{\# existing apps}}$

TP	Matching changed abstract action and active AIP exist
FP	Active AIP without matching abstract action exist
FP	Active AIP with matching unchanged abstract action exist
FN	Changed abstract action without matching AIP exist
FN	Changed abstract action with matching deactivated AIP exist

Table 5.3.: The classification of matching results regarding adaptive action instrumentation for the calculation of an F-Score.

M11.2 The previously presented metric **M8.1** compares a manually created and validated model to an automatically created model. During the creation of the manual model, SICK AppSpace apps were found and mapped to components in the PCM repository model. A correct component detection is therefore implied by the correct matching of the components.

5.3. Case Study Requirements and Application Concepts

This section discusses requirements and applications concepts for the case studies that were used for the evaluation of the prototypical implementation.

The following requirements should be satisfied by the case study application concepts:

REQ1 The case study is an application implemented in the SICK AppSpace ecosystem using the Lua programming language.

REQ2 The case study has a sufficiently long development history, reflected by a Git repository with several commits.

REQ3 The case study is a real world application for use on programmable sensor nodes and SIMs.

REQ4 The case study contains multiple components which can be deployed using multiple allocations on the hardware nodes.

Based on these requirements we devised two applications concepts for the case studies of the evaluation: When the satisfaction of a requirement by the application concept is contentious, additional discussion is provided.

CSAC1 Realistic application using SICK AppSpace samples
SICK provides a large number of small SICK AppSpace application samples to the public [40]. Each sample demonstrates a specific functionality, e.g. how to read a barcode using a sensor. Theses samples can be combined to implement a realistic application, that could be used in an industrial setting.

Satisfaction of REQ2 The SICK AppSpace samples do not have sufficiently long development histories because of their compact nature. In addition, combining the samples into a functioning application is not expected to require enough code to yield a sufficiently long development history either. To satisfy **REQ2** the

following proposition is made:
Merging the repositories of the used SICK AppSpace samples into the repository of the application will result in a single repository. Under the assumption that this repository reflects the development history of both the samples and the application, the development history should be sufficiently long to satisfy **REQ2**. While the development history is artificial, it is not unrealistic per se. The development history would reflect the development using a strictly component-based architecture. This makes this case study candidate attractive because the PCM described in section 2.5 is intended for use with component-based architectures.

Satisfaction of REQ3 The application is implemented for the purpose of evaluating the CIPM approach and therefore artificial in nature. It is still a realistic application for use in an industrial Internet of Things (IoT) use case. Running the application on the intended sensor hardware is feasible. This makes this a credible candidate for study with regard to **REQ3**.

CSAC2 Real-world application from the SICK AppPool

SICK sells fully featured applications ("SensorApps") through the SICK AppPool [38]. These applications can be used by customers without customization and are in use in industrial settings. The applications from the SICK AppPool are more realistic than the application described in **CSAC1**.

Satisfaction of REQ2 The applications from the SICK AppPool are fully featured and usually have a sufficiently long development history to satisfy **REQ2**. The studying the "real" development history is attractive, as it would make the case study more credible. A significant downside is, that this development history is not available to third parties to reproduce the evaluation.

Satisfaction of REQ4 The applications from the SICK AppPool are usually not implemented to run on multiple hardware nodes. The applications usually contain only one app and would be therefore be modelled using one component in our case. As the component based architecture modelling approach we use only considers external calls, this is an issue because no external calls would exist for this. We intended to split a real-world application into multiple apps to better satisfy this requirement. Because of the time constraints of this thesis we could not split an application into multiple apps. Instead we, changed the reconstruction mapping of function calls to actions to simulate the splitting of the application into multiple apps. This is further discussed in section 5.7.

REQ	CSAC1	CSAC2
REQ1	✓	✓
REQ2	(✓)	(✓)
REQ3	(✓)	✓
REQ4	✓	(✓)

Table 5.4.: Requirement satisfaction of the case study application concepts.

An overview over the application concepts with regard to the satisfaction of the previously presented requirements is depicted in Table 5.4. Check marks in brackets denote contentious requirements. These are discussed in the previous listing.

5.4. Case Studies

This section describes the example applications which serve as case studies for the evaluation of the adapted CIPM approach.

5.4.1. Case Study 1

For case study 1 the author of this thesis developed an example application using **CSAC1**. The example application can take images using a camera module, detect barcodes in it and send this information to another device for storage in a database using HTTP. The application consists of 4 SICK AppSpace samples which in unison represent the application for the evaluation. We used the following SICK AppSpace samples:

- A barcode scanner app that can be used to take images from a camera module detect barcodes in the image. We simplify the setup of the application by not working with the camera module, but instead a directory image provider that yields images from a storage directory on the device.
- A HTTP client App that provides a serve call to submit the information contained in the barcode to a webserver.
- A HTTP server App that enables the webserver functionality of the underlying SIM and hooks into received information from the be client. Once a new set of information is received the information is printed to the command line and the information is inserted forwarded to the Database App
- The database application provides a function to the HTTP Server app, that allows the insertion of infos about a new barcode, like its type and value into the database. In addition, the database app provides a web user interface so users can see which barcodes were scanned by the application.

A component diagram of the application can be seen in Figure 5.1. The four previously described apps are deployed on two app engines *A* and *B*. After a new image is captured by the hardware the `NewImage` event is sent to the BarcodeReader app. This causes the app to detect barcodes in the image and extracts the data from the barcode. The app submits each data objects to the HTTPClient app using a *direct* CROWN call. The HTTPClient app in turn creates a HTTP request using the data object and submits it to the HTTP implementation that is provided of by AppEngine *A*. AppEngine *A* executes the HTTP request in which AppEngine *B* acts as the server. *B* is configured to pass HTTP requests through to the HTTPServer app. The HTTPServer app prints information about the received request to the command line and submits the received data to the DatabaseAPI. The DatabaseAPI is called using a direct CROWN call and inserts the data object into the underlying database. Finally, AppEngine *B* make the web frontend available which is provided by the DatabaseAPI app and allows users to inspect which barcodes where scanned.

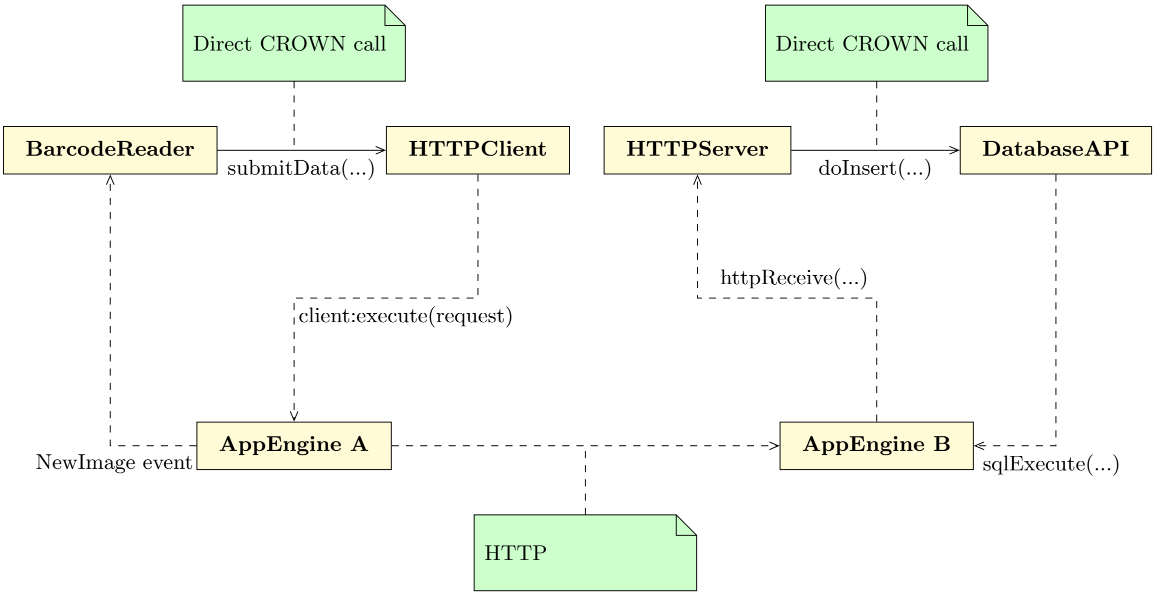


Figure 5.1.: The components of Case Study 1 and their interaction.

The investigated development history of case study consists of seven commits. An overview regarding the changes in commits can be seen in Table 5.5.

At commit 1 the application consists of only three apps: The BarcodeReader, the HTTPClient and the HTTPServer. In commit 2 the DatabaseAPI app is introduced into the application. Functionality that was part of the DatabaseAPI but not relevant for the case study is removed in commit 3 Commit 4 adds a conditional to prevent the submission of empty barcode API objects to the HTTPClient. Further, commit 5 prints a message if an empty API object was found. All serve calls are removed from the DatabaseAPI in commit 6. Subsequently, the DatabaseAPI app is removed from the application completely in commit 7.

This commit history was created to emulate the development process of a larger application. We therefore included the artificial deletion of the DatabaseAPI to more reactions would need to be executed during the propagation of the commit history.
All commits of the case study 1 development history are architecturally relevant.

5.4.2. Case Study 2

The second case study uses the **CSAC2**.

The application "Color Inspection and Sorting" from the SICK AppPool allows the detection of objects using attributes like their color. The image processing contains which multiple distinct operations: image resizing, image separation, blob detection and the processing of the blobs.

Commit Index	Hash	Added Lines	Removed Lines	Changed Files
1	e25fb6b	575	0	4
2	7126aab	204	0	1
3	d92b459	76	81	3
4	e6d87e0	4	1	1
5	542d2e9	2	0	1
6	6b7b35f	1	21	2
7	1f2fb08	0	180	1

Table 5.5.: The commits of case study 1

Commit Index	Total Files	Blank Lines	Comment Lines	Lines of Code
1	4	98	89	388
2	5	125	124	530
3	5	128	130	516
4	5	129	130	518
5	5	129	130	520
6	5	126	127	506
7	4	99	94	386

Table 5.6.: Characterisation of the case study 1 application for the given commits

We intended to split this real world app into multiple apps and therefore multiple components for each of the aforementioned distinct operations. Because of time constraints, we were unable to split the case study 2 application into multiple apps / components. We still use the application for the evaluation of the approach to evaluate the approach using a real world application, even if a single component application is not really the intended use case for the adapted CIPM approach.

The complete development history of the application up to its current master version contains 47 commits. From those 47 commits we filtered out commits that do not change a Lua source file.

The selected commits of the development history of case study 2 is depicted in Table 5.7. The changes in the selected commits are described in Table 5.8. The total amount of Lua files and content in terms of Lines of Code (LOC) are listed in Table 5.9.

5.5. Experiments

This section lists the experiments which were conducted to acquire the metrics for the GQM plan. The propagation of case study 1 which forms experiment one discussed in subsection 5.5.1. Experiment 2, the propagation of case study 2 is further, discussed in subsection 5.5.2.

Commit Index	Hash	Added Lines	Removed Lines	Changed Files
1	f713180	1855	0	7
2	1466c57	3	1	2
3	f57823c	10	4	3
4	40bf36e	451	122	7
5	473c9d9	1	1	1
6	995ecc0	505	187	7
7	956aeb9	448	234	7
8	0da3169	88	19	3
9	00b44f8	60	44	3
10	88d005a	41	14	1
11	92cb3bc	3188	2036	13
12	916fc52	1	1	1

Table 5.7.: The selected git commit history of case study 2.

Commit	Description
1	Initial import of existing application into new repository
2	This commits contains the minor changing of some texts.
3	In this commit a bug was fixed in the application.
4	New features for the applications, e.g. the handling of more objects and jobs.
5	Minor bug fix
6	New features including different LED behaviour and a different camera trigger mode
7	Minor bug fix, v2.6.0
8	New features, improvements and bug fixes, v2.7.0
9	Improvements of the UI, v3.0.0
10	New feature and bug fixes, v3.1.0
11	Refactoring of the code structure, compatibility with new hardware, 10 new features
12	Minor bug fix

Table 5.8.: The selected git commit history of case study 2.

5.5.1. E1: Propagation of Case Study 1

This section describes experiment E1 and lists and discusses its results.

Goal of experiment E1 was to acquire metrics for the GQM plan using case study 1, see subsection 5.4.1. The commits with index 1 to 7 were used for the experiment.

The experiment was executed using a version that did not implement the changed statement reactions as described subsection 4.5.6.

The experiment was conducted as follows:

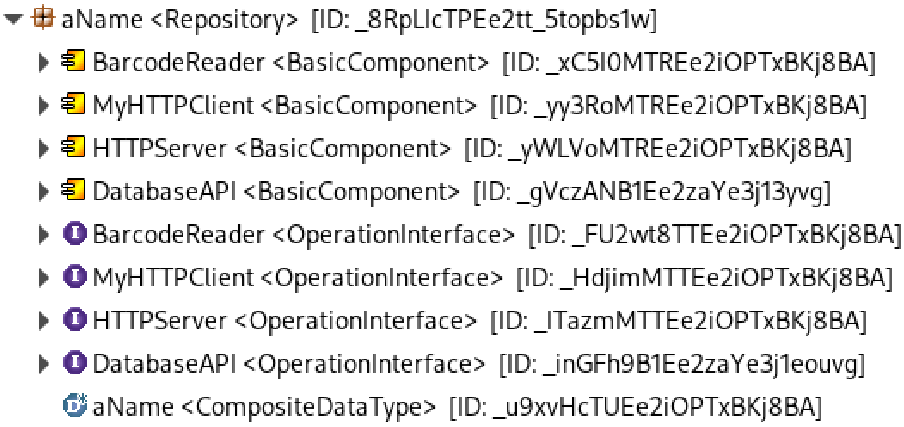
Commit Index	Total Files	Blank Lines	Comment Lines	Lines of Code
1	7	261	291	1303
2	7	261	291	1305
3	7	262	291	1310
4	7	292	322	1578
5	7	292	322	1578
6	7	313	481	1716
7	7	338	489	1897
8	7	342	494	1957
9	7	346	496	1967
10	7	350	498	1988
11	10	487	697	2804
12	10	487	697	2804

Table 5.9.: Total source code size for the commits of case study 2.

1. For each commit: Start with an empty VSUM and propagate the commit. After the propagation a copy of the commit integration state is stored persistently. The commit integration state consists among other things all the models which are part of the VSUM. In addition models, such as the parsed code model are also part of the commit integration state. This yields models versions for each commit, which we'll refer to as the "automatic" models.

2. We start with another empty VSUM. Then we propagate all the commits from first to last to the VSUM. After each propagation another copy of the commit integration state in created for the purpose of the evaluation.

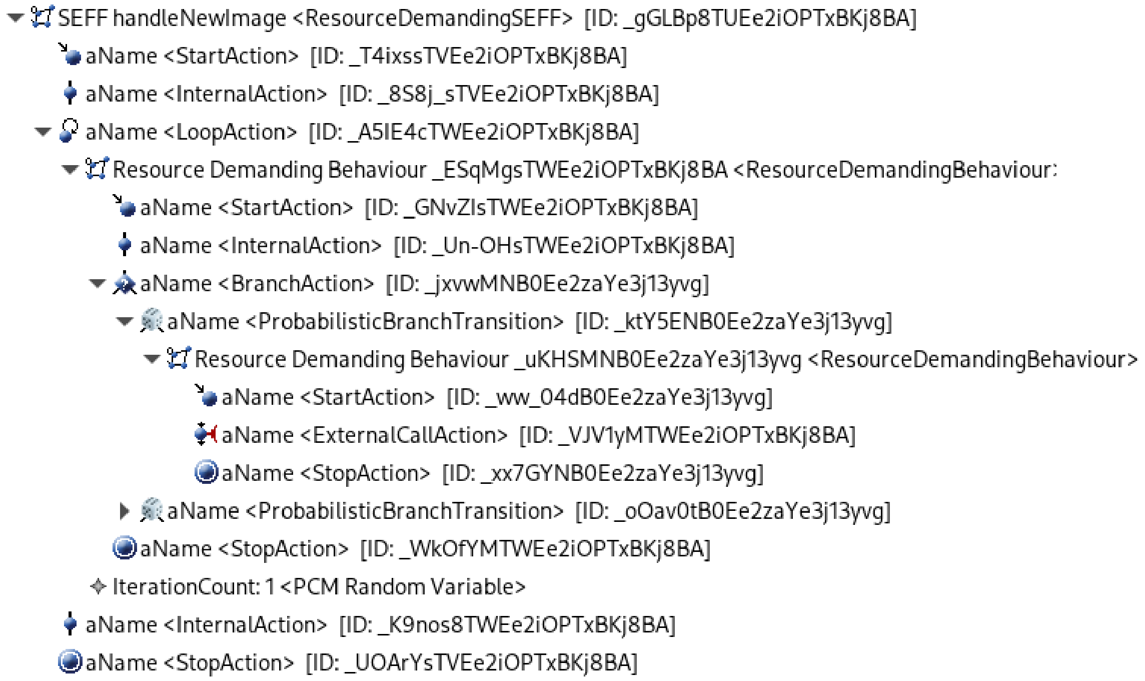
3. Having created all needed model versions, we evaluate all the propagations of the previous step and calculate the required metrics for the GQM plan.



aName <Repository> [ID: _8RpLIcTPEe2tt_5topbs1w]
BarcodeReader <BasicComponent> [ID: _xC5I0MTREe2iOPTxBKj8BA]
MyHTTPClient <BasicComponent> [ID: _yy3RoMTREe2iOPTxBKj8BA]
HTTPServer <BasicComponent> [ID: _yWLVoMTREe2iOPTxBKj8BA]
DatabaseAPI <BasicComponent> [ID: _gVczANB1Ee2zaYe3j13yvg]
BarcodeReader <OperationInterface> [ID: _FU2wt8TTEe2iOPTxBKj8BA]
MyHTTPClient <OperationInterface> [ID: _HdjimMTTEe2iOPTxBKj8BA]
HTTPServer <OperationInterface> [ID: _ITazmMTTEe2iOPTxBKj8BA]
DatabaseAPI <OperationInterface> [ID: _inGFh9B1Ee2zaYe3j1eouvg]
aName <CompositeDataType> [ID: _u9xvHcTUEe2iOPTxBKj8BA]

Figure 5.2.: An exemplary overview over the contents of the repository after the propagation of commit 5 of case study 1.

An exemplary overview over an instance of PCM repository model can be seen in Figure 5.2. The Repository instance contains the four BasicComponents and OperationInterfaces that were reconstructed for the case study 1 application. Is addition the composite data type that is used by the approach to model *any* type can be seen.



```
SEFF handleNewImage <ResourceDemandingSEFF> [ID: _gGLBp8TUEe2iOPTxBKj8BA]
  aName <StartAction> [ID: _T4ixssTVEe2iOPTxBKj8BA]
  aName <InternalAction> [ID: _8S8j_sTVEe2iOPTxBKj8BA]
  aName <LoopAction> [ID: _A5IE4cTWEe2iOPTxBKj8BA]
    Resource Demanding Behaviour _ESqMgsTWEe2iOPTxBKj8BA <ResourceDemandingBehaviour:
      aName <StartAction> [ID: _GNvZIsTWEe2iOPTxBKj8BA]
      aName <InternalAction> [ID: _Un-OHsTWEe2iOPTxBKj8BA]
      aName <BranchAction> [ID: _jxvwMNB0Ee2zaYe3j13yvg]
        aName <ProbabilisticBranchTransition> [ID: _ktY5ENB0Ee2zaYe3j13yvg]
          Resource Demanding Behaviour _uKHSMNB0Ee2zaYe3j13yvg <ResourceDemandingBehaviour>
            aName <StartAction> [ID: _ww_04dB0Ee2zaYe3j13yvg]
            aName <ExternalCallAction> [ID: _VJV1yMTWEe2iOPTxBKj8BA]
            aName <StopAction> [ID: _xx7GYNB0Ee2zaYe3j13yvg]
        aName <ProbabilisticBranchTransition> [ID: _oOav0tB0Ee2zaYe3j13yvg]
      aName <StopAction> [ID: _WkOfYMTWEe2iOPTxBKj8BA]
    IterationCount: 1 <PCM Random Variable>
  aName <InternalAction> [ID: _K9nos8TWEe2iOPTxBKj8BA]
  aName <StopAction> [ID: _UOArYsTVEe2iOPTxBKj8BA]
```

Figure 5.3.: An exemplary SEFF of the previous repository after the propagation of commit 5 of case study 1.

In addition, one SEFF of the previously depicted Repository can be seen in Figure 5.3. The `handleNewImage` SEFF exhibits the used type of `AbstractAction`: StartAction, StopAction, InternalAction, LoopAction, BranchAction and ExternalCallAction.

5.5.1.1. Results and Analysis

This section will list the results of experiment E1, split by goals and metrics according to the GQM plan presented in section 5.2.

G1.1: Code Model The obtained code model was evaluated regarding its correctness and regarding that the approach correctly updates code model instances of the VSUM.

M4: Code Model Correctness This metric is used to determine if the code model used for the approach is correct. The result of experiment E1 with regard to this metric are displayed in Table 5.10.

No identical or dissimilar were found by the comparison. All parsed files of all commits were classified as *similar* by the textual comparison. As explained in the GQM plan, the Xtext printer may not reproduce lines with comments and white space of the original source file correctly. This seems to be the case

Commit Index	Total Files	Identical Files	Similar Files	Dissimilar Files
1	4	0	4	0
2	5	0	5	0
3	5	0	5	0
4	5	0	5	0
5	5	0	5	0
6	5	0	5	0
7	4	0	4	0

Table 5.10.: The results of experiment E1: Correctness of the code model.

for every commit, hence no printed file is identical to its original counterpart. After the filtering of the comment lines, etc. all files are equal, leading to a classification as similar. The semantically relevant source code is reproduced by the parsing and printing.

This indicates that the approach can obtain a correct code model for each commit of case study 1. Consequently, **G1.1** was achieved regarding case study 1.

M5: Code Model Update We determine if the prototypical implementation correctly updates the code model of the VSUM using this metric. The result of experiment E1 with regard to this metric are listed in Table 5.11.

Commit Index	Unmatched	Intersection	Union	Jaccard Coefficient
1	0	1746	1746	1
2	0	2195	2195	1
3	0	2144	2144	1
4	0	2151	2151	1
5	0	2155	2155	1
6	0	2097	2097	1
7	0	1726	1726	1

Table 5.11.: The results of experiment E1: Correctness of the code model update process.

The calculated Jaccard Coefficient is 1 for every propagation. Consequently, the intersection and the union of the model comparison have identical cardinality after every propagation. This indicates on the one hand that the change derivation approach using the hierarchical matching can correctly match the code model instances and obtain the changes between the models. On the other hand this indicates that the code model of the VSUM is updated correctly.

As shown by the previous two metrics, the code model is both correct and changes to it can be correctly derived. The code model therefore be used for the wider approach. This experiment consequently indicates that **G1.1** was achieved regarding case study 1.

G1.2.1: CPRs between the Lua code model and the PCM The CPRs between the Lua code model and the PCM were evaluated by comparing the updated PCM instances to manually and automatically created reference model instances:

M8.1: Manually created Reference Model We manually created and validated PCM Repository Models for the commits 1 and 5 of the case study. These manually created reference models where compared with the PCM repository model which was updated by the prototypical implementation during the propagation of commits 1 to 5. The result of the model comparisons are listed in Table 5.12.

Commit Index	Unmatched	Intersection	Union	Jaccard Coefficient
1	0	50	50	1
5	0	121	121	1

Table 5.12.: Results of experiment E1: Correctness of the CPRs between the Lua code model and the PCM repository model measured by comparison with manually created reference models.

Commit 1 is the initial commit of the case study. Therefore, no change derivation is needed during the propagation of commit 1. Instead commit 1 can is used to analyse if the prototypical implementation can statically reverse engineer the PCM repository model for the initial version of the case study application. Table 5.12 indicates that the PCM repository model reverse engineered by the prototypical implementation matches the manually created reference model completely, which indicates that the prototypical implementation can correctly reverse engineer the repository model for case study 1.

Commit 5 was selected for this evaluation, because the PCM repository model contains the most objects after this commit as can be seen in Table 5.13. The correct propagation of commit 5 to the VSUM requires the correct functioning of both the change derivation implemented in the prototypical implementation and the correct implementation of the CPRs, in particular the SEFF update. The change derivation further relies on the custom matching as previously described.

As can be seen in Table 5.12 the manually created reference model did match the PCM repository model updated by the prototypical implementation. This indicates that the prototypical implementation updated the PCM repository model correctly for the propagation of the commits 2 to 5. This result implies that the change derivation and consequently the implemented hierarchical matching works correctly for the case study 1 application.

M8.2: Automatically created Reference Model As we were unable to manually create and validate reference models for all versions of case study 1, we extended the evaluation of the CPRs from the Lua code model to the PCM repository model: We automatically create a PCM repository model for each commit of case study 1 by propagating this single commit into an empty Vitruv VSUM. The models which are obtained by this process are then compared to the actual VSUM

Repository models, which were updated commit by commit. The result of the model comparisons are listed in Table 5.13.

Commit Index	Unmatched	Intersection	Union	Jaccard Coefficient
1	0	50	50	1
2	0	77	77	1
3	0	111	111	1
4	0	120	120	1
5	0	121	121	1
6	0	68	68	1
7	0	60	60	1

Table 5.13.: Results of experiment E1: Correctness of the CPRs between the Lua code model and the PCM repository model measured by comparison with an automatically created reference model.

The model comparison yielded an Jaccard Coefficient of 1 for the commit propagation of all commits. This indicates that the PCM repository models that are updated by the adapted approach match the automatically created PCM repository models. As further expanded upon in subsection 5.7.1, we can make no strong assertions regarding the correct updating of the PCM repository model, because the compared models were created by the same set of CPRs. Still, these CPRs operate differently when creating new objects in the PCM repository model compared to when they update already existing objects. So different code paths may have been compared to each other, depending on the nature of the propagated changes. Comparing Table 5.13 with Table 5.12, one can see that for the commits 1 and 5 the object count for the intersection and the union are identical, as expected.

Because of the time constrains of this thesis we were unable to create manual reference models for all commits of case study 1. Instead we manually created and validated two reference models for the the first commit and the commit with PCM repository model with the most elements. In addition we compared the repository models that were updated by the prototypical implementation with repository models that were automatically created by the prototypical implementation for this specific commit.

The comparison of the manual reference model for commit 1 showed no difference between the reference model and the model reverse engineered by the prototypical implementation. This leads us to believe that the reverse engineering process used to create the *automatic* reference models is working correctly. The results of the evaluation regarding **M8.2** indicate that architectural changes to the code model are propagated PCM repository model correctly by the prototypical implementation. The prototypical implementation therefore achieved **G1.2.1**.

G1.2.2: CPRs to the IM The following metrics were acquired to determine if the CPRs update the instrumentation model correctly,

M9.1: Instrumentation of Services This metric was acquired to determine if services as described by SEFFs in the PCM repository model, are able to be instrumented using the instrumentation model. This requires the existence of a SIP in the instrumentation model for each SEFF of the PCM repository model.

The result of experiment E1 with regard to this metric are listed in Table 5.14. The calculated F-Score for the matching SEFFs and SIPs is 1 for all propagations.

Commit Index	SIPs	Matched SIPs	F-Score
1	5	5	1
2	6	6	1
3	16	16	1
4	16	16	1
5	16	16	1
6	6	6	1
7	5	5	1

Table 5.14.: Results of experiment E1: Correctness of the CPRs between the PCM repository model and the instrumentation model, with regard to the instrumentation of services.

The results displayed in Table 5.14 indicate that SIPs were correctly created and removed for each SEFF of the PCM repository model and consequently that the change propagation from the PCM repository model to the instrumentation model works correctly with regard to the service instrumentation.

M9.2: Instrumentation of Actions The result of experiment E1 with regard to this metric are listed in Table 5.15. The calculated F-Score for the matching actions and AIPs is 1 for all propagations.

Commit Index	AIPs	Matched AIPs	F-Score
1	9	9	1
2	10	10	1
3	17	17	1
4	18	18	1
5	19	19	1
6	12	12	1
7	11	11	1

Table 5.15.: Results of experiment E1: Correctness of the CPRs between the PCM repository model and the instrumentation model, with regard to the instrumentation of actions.

As depicted in Table 5.15, AIPs were correctly created or removed for the actions of the PCM repository model. This indicates that the changes to the PCM repository model are correctly propagated to the instrumentation model with regard to the AIPs.

M10: Instrumentation of Changed Actions This evaluation is used to asses if the the instrumentation points are activated and deactivated correctly during the change propagation process of the prototypical implementation. Only actions which were added or changed since the last instrumentation and monitoring cycle need instrumentation. The result of experiment E1 with regard to this metric are listed in Table 5.16.

Commit Index	AIPs	Active AIPs	Matched Active AIPs	Added Actions	Changed Actions	Active AIP Ratio	F-Score
1	9	9	9	9	0	1	1
2	10	1	1	1	0	0.1	1
3	17	9	9	8	1	0.529	1
4	18	2	2	2	0	0.111	1
5	19	1	1	1	0	0.053	1
6	12	0	0	0	0	0	-
7	11	0	0	0	0	0	-

Table 5.16.: Results of experiment E1: Correctness of the CPRs between the PCM repository model and the instrumentation model, with regard to the instrumentation of actions since the last instrumentation.

The calculated F-Score for the matching actions and AIPs is 1 for the first 5 propagations. This indicates that all added or changed actions could be matched to an active AIP and vice versa. For the commits 6 and 7 no changed or added actions exist. Therefore no F-Score can be calculated, because the F-Score is undefined if no *true positives* exist (see subsection 5.1.2). Further, no active AIPs exists as expected.

This result indicates that the prototypical implementation correctly deactivated and activated the instrumentation points of the instrumentation model. Consequently the instrumentation model can be used to further implement the adaptive instruction as was intended for this thesis.

The results of the experiment regarding this goal indicate that the changes to the PCM repository model are correctly propagted to the instrumentation model by the prototypical implementation. Further, the instrumentation points are activated and deactivated correctly. Hence, **G1.2.2** was achieved by the prototypical implementation.

G1.2: CPRs We additionally evaluated how many CPRs are executed during the execution of the experiment.

M7: Ratio of executed Reactions This metric serves as an indicator for how much of the implemented functionality of the reactions was actually executed during this experiment.

The coverage results for the implemented reactions from the Lua code model to the PCM repository model and the instrumentation model implemented can be seen in the appendix in Figure A.1. All reactions except the `RemovedBlockReturnReaction` were executed during the propagation of case study 1. The total instruction coverage is 93% while the total branch coverage of the reactions is 81%.

The reaction coverage from the PCM repository model to the instrumentation model are depicted further in Figure A.2. All reactions were executed. A instruction coverage of 95% and a branch coverage of 84% was achieved.

The coverage data shows that all implemented reactions except one were executed during experiment 1. The `RemovedBlockReturnReaction` was not executed because the case study does not contain a commit, that changes a function with a return value into a function without one.

While this coverage result does not indicate that all the covered reactions are correct, only one reaction was not executed, which conceals all possible errors for this reaction.

G2: Detection of SICK AppSpace Apps A manual inspection was used to evaluate the correct component detection.

M11.1: Manual Inspection The number of existing apps and the amount of created PCM components after each propagation is listed in Table 5.17.

After the propagation of every commit the number of PCM components matches the number of existing SICK AppSpace Apps in the repository.

Commit Index	Existing Apps	Created PCM Components
1	3	3
2	4	4
3	4	4
4	4	4
5	4	4
6	4	4
7	3	3

Table 5.17.: The results of experiment E1: Existing Apps and created PCM Components.

As shown by the previous table, all existing Apps were detected for every commit. This indicates that the prototypical implementation can correctly detect Apps and map them to PCM components.

G1: Overall Approach Reduction of Monitoring Overhead: M2 This metric was calculated to quantify the potential saving of monitoring overhead of the instrumentation of action instrumentation points that is achievable with the the adapted CIPM approach. The results are displayed in Table 5.18.

AIPs	Active AIPs	TIPA	1 - TIPA
96	22	0.229	0.771

Table 5.18.: Results of experiment E1: Potential reduction of monitoring overhead.

The propagation of the 7 commits of case study 1 would have caused the instrumentation of 96 action instrumentation points without the adaptive instrumentation of the CIPM approach. Instead only 22 action instrumentation points needed to be instrumented during the propagation of the case study. Instrumentation points require a different number of instrumentation statements depending on their type. Therefore no fine-grained assertion can be made regarding the reduction of the actual monitoring overhead. Still, the monitoring overhead is correlated with the amount of instrumentation points and therefore the 77.1% reduction of activated action instrumentation points indicates a significant reduction of the monitoring overhead.

Execution Time: M3 We measured the execution time of the change propagation of the commits of case study 1. The used hardware uses an AMD Ryzen 7 5850U 8 core processor, 32GiB of system memory and an SSD. The execution times of each commit can be seen in Table 5.19. Most propagations take around 250ms to execute. Outliers are the commits 2 and 3 with a change propagation time of 1092ms and 772ms respectively. The propagation time per propagated changes varies significant between 0.056ms and 4.216ms. Between 51 and 4832 changes are propagated for the commits.

Commit	Time (ms)	Time per Change (ms)	Propagated Changes			
			Total	Lua	Repository	IM
1	269	0.056	4832	4551	209	72
2	1092	0.847	1290	1161	110	10
3	772	0.622	1241	901	242	93
4	253	2.433	104	32	50	14
5	215	4.216	51	10	34	5
6	251	1	251	145	83	22
7	201	0.235	855	833	18	4

Table 5.19.: Execution time and numbers of changes of the change propagation of the commits of case study 1

During the change propagation, changes to the Lua code model are derived and propagated to the PCM repository model and the instrumentation model of the VSUM using CPRs.

The propagation of commit 1 is significantly faster per propagated change than the other commits because no change derivation is needed for the first commit. Instead a repository is reverse engineered statically.

Each commit is propagated in significantly less than or around 1 second. We feel this is completely acceptable for the use in a commit integration pipeline. Because of the compact size of case study 1 no assertions can be made regarding larger real-world applications, based on this data.

As previously discussed the results of experiment 1 indicate that the subgoals **G1.1** and **G1.2** were achieved. Further, the evaluation regarding **M2** showed a significant reduction in the amount of instrumentation points that need instrumentation. The measured execution times characterised in the evaluation regarding **M3** show that the approach is able to propagate the commits of case study 1 with negligible delay.

Summing up, the satisfaction of the subgoals indicate that the prototypical implementation achieved **G1**.

5.5.2. E2: Propagation of Case Study 2

Experiment 2 follows the same procedure as experiment 1 which was described in subsection 5.5.1.

The prototypical implementation was slightly adapted since the execution of experiment 1. The previously described changed statement reaction was added for experiment 2.

5.5.2.1. Results and Analysis

This section will list the results of experiment E2, split by goals and metrics according to the GQM plan presented in section 5.2.

G1.1: Code Model The obtained code model was evaluated regarding its correctness and regarding that the approach correctly updates code model instances of the VSUM.

M4: Code Model Correctness This metric is used to determine if the code model used for the approach is correct. The result of experiment E2 with regard to this metric are displayed in Table 5.20.

All parsed files during the propagation of all commits were classified as *similar*. This indicates that the prototypical implementation correctly parses and prints all these files. Further, this indicates that the code model is correct.

M5: Code Model Update We determine if the prototypical implementation correctly updates the code model of the VSUM using this metric. The result of experiment E2 with regard to this metric are listed in Table 5.21.

For the commits 1 to 6 the Jaccard Coefficient (JC) is 1 which means that the parsed code models matches the code model updated by the VSUM. This indicates that the change derivation and the code model update process works correctly for these commits.

Commit Index	Total Files	Identical Files	Similar Files	Dissimilar Files
1	7	0	7	0
2	7	0	7	0
3	7	0	7	0
4	7	0	7	0
5	7	0	7	0
6	7	0	7	0
7	7	0	7	0
8	7	0	7	0
9	7	0	7	0
10	7	0	7	0
11	10	0	10	0
12	10	0	10	0

Table 5.20.: Results of experiment 2 regarding the code model correctness

Commit Index	Unmatched		Intersection	Union	JC
	Reference	VSUM			
1	0	0	6017	6017	1
2	0	0	6025	6025	1
3	0	0	6054	6054	1
4	0	0	7219	7219	1
5	0	0	7219	7219	1
6	0	0	7892	7892	1
7	37	37	8756	8830	0.9916
8	37	37	9074	9148	0.9919
9	37	37	9108	9182	0.9919
10	37	37	9160	9234	0.992
11	354	354	12712	13420	0.9472
12	354	354	12712	13420	0.9472

Table 5.21.: code model update

Commits 7 to 12 present a different picture. The JC is close to 0.992 for the commits 7 to 10. 37 unmatched elements were found for all of these commits in both the automatically created reference model and the VSUM code model. Manual inspection of the models reveals that these unmatched elements are the same elements for commits 7 to 10. Further inspection shows that these unmatched elements are represent statement that is *out of order* in the VSUM code model.

A JC of 0.9472 for was achieved for the commits 11 and 12. Again 354 unmatched elements were found both in the automatically created reference and the VSUM code model. Like with the commits 7 to 10, closer inspection reveals that these mismatched elements correspond to statements that are at an incorrect position in their containing block.

Because of the time constraints of this thesis, we could only perform a limited investigation regarding the code model elements that are created correctly but appear out of order in the VSUM code model. The prototypical implementation uses EMF compare to find the differences between the old VSUM code model and the newly parsed code model. These differences are then merged onto the old VSUM code model while the changes to this model are recorded. The recorded changes then represent the change sequence between the models.

EMF compare find the differences between the models by matching them against each other. The used matching is the hierarchical matching approach described in subsection 4.5.5. We suspect that the matching implemented in the prototypical implementation does not match the out of order elements correctly for some reason. This matching is used during the initial model matching *and* the merging phase of the change derivation. During the merging the matching is used to determine where to insert an element into an container. An insertion index is calculated by finding the longest common subsequence of elements of the container in the model which is currently merged onto and the newly parsed code model. The incorrect matching seems to cause EMF compare to not find any common subsequence, which results in the element being inserted at the end of its container.

Concluding, the VSUM code model is not correctly updated during all propagations. We feel that this represents a minor technical difficulty in the custom Lua matching. Given more time, we would have likely found the underlying matching issue, which would then result in the correct updating of the VSUM code model. This represents no conceptual difficulties of the approach, but only technical difficulties with the prototypical implementation.

The discussion of **M4** showed that the code model is correct. The evaluation of the code model update revealed that the code model is updated correctly in nearly all cases. For a small portion of code model elements, we experience matching issues, which causes the incorrect updating of the code model. Therefore, **G1.1** was not achieved completely with regard to case study 2. Minor issues with the prototypical implementation remain, which could likely be fixed by future work.

G1.2.1: CPRs between the Lua code model and the PCM The CPRs between the Lua code model and the PCM were evaluated by comparing the updated PCM instances to automatically created reference model instances:

M8.2: Automatically created Reference Models We automatically create a reference PCM repository model for each commit of case study 2 by propagating this single commit into an empty Vitruv VSUM. The models which are obtained by this process are then compared to the actual VSUM Repository models, which were updated commit by commit. The result of the model comparisons are listed in Table 5.22.

Index	Unmatched		Intersection	Union	JC
	Reference	VSUM			
1	0	0	600	600	1
2	0	0	600	600	1
3	0	0	600	600	1
4	0	0	693	693	1
5	0	0	693	693	1
6	0	0	745	745	1
7	2	1	810	813	0.9963
8	2	1	831	834	0.9964
9	2	1	841	844	0.9964
10	2	1	847	850	0.9965
11	11	9	632	652	0.9693
12	11	9	632	652	0.9693

Table 5.22.: Results of experiment E2: Correctness of the CPRs between the Lua code model and the PCM repository model measured by comparison with an automatically created reference model.

The PCM repository model matches the automatically created reference model for the commits 1 to 6. For the commits 7 to 12 the PCM repository model updated by the CPRs does not match the automatically created reference. This is to be expected as the update of the PCM repository model is based on the changes to the code model, which we discovered in the evaluation of the previous metric does not work correctly for these exact commits. In this previous evaluation we discovered that some statements were placed out of order in their container during the code model update. This caused the existence of an equal amount of unmatched elements in the reference and the updated model. Regarding the results in Table 5.22, we find an unequal amount of unmatched elements for the reference and the VSUM PCM repository model. The unmatched elements are caused by the incorrect order of the elements in the code model. Further, the unequal amount of unmatched elements is explained by the internal action fusing described in subsection 4.5.6: Depending on the ordering adjacent internal actions are fused or not, which causes the unequal amount of unmatch elements. Manual inspection of some of the mismatched elements reveals that the incorrect order of the source elements in the code model cause the incorrect update of the PCM repository model in the VSUM. We suspect that the CPRs from the Lua code model to the PCM repository model work correctly and all mismatched elements are caused by the incorrect update of the code model.

Because of the time constraints of this thesis, we were unable to create a manual reference model for commits of case study 2, like we did for case study 1. The evaluation of the CPRs from the Lua code model to the PCM repository model is hindered by the incorrect code model discussed regarding the previous goal. Because of this we can make no strong assertions regarding the correct operation of these

CPRs. Still, we suspect that the CPRs actually work correctly. We therefore suspect that the prototypical implementation achieved **G1.2.1** with regard to case study 2. Future work could investigate this suspicion further, by fixing the code model update process and repeating this experiment.

G1.2.2: CPRs to the IM The following metrics were acquired to determine if the CPRs update the instrumentation model correctly,

M9.1: Instrumentation of Services This metric was acquired to determine if services as described by SEFFs in the PCM repository model, are able to be instrumented using the instrumentation model. This requires the existence of a SIP in the instrumentation model for each SEFF of the PCM repository model.

The result of experiment E2 with regard to this metric are listed in Table 5.23. The calculated F-Score for the matching SEFFs and SIPs is 1 for all propagations.

Commit Index	SIPs	Matched SIPs	F-Score
1	119	119	1
2	119	119	1
3	119	119	1
4	137	137	1
5	137	137	1
6	148	148	1
7	162	162	1
8	166	166	1
9	168	168	1
10	170	170	1
11	111	111	1
12	111	111	1

Table 5.23.: Results of experiment E2: Correctness of the CPRs between the PCM repository model and the instrumentation model, with regard to the instrumentation of services.

The results displayed in Table 5.23 indicate that SIPs were correctly created and removed for each SEFF of the PCM repository model and consequently that the change propagation from the PCM repository model to the instrumentation model works correctly with regard to the service instrumentation.

M9.2: Instrumentation of Actions The result of experiment E2 with regard to this metric are listed in Table 5.24.

The calculated F-Score for the matching actions and AIPs is 1 for all propagations. AIPs were correctly created or removed for the actions of the PCM repository model. This indicates that the changes to the PCM repository model are correctly propagated to the instrumentation model with regard to the AIPs.

M10: Instrumentation of Changed Actions The result of experiment E2 with regard to this metric are listed in Table 5.25.

Commit Index	AIPs	Matched AIPs	F-Score
1	69	69	1
2	69	69	1
3	69	69	1
4	78	78	1
5	78	78	1
6	86	86	1
7	90	90	1
8	93	93	1
9	94	94	1
10	91	91	1
11	103	103	1
12	103	103	1

Table 5.24.: Results of experiment E2: Correctness of the CPRs between the PCM repository model and the instrumentation model, with regard to the instrumentation of actions.

Commit Index	AIPs	Active AIPs	Matched Active AIPs	Added Actions	Changed Actions	Active AIP Ratio	F-Score
1	69	69	69	69	0	1	1
2	69	1	1	1	0	0.0145	1
3	69	2	2	2	0	0.029	1
4	78	29	29	29	0	0.3718	1
5	78	0	0	0	0	0	-
6	86	37	37	37	0	0.4302	1
7	90	36	36	35	1	0.4	1
8	93	11	11	11	0	0.1183	1
9	94	7	7	7	0	0.0745	1
10	91	5	5	5	0	0.0549	1
11	103	76	76	76	1	0.7379	0.9935
12	103	5	5	5	0	0.0485	1

Table 5.25.: Results of experiment E2: Correctness of the CPRs between the PCM repository model and the instrumentation model, with regard to the instrumentation of actions since the last instrumentation.

The F-Score is 1 for all commits of case study 2 except 5 and 11. For the commit 5 no F-Score can be calculated as no actions were added or changed. During the propagation of commit 11, 76 actions were added to the PCM repository model of the vsum. One action of the PCM repository model was changed by the CPRs. The AIP corresponding to this changed action was not activated correctly. We could not execute an in-depth investigation the cause of this because of the time constraints of this thesis. Commit 11 is the largest commit of case study 2 and changes large portions of the Color Inspection and sorting app. We suspect that this missing activation is the result of a missing corner case in the implementation of the CPRs from the Lua code model. Given more time, would have likely been able to fix this minor issue.

As presented, the CPRs correctly create and remove SIPs for PCM services and AIPs for actions in the PCM repository model. A minor issue with the implementation of the CPRs results in an AIP not being activated correctly. All other added actions and changed actions are activated correctly. Overall we feel that the prototypical implementation achieved **G1.2.2** with regard to case study 2, with the exception of one minor technical issue.

G1.2: CPRs In addition to the to the evaluation of the CPRs from the code model and to the instrumentation model we, evaluated how many CPRs were executed during the execution of the experiment.

M7: Ratio of executed Reactions The coverage results for the implemented reactions from the Lua code model to the PCM repository model and the instrumentation model implemented can be seen in the appendix in Figure A.3. All reactions except the `RemovedLuaComponentReaction` and `RemovedBlockReturnReaction` were executed during the propagation of case study 2. The total instruction coverage is 80% while the total branch coverage of the reactions is 75%.

The reaction coverage from the PCM repository model to the instrumentation model are depicted further in Figure A.4. All reactions were executed. An instruction coverage of 86% and a branch coverage of 74% was achieved.

The coverage data shows that all implemented reactions except one were executed during experiment 1. The `RemovedBlockReturnReaction` was not executed because the case study does not contain a commit, that changes a function with a return value into a function without one. Further, the `RemovedLuaComponentReaction` was not executed because the evaluated commit history of case study 2 never removes a SICK AppSpace app.

The metric indicates that nearly all of the reactions were executed and consequently nearly all functionality of the CPRs was at least executed once.

G2: Detection of SICK AppSpace Apps This goal is achieved if all existing SICK AppSpace apps can be detected by the approach.

M11.1: Manual Inspection Case study 2 contains only one SICK AppSpace app during all the used commits. We manually verified that after each propagation one corresponding `BasicComponent` exists in the PCM repository model. This indicates that the prototypical implementation can correctly detect apps of the SICK AppSpace ecosystem and map them to PCM components.

G1: Overall Approach To evaluate the overall approach we estimate the reduction of the monitoring overhead that is achievable by the approach and measure the propagation time of commits.

Reduction of Monitoring Overhead: M2 This metric was calculated to quantify the potential saving of monitoring overhead of the instrumentation of action instrumentation points that is achievable with the the adapted CIPM approach. The results are displayed in Table 5.26.

AIPs	Active AIPs	TIPA	1 - TIPA
1023	270	0.264	0.736

Table 5.26.: Results of experiment E2: Potential reduction of monitoring overhead.

The propagation of the 12 commits of case study 2 would have caused the instrumentation of 1023 action instrumentation points without the adaptive instrumentation of the CIPM approach. Instead 270 action instrumentation points would have been instrumented during the propagation of the case study. As previously discussed in the evaluation of the CPRs to the instrumentation model, the prototypical implementation did not correctly activate one AIP during the propagation. Factoring in this missing activated AIP would result in the activation of 271 AIPs and reduction of the activated AIPs of 73,51%.

As previously described no fine-grained assertion can be made regarding the reduction of the actual monitoring overhead. The monitoring overhead is correlated with the amount of instrumentation points and therefore the 73.6% (or 73,51%) reduction of activated action instrumentation points indicates a significant reduction of the monitoring overhead.

Execution Time: M3 We measured the execution time of the change propagation of the commits of case study 2. The execution times of each commit can be seen in Table 5.27.

During the change propagation, changes to the Lua code model are derived and propagated to the PCM repository model and the instrumentation model of the VSUM using CPRs. The median propagation time is 1165.5 ms. Commit 11 is the commit with the most propagated changes and consequently it takes the longest to propagate with around 21.1 seconds. The propagations times show that the prototypical implementation can propagate changes relatively quickly regarding its intended use in a commit propagation pipeline.

Overall the prototypical implementation was able to operate as expected. Because of the time constraints of this thesis we were unable to overcome a few minor technical issues with the prototypical implementation: The code model update erroneously creates elements in an incorrect order in a small portion of the propagations. Because of this the evaluation of the CPRs from the Lua code model is hindered. The evaluation of the CPRs to the instrumentation model revealed a single case in which a changed action is not correctly activated.

Commit	Time (ms)	Time per Change (ms)	Propagated Changes			
			Total	Lua	Repository	IM
1	2092	0.074	28257	15759	7636	4862
2	568	3.463	164	47	27	21
3	577	0.645	894	859	20	14
4	3553	0.23	15418	12501	1756	1157
5	219	0.566	387	359	0	28
6	4023	0.244	16454	12069	2767	1616
7	7617	0.321	23727	17212	4013	2461
8	1488	0.179	8311	7104	704	462
9	843	0.38	2216	1758	283	162
10	605	0.602	1005	707	175	114
11	21126	0.48	44043	37638	4051	2339
12	479	0.251	1905	1702	76	51

Table 5.27.: Execution time and numbers of changes of the change propagation of the commits of case study 2

Summing up, the prototypical implementation falls short of achieving **G1** with regard to case study 2. Future work could sort out the remaining minor technical difficulties, which would allow the reevaluation of the prototypical implementation to determine if **G1** is finally achieved with regard to case study 2.

5.6. Discussion

This section will discuss the findings of the evaluation of the prototypical implementation of the CIPM approach for Lua-based sensor applications.

We used two case studies for the evaluation of the prototypical implementation: Case study 1 was created by us as a minimal running example, by combining multiple existing application samples from the SICK AppSpace ecosystem. The Color Inspection and Sorting app from the SICK AppPool was used as case study 2. The application of case study 1 contains 3 to 4 SICK AppSpace apps, while the case study 2 application contains a single app. Because component based applications with only one component exhibit no calls between components we changed the operation of the SEFF reconstruction for the propagation of experiment 1: Calls to SEFFs of the same component are reconstructed as if they were external function calls. This emulates the behaviour of case study 2, had we had the time to split the application into multiple apps as originally planned. Case study 1 is an near-real-world application, while case study 2 is a real-world application. The selected commit history of case study applications contains 7 and 12 commits respectively. The maximum extent with regard to the Lua source code of the case study applications during their development are 530 LOC and 2804 LOC.

We will now discuss the overall results of the evaluation based on the results of both experiments with regard to the achievement of the goals of the GQM plan presented in section 5.2.

G1.1: Code Model The results of both results experiments indicate that the contributed Lua code model is correct.

While experiment 1 showed no issues with the updating of the VSUM code model by the prototypical implementation, experiment 2 revealed that in a small portion of changes to the code model the ordering of `Statements` in a `Blocks` elements is incorrect. Our investigation of the misplaced statements indicates that the matching which we implemented for the Lua code model and the procedure by which EMF compare inserts elements into a model during the change derivation cause the misplaced elements. We feel that this is a minor technical issue, which could be remedied by e.g. extending the implemented matching. Overall, the prototypical implementation therefore achieved **G1.1** but still experiences minor technical difficulty which could be addressed by future work.

G1.2.1: CPRs between the Lua code model and the PCM No reference PCM repository models were available for the case study applications. Therefore the author of this thesis manually created and validated repository models for two commits of case study 1. Because of the time constraints of this thesis we could not create more reference models for case study 1 and especially case study 2, because of the extent of the case study application. Instead we automatically created reference models, by propagating only a single commit into an empty VSUM. The automatically created reference models were used for all commits, and the manual reference models were used where available.

Experiment 1 indicates that the changes to the code model could be correctly propagated to the PCM repository model. The previously discussed incorrect updating of the VSUM code model interfered with the evaluation of the CPRs in experiment 2. Because of the correlation of the issues with the code model update and the mismatches in the PCM repository model, we suspect that the PCM repository model is actually correctly updated.

Because of the successful updating of the PCM repository model in experiment 1, we suggest that the prototypical implementation achieved **G1.2.1**. Future work is required to investigate this claim more thoroughly.

G1.2.2: CPRs to the IM Both experiments indicated that the instrumentation points are created and removed correctly in the instrumentation model. Regarding the correct activation of changes by the prototypical implementation a single AIP was not correctly activated in experiment 2. Because of the time constraints of this thesis we could not thoroughly investigate the cause for this missing activation. We suspect that this represents a missed corner case in the implementation of the CPRs, which could be addressed by future work.

Overall, the prototypical implementation correctly populates the instrumentation model with instrumentation points and activates them correctly, except for a single action instrumentation point. The instrumentation model can be used for the implementation of adaptive instrumentation by future work. We feel that the prototypical implementation achieved **1.2.2** except for a single technical issue.

G1.2: CPRs The coverage data of both experiments indicate that nearly all reactions were executed during the experiments.

The sub goal **G1.2.1** was achieved by the prototypical implementation, but requires more investigation. **G1.2.2** was nearly achieved, because of a minor technical issue. Therefore we think that the prototypical implementation achieved **G1.2** with limitations.

G2: Detection of SICK AppSpace Apps Both experiments showed that the prototypical implementation could detect SICK AppSpace apps correctly. Therefore the prototypical implementation achieved **G2**.

G1: Overall Approach Both experiments indicate that a significant amount of instrumentation points must not be instrumented by the approach because of the adaptive instrumentation enabled by the instrumentation model. Experiment 1 shows that 77.1% of PCM actions would not have been instrumented during the experiment. For experiment 2 this number is 73.51%

The approach was able to propagate the changes quickly to the VSUM. The median propagation time per commit during the experiments were 253ms and 1165.5ms respectively. An outlier was commit 11 of case study 2 which took around 21 seconds to propagate. This commit completely restructured the whole case study application.

Concluding the discussion of the experiment results, the prototypical implementation falls short of reaching the main goal **G1**. While the evaluation demonstrated that the CIPM approach is amendable to being adapted to another programming language, we revealed multiple minor technical difficulties with the prototypical implementation.

Future work is needed to address these issues and to reevaluated the overall approach.

In the following we will discuss issues we faced during the development of the prototypical implementation.

Matching of the Code Models The integration of the used Lua code model into the approach required the implementation of a new matching for the code models. This matching was implemented by reusing the hierarchical match engine from SPLevo [22]. We implemented an *equality helper* that is used to by the hierarchical match engine to compare single model elements. In the beginning we implemented as few matchings ourselves as possible and deferred to the *edition distance matcher* from EMF compare for other elements. This approach proved unreliable and caused issues with the change derivation. We therefore extended our equality helper, so it can match all elements of the code model.

Initially we compared *variable expressions*, which have a reference to the assignment of the variable by comparing the name of the referenced variable. This approach yielded invalid code models, when changes to the source code cause the referenced assignment to change position in the code model. The cause for this is that the comparison of the variable expression which references the moved assignment does not detect the changes is position of the referenced assignment. The reference of the variable expression is therefore not updated during the change derivation, which causes the reference to be *dangling* and the code model to be unusable. We worked around this issue by additionally comparing the scope and position of the elements referenced by variable expressions.

An previously described EMF compare uses the matching both during the calculation of the code model differences and during the merging process. In the latter case the matching is used during the calculation of the insertion index of elements that are inserted into a list. We often encountered matches that worked acceptable during the difference calculation of the models, but caused errors in the merging process. This causes out of order elements in the updated code model. The presented evaluation revealed that the final prototypical implementation still experiences some of these issues.

The difficulty of implementing a matching for the code model is further amplified by its integration into the CIPM approach. The changes derived using the matching are used in the CPRs to propagate the changes to the other models of the approach. Depending on the matching changed elements may be matched or they may not be. When elements are matched only the actual changes in an element are passed through to the CPRs. If a changed element can not be matched, it causes the deletion of the old element and the creation of a new element. Consequently, changes to the matching may require further changes to the CPRs, which increases the difficulty of the implementation of both the matching and the CPRs

Future work could investigate different approaches for the model matching.

Change Order processed by the CPRs The CPRs propagate changes in the same order as the change derivation process created the changes. EMF compare for example orders the changes so, e.g. a referenced code model element is created before its reference.

We will now illustrate issues with the change ordering using an example. An external function call is added to a component, which calls a function of another component that is added in the same propagation. The external call may be processed before the newly created function is processed. During the propagation of the added external call, an `ExternalCallAction` is created. The `ExternalCallAction` has an attribute to the `OperationSignature` of the service it is calling. This `OperationSignature` does not exist at this point in time, because the added function has not yet been created. Therefore, the `ExternalCallAction` is created but invalid and needs to be repaired later, when the signature is actually created. This requires some kind of linking data structure that so the external call can be completed, when the added function is finally processed.

Another example issue is the handling of *serve* calls in the CPRs. Serve calls register a function with the SICK AppEngine and our approach uses these calls to determine which functions can be called by other apps. When the code around a serve call is modified the serve calls may not be matched correctly even if it is unchanged. This causes the removal of the old serve call and the creation of a new serve call. These two operations have no strict order and consequently the creation of the new may be processed before the removal of the old serve call. Processing these two changes naively causes issues, as the creation of a new serve call does nothing (because the SEFF which is served by the call does already exist). The removal of the old serve call would then cause the removal of the corresponding SEFF which unintended. This situation can be correctly handled by additional checks before the removal of a SEFF.

These two examples demonstrate, that while it is possible to implement CPRs that result in the correct updating of the other models, the resulting CPRs are complicated, difficult to reason about and to implement. This interferes with the maintainability of the CPRs.

5.7. Threats to Validity

This section lists threats to the validity of the evaluation. Threats to the internal threats to validity are discussed in subsection 5.7.1. Further, external threats to validity are discussed in subsection 5.7.2.

5.7.1. Internal Threats to Validity

Internal threats to the validity of an experiment e.g. refers to whether the tested subject matter actually makes a difference to the acquired metrics. Experiments with high internal validity have sufficient evidence to support their claim [52].

Reference Repository Models Previous work could rely on the availability of reference models for the used case study applications, because these applications were designed as benchmarks for research into component based software.

No such reference models were available for the case study applications presented in this thesis. Because of the time constraints of this thesis, the author of this thesis manually created and validated two PCM repository model instances for the evaluation of the model update process.

For the evaluation of the updating process of the PCM Repository models, we use an additional comparison with automatically created models in Table 5.5.1.1. The automatically created models are created by the same CPRs that update the evaluated models. This is certainly a threat to the internal validity of this particular evaluation. Creating and updating objects in the PCM repository model uses different logic, so depending on the propagated change not the same CPR logic is used in the process of the model propagation. We feel that this warrants the addition of comparison with automatically created PCM repository models to the evaluation.

Future work could include the creation of reference models for all commits of the case studies, ideally by a third party that was not part of the development process of this approach. Further, the created model could be validated by such third parties.

Same Matching in Approach and Evaluation Because of our new Lua code model, no matching implementation was available to match code model instances. We therefore also implemented a new custom hierarchical matching for the new code model. This matching is used during the change derivation for the updating of the VSUM code model. Further, we have to use this matching during the evaluation of the code model updating process. In specific, the matching is used to compare the updated code model of the VSUM with the parsed code model.

The use of the same matching for the code model update and the evaluation of the former clearly presents a threat to the internal validity of the evaluation. The matching is an integral part of the wider approach. Errors in the matching would likely have caused significant errors in other parts of the evaluation, in particular the comparison with the manually created PCM repository model. We therefore feel that the risk for a faulty matching is low, and consequently that the use of the matching during the evaluation of the code model update does not invalidate the results of said evaluation.

Low Complexity of Case Study 1 No software from the AppSpace ecosystem was immediately usable to for the evaluation of the approach. This is caused by the fact that the applications mostly consist of a single AppSpace app that implements a specific function. In specific, no applications using multiple apps in unison were available for the evaluation of this thesis, which is be required for a substantial evaluation of an approach for component based software.

Instead a relatively minimal case study application was implemented by the author of this thesis. The case study 1 applications consists of 4 AppSpace apps and therefore contains 4 components. We implemented the case study application to demonstrate all parts of the approach, and therefore also include e.g. the removal of a component from the application.

This threat to validity could be mitigated if a third party provided an application with multiple apps for evaluation in future work.

Case Study 2 Application As described above no applications with multiple apps were directly available for the evaluation of the approach. For case study 2 we resorted to evaluating an application consisting of an application consisting of a single SICK AppSpace app. External calls are calls between the components of an application and are the only action that is classified by the approach as architecturally relevant. As previously described, the control flow is only reconstructed for such architecturally relevant calls.

Having no architecturally relevant calls, results in the reconstruction of trivial SEFFs. To remedy this we map calls to functions which are served (and therefore have a SEFF) as external calls, even though the called function resides in the same component. By doing this we emulate the behaviour of the split application we originally intended to create.

5.7.2. External Threats to Validity

External validity of an experiment characterises to what degree its results can be generalised [52]. Consequently, threats to external validity are limitations or constraints of the evaluation.

Narrow Scope Only applications from the SICK AppSpace ecosystem are used for the evaluation of the approach.

We use the notion of an app as components for the reverse engineered component based architecture. This approach may not be generalisable to other component based Lua applications.

Further, we use the notion of *served* functions from the SICK AppSpace ecosystem to determine which functions can be called by other components. Other Lua applications may use a looser definition of exported functions which therefore may not be amendable to our approach.

This threatens the generalisability of the evaluation results to other Lua applications. Future work could investigate other Lua applications to mitigate this threat.

6. Related Work

This section will present related work to our approach.

6.1. Lua Analysis in Rascal

Klint et al. presented the Lua Analysis in Rascal (AiR) framework the static analysis of Lua programs that are used in the context of game engines [23]. Within the context of the their presented work, Lua scripts are used to customize the behaviour of a game without changing the game engine itself. The Lua code is embedded into the game engine and interacts with it through some sort *glue code*, which usually implemented in C/C++. While Lua allows for quick prototype creation, they identify two issues with the scalability of this approach: Once larger quantities of Lua scripts have accumulated, the maintenance of the overall system is hindered by a lack of tooling for the static analysis of the Lua scripts. Further, the glue code between the Lua scripts and the game engine and its libraries becomes an issue. The glue code may be generated by a standard generator, but this approach is prone to producing suboptimal bindings, which is not acceptable in this context. Another option is the manual implementation of the glue code which further strains the maintainability of the overall system. The authors approach is to enable the static analysis of Lua applications in their embedded context. This means that the glue code or bindings to the other libraries of the system are also considered during the static analysis of the system.

They implement Lua AiR as a Rascal *meta-program* [23, 24]. Rascal is a meta-programming language:

> Meta-programs are programs that analyze, transform or generate other programs. Ordinary programs work on data; meta-programs work on programs. [24]

Rascal programs follow the Extract Analyse Synthesize (EASY) paradigm: Interesting artifacts arise extracted, analysed and finally results are synthesized. Rascal uses its own grammar to textually define a meta-program. A parser can then be generated using this grammar, which can parse the target language into a *parse tree* a custom representation of the source code.

In the context of Lua AiR the Lua scripts interact with a game engine through generated glue code [23]. The structure and functionality of the glue code is modelled using two DSLs: The Interface Definition Language (IDL) defines which interfaces exist in the glue code for the Lua scripts. In addition the signature of the functions of the interface are defined, including parameter types. The mapping from game engine functionality to the interfaces modelled using the IDL is defined using the Interface Generator Language (IGL).

The analysis of a system using Lua AiR works as follows [23]. First the Lua code is parsed using the parser generated by Rascal. The parse tree obtained from the Rascal parsed is then *imploded* by matching its nodes against *algebraical data types* which then comprise the Abstract Syntax Tree (AST) used by the approach. The AST is then reduced to simplify the further analysis. A checker statically checks the types and annotates AST with scope information. A Control Flow Graph (CFG) is then generated and used by another analyzer to perform a *reaching definition* check.

Category	**Lua AiR**	**CIPM for Lua**
Parser	Rascal grammar (130 LOC)	Xtext grammar (370 LOC)
Model	Rascal Parse Tree, AST	Ecore-based code model
Application Context	Game Engine	SICK AppEngine, Sensor Hardware
Interface Definition	IDL	XML Manifest, Profile, Serve-calls
Typing	Types of IDL used for type checking	Not modelled
Goal	Static Analysis	Architecture and Performance model

Table 6.1.: Comparison of LuaAiR to the presented approach.

We compare Lua AiR to our approach it Table 6.1. While we use an Xtext-based grammar for our code model, LuaAiR uses a Rascal grammar. It is noteworthy that the grammar used by Lua AiR was implemented in less than half the LOC of our Xtext grammar. While Lua AiR parses the Lua source code into an Rascal parse tree and the converts this parse tree to an AST, we parse directly parse Lua source code into an Ecore-based model. This makes our approach more versatile, because Ecore-based models are automatically supported by tooling of the popular EMF. Applications of our approach are executed by the SICK AppEngine on the sensor hardware or on SIMs. SICK AppSpace apps can interface with other apps, or with functionality implemented by the platform to facilitate e.g. the reading of sensor data. The functions exposed by one app to the others is defined using an XML-based manifest. The actual bindings are created within an app by calling a function of the AppEngine that registers an exported function. In Lua AiR, script are embedded into a game engine to customize the behaviour for the respective game product. The Lua scripts interface with generated bindings which are defined using the IDL language. Lua AiR uses the type information expressed in the IDL model to implement static type checking in the Lua scripts. The presented approach does currently not model any types of function arguments and return values. Because this information is also available in the app manifests a similar approach to the one used with Lua AiR is thinkable. Overall both approaches have the goal of aiding the development of software with good code quality. Lua AiR uses static analysis, so bugs like the usage of undefined variables can be detected early during the implementation, ideally through the use of IDE support. The presented adapted CIPM approach aims to increase the code quality by making on the one hand up-to-date architectural models of the software available automatically during its development, which we hope can help developers to engineer better structured software. On the other hand, by providing the developers with the means to get insights into the performance of software during its development we believe that the developer can easier diagnose the performance issues, which leads to better overall performance.

6.2. SiDiff

Schmidt and Gloetzner presented the SiDiff frame work [36, 46]. They identify the importance of complete tooling for MBSE. This includes finding the difference between two models, which is often referred to as state-based model comparison. Many different modelling languages exist which exacerbates the issue of available tooling for these languages. Further, Domain-Specific Language are created and used for a specific use case. Tools for the differencing and merging of models are created by implementing an matching algorithm for the specific models.

The authors present a general approach to simplify the implementation process of tooling for model differencing and merging [36]. The use a model agnostic *kernel* that can be adapted to a large number of models through *configuration*. The kernel supports different matching types: Both a Top-Down-Matching and Bottom-Up-Matching are supported. The former starts at the root of both elements and only compares elements which parent elements were matching. A Bottom-Up-Matching is used for models where the similarity of two elements depends on the similarity of their respective descendants. In addition to these two matching types elements can be matched based on three further matching disciplines: Identifier-based matching compares elements based on their identifier. Signature-based matching calculates a signature of an element based on its attributes and elements are considered matching when their signatures are identical. Finally, model elements can be compared based on their similarity, rather than on their identifiers or signature. This makes the approach usable for models generated from another source, which inherently cannot be compared using persistent identifiers. The kernel configuration is used to specify which attributes of elements are used for the similarity calculation of elements. In addition, SiDiff provides 20 algorithms than can be selected for the similarity calculation of elements. Further, the differences in the models can be filtered according to user preferences, for changes which may not be relevant in a specific use case. Model elements which are similar enough, depending on some threshold are said to be corresponding. Changed elements are identified by their lack of correspondences into the other model.

The SiDiff framework workflow initially annotates both models that are compared. Based on the annotations and the kernel configuration both models are matched, yielding element correspondences. The correspondences are processed by a difference engine, to create representations of the differences in the models.

The authors use a configuration based approach to be able to compare models of many meta models without the need for the implementation of a custom matching algorithm. Our presented approach uses a custom matching paired with a hierarchical match engine and EMF compare to find the differences in different versions of a code model.

6.3. Semantic Lifting

Kehrer et al. presented an approach to semantic lifting of differences between software models [20]. State-based model comparison works by matching the elements of two models using a matching algorithm, which yields correspondences between matched model elements. In a second step *differences* are generated by processing model elements which have no correspondences. Such differences are low-level representations of the changes to the model and as such they are nearly incomprehensible to the users according to the authors.

The presented approach aims to *semantically lift* these low level differences to the level of conceptual descriptions of the model modifications. They achieve this by transforming the low level changes into what they call *user* level changes. The idea is to partition the set of low-level changes into so called *semantic change sets*. A semantic change set contains the low level changes belonging to one user level change. An example for such a user level change is presented in the form of the *pull up attribute* editing function of modern IDEs. The semantic change sets are created using Henshin transformation rules [44]. Henshin is a model transformation framework for EMF-based models, which is based on graph transformation concepts. According to the authors the semantic change set recognition rules implemented using Henshin are highly complex but also schematic in nature. Therefore the can be automatically generated by the authors. They provide recognition rules for the detection and transformation of 41 edit operations. Given a set of low-level changes, they recognize all possible edit operations. Because the detected operations may be overlapping, the authors post process the recognized edit operations to ensure that a minimum number of edit operations that simultaneously cover all low-level changes is selected. The evaluation of the approach indicates that on average between 6 and 18 low level changes can be semantically lifted into a user level change, depending on the type of used model.

The authors semantically lift changes to make them more comprehensible to users of the models. Our presented approach uses a matching based on the hierarchical match engine from SPLevo and EMF compare to find the differences between two versions of a code model. The low level changes are used verbatim by the the Vitruvius framework CPRs to propagate the changes to other models.

6.4. A posteriori Operation Detection in Evolving Software Models

Langer et al. presented an approach for the a posteriori detection of change operations for software models [26]. Logging or recording based approaches exist for finding changes or operations which are made to a software model. The authors state that such approaches are limited, because the recording depends on the modelling layout and only changes supported by the model editor can be recorded. The authors approach is based on EMF and is consequently usable for all Ecore based models. A posteriori approaches like state based model comparison can be used to find the differences of a model before and after the executed changes. They differentiate changes to a model into two classes: Atomic operations are additions, removals, updates and moves of model elements. On the other hand composite operations are composed of a set of such atomic operations. Atomic changes can be derived using approaches for state base model comparison. According tho the authors relying solely on atomic changes does not scale. Therefore, they favour combining atomic changes into composite changes. The authors extend the usual state based model comparison with an additional phase for the detection composite operations. They reuse existing operation specifications as a basis for the creation of composite operations. The differences which were created in previous phases using a state base model comparison are matched against the pre- and post-conditions of the operation specifications. This matching process is iteratively repeated until a fixpoint is reached, in order to handle possibly overlapping composite operations.

The authors combine atomic model changes to composite changes by reusing existing operation specifications. Our presented approach uses the atomic changes verbatim to propagate the changes to other models.

7. Conclusion and Future Work

In this last chapter we conclude this thesis with a summary of the presented approach. We further outline possible future work regarding the approach.

7.1. Conclusion

The Continuous Integration of Performance Models (CIPM) approach makes calibrated performance and architecture models of an application automatically available during its development [31]. Three software models are used by the CIPM approach: A model of the applications source code; the Palladio Component Model (PCM), a hybrid architecural- and performance-model, and an instrumentation model [6]. Changes to the source code of an application are automatically propagated to these models by integrating the models into Vitrivius, a view-based model consistency framework [48]. Consistency Preservation Rules (CPRs) propagate changes from changed model the others, to keep the models consistent. The CIPM approach was previously prototypically implemented and evaluated for microservice-based web applications implemented in the Java programming language.

We presented an adapted CIPM approach for Lua-based sensor applications in this thesis. Lua source code can be parsed into a contributed Lua code model based on an Xtext grammar. The code model is integrated into the CIPM approach by implementing and adapting the required Vitruvius CPRs. In addition we provided a new approach to updating existing Service Effect Specifications (SEFFs) of the PCM, which model the behaviour and resource demand of the services of a component.

A prototype of the presented approach was implemented. We were unable to implement the adaptive instrumentation required to complete the presented approach, because of technical difficulties and the time constraints of this thesis. Consequently the obtained PCM model is not calibrated by the implementation. Because of the difficulty of type inference in Lua code, the prototypical implementation does not model Lua types.

The prototypical implementation was evaluated using two case study applications from the SICK AppSpace ecosystem, which uses the Lua programming language to implement applications running on sensor- and related hardware. The first case study was implemented by us as a minimal running example by combining existing example applications from the SICK AppSpace, while the second case study is a commercial SICK sensor application for the detection and sorting of objects using their color. The evaluation of the case studies showed that the prototypical implementation was able create correct code models for the case studies. Further, the evaluation revealed minor technical issues with the prototypical implementation. The code models that are updated by the prototypical implementation contain out-of-order elements in a small number of propagations, which makes them unusable for the approach. In addition, an instrumentation point was not correctly activated in a single case. Still, we demonstrated that the overall CIPM approach is amendable to the integration of support for other programming languages. The applicability of the CIPM approach was extended by integrating support for the Lua programming language.

7.2. Future Work

We did not complete the prototypical implementation of the presented approach because of the time constraints of this thesis. Future work could complete the prototypical implementation by implementing the adaptive instrumentation of Lua applications using the instrumentation model that is updated by the current implementation. Further, the monitoring and calibration which was previously implemented may require adapting to the prototypical implementation of the presented approach. The technical difficulties which were revealed by the evaluation can be addressed by future work.

We limited the scope of the prototypical implementation of the approach. The PCM repository model can model types for e.g. the arguments and return values of SEFFs. We do not model the types of the Lua functions, because the inferrence of Lua types is challenging. This difficulty is demonstrated by the lack of tooling for the static analysis of Lua code that supports type inference [23]. Lua uses syntactic sugar for e.g. the declaration of functions. The prototypical implementation has only partial support for the Lua syntactic sugar.

Future work could complete the partial implementation of the presented approach, evaluate the the approach as a whole, and address the limitations and minor technical issues of the prototypical implementation.

Bibliography

[1] *ANTLR 3*. `https://www.antlr3.org/`. Accessed: 2023-03-10.

[2] Martin Armbruster. "Commit-Based Continuous Integration of Performance Models". MA thesis. Karlsruhe Institute of Technology (KIT), 2021.

[3] 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.

[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: 3rd and 4th International Conferences, ENASE 2008/2009, Funchal, Madeira, Portugal, May 4-7, 2008/Milan, Italy, May 9-10, 2009. Revised Selected Papers 3*. Springer. 2010, pp. 206–219.

[5] Kent L. Beck et al. "Manifesto for Agile Software Development". In: 2013.

[6] 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`. URL: `http://dx.doi.org/10.1016/j.jss.2008.03.066`.

[7] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings Publishing Company, 1991. ISBN: 9780805300918. URL: `https://books.google.de/books?id=w5VQAAAAMAAJ`.

[8] Alan W. Brown. "Model driven architecture: Principles and practice". In: *Software and Systems Modeling* 3.4 (Dec. 2004), pp. 314–327. ISSN: 1619-1374. DOI: `10.1007/s10270-004-0061-2`. URL: `https://doi.org/10.1007/s10270-004-0061-2`.

[9] Noureddine Dahmane. "Adaptive Monitoring for Continuous Performance Model Integration". MA thesis. Karlsruhe Institute of Technology (KIT), 2019.

[10] Leon Derczynski. "Complementarity, F-score, and NLP Evaluation". In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. Portorož, Slovenia: European Language Resources Association (ELRA), May 2016, pp. 261–266. URL: `https://aclanthology.org/L16-1040`.

[11] *Eclipse IDE*. `https://www.eclipse.org/ide/`. Accessed: 2023-04-05.

[12] *Eclipse Modeling Framework (EMF)*. `https://www.eclipse.org/modeling/emf/`. Accessed: 2022-07-19.

[13] *EMF Compare Webpage (Archive)*. `https://web.archive.org/web/20210415020438/https://www.eclipse.org/emf/compare/`. Accessed: 2021-04-15.

[14] *Git Version Control System*. `https://git-scm.com/`. Accessed: 2022-07-11.

[15] Object Management Group. *OMG - Standards Development Organization*. `https://omg.org`. Accessed: 2023-04-06.

[16] Elliotte Rusty Harold and W Scott Means. *XML in a nutshell: a desktop quick reference.* " O'Reilly Media, Inc.", 2004.

[17] Florian Heidenreich et al. "Closing the gap between modelling and java". In: *International Conference on Software Language Engineering*. Springer. 2009, pp. 374–383.

[18] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[19] Paul Jaccard. "The distribution of the flora in the alpine zone. 1". In: *New phytologist* 11.2 (1912), pp. 37–50.

[20] Timo Kehrer, Udo Kelter, and Gabriele 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`.

[21] Heiko Klare et al. "Enabling consistency in view-based system development - The Vitruvius approach". In: *The journal of systems and software* 171 (2021), Article no: 110815. ISSN: 0164-1212. DOI: `10.1016/j.jss.2020.110815`.

[22] Benjamin Klatt. *Consolidation of customized product copies into software product lines.* Vol. 16. KIT Scientific Publishing, 2016.

[23] Paul Klint, Loren Roosendaal, and Riemer van Rozen. "Game Developers Need Lua AiR". In: *Entertainment Computing - ICEC 2012*. Ed. by Marc Herrlich, Rainer Malaka, and Maic Masuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 530–535. ISBN: 978-3-642-33542-6.

[24] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "EASY Meta-programming with Rascal". In: *Generative and Transformational Techniques in Software Engineering III: International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Ed. by João M. Fernandes et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–289. ISBN: 978-3-642-18023-1. DOI: `10.1007/978-3-642-18023-1_6`. URL: `https://doi.org/10.1007/978-3-642-18023-1_6`.

[25] Klaus Krogmann. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis.* Vol. 4. KIT Scientific Publishing, 2014.

[26] Philip Langer et al. "A posteriori operation detection in evolving software models". In: *Journal of Systems and Software* 86.2 (2013), pp. 551–566. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2012.09.037`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121212002762`.

[27] Michael Langhammer. *Automated Coevolution of Source Code and Software Architecture Models.* Karlsruhe: KIT Scientific Publishing, Aug. 2019, p. 376. ISBN: 978-3-7315-0783-3. DOI: `10.5445/KSP/1000081447`.

[28] Michael Langhammer and Klaus Krogmann. “A co-evolution approach for source code and component-based architecture models”. In: *17. Workshop Software-Reengineering und-Evolution*. Vol. 4. 2015.

[29] *Language Server Protocol Specification*. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/. Accessed: 2023-03-25.

[30] *lua-language-server*. https://github.com/LuaLS/lua-language-server. Accessed: 2023-03-25.

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

[32] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification*. Version 2.4.1. 2013.

[33] David Monschein et al. “Enabling Consistency between Software Artefacts for Software Adaption and Evolution”. In: *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. 2021, pp. 1–12. DOI: 10.1109/ICSA51549.2021.00009.

[34] T. J. Parr and R. W. Quong. “ANTLR: A predicated-LL(k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. DOI: https://doi.org/10.1002/spe.4380250705. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250705.

[35] *PRICoBE*. https://sdqweb.ipd.kit.edu/wiki/PRICoBE. Accessed: 2022-06-23.

[36] Maik Schmidt and Tilman Gloetzner. “Constructing Difference Tools for Models Using the SiDiff Framework”. In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion ’08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 947–948. ISBN: 9781605580791. DOI: 10.1145/1370175.1370201. URL: https://doi.org/10.1145/1370175.1370201.

[37] *SICK AppEngine*. https://www.sick.com/de/en/sick-appspace/sick-appspace-software-tools/sick-appengine/c/g547567. Accessed: 2023-04-14.

[38] *SICK AppPool: SensorApps*. https://apppool.cloud.sick.com/publications. Accessed: 2022-07-06.

[39] *SICK AppSpace*. https://www.sick.com/de/de/sick-appspace/c/g555725?q=:Def_Type:ProductFamily. Accessed: 2023-04-14.

[40] *SICK AppSpace samples*. https://gitlab.com/sick-appspace/samples. Accessed: 2022-07-06.

[41] *SICK Sensor Integration Machine*. https://www.sick.com/de/de/integrationsprodukte/sensor-integration-machine/c/g386451. Accessed: 2023-04-14.

[42] Herbert Stachowiak. *Allgemeine Modelltheorie.* 1973. URL: http://www.amazon.de/Allgemeine-Modelltheorie-Herbert-Stachowiak/dp/3211811060/ref=sr_1_2/028-1073608-4157317?ie=UTF8&s=books&qid=1190302420&sr=1-2.

[43] Thomas Stahl. "Modellgetriebene Softwareentwicklung: Techniken, Engeneering, Management. 2., aktual. und erw". In: *Aufl. Heidelberg: Dpunkt. verlag* (2007), pp. 978–3898644488.

[44] Daniel Strüber et al. "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Graph Transformation.* Ed. by Juan de Lara and Detlef Plump. Cham: Springer International Publishing, 2017, pp. 196–208. ISBN: 978-3-319-61470-0.

[45] *The Melange Language Workbench.* http://melange.inria.fr/. Accessed: 2022-07-11.

[46] *The SiDiff Project (Archive).* https://web.archive.org/web/20210727062039/https://pi.informatik.uni-siegen.de/Projekte/sidiff/overview.php. Accessed: 2021-07-27.

[47] Rini Van Solingen et al. "Goal question metric (gqm) approach". In: *Encyclopedia of software engineering* (2002).

[48] *Vitruv - Git Repository.* https://github.com/vitruv-tools/Vitruv. Accessed: 2022-07-28.

[49] Joakim Von Kistowski et al. "Teastore: A micro-service reference application for benchmarking, modeling and resource management research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* IEEE. 2018, pp. 223–236.

[50] *Xtext - Language Engineering for Everyone!* https://www.eclipse.org/Xtext/index.html. Accessed: 2022-07-11.

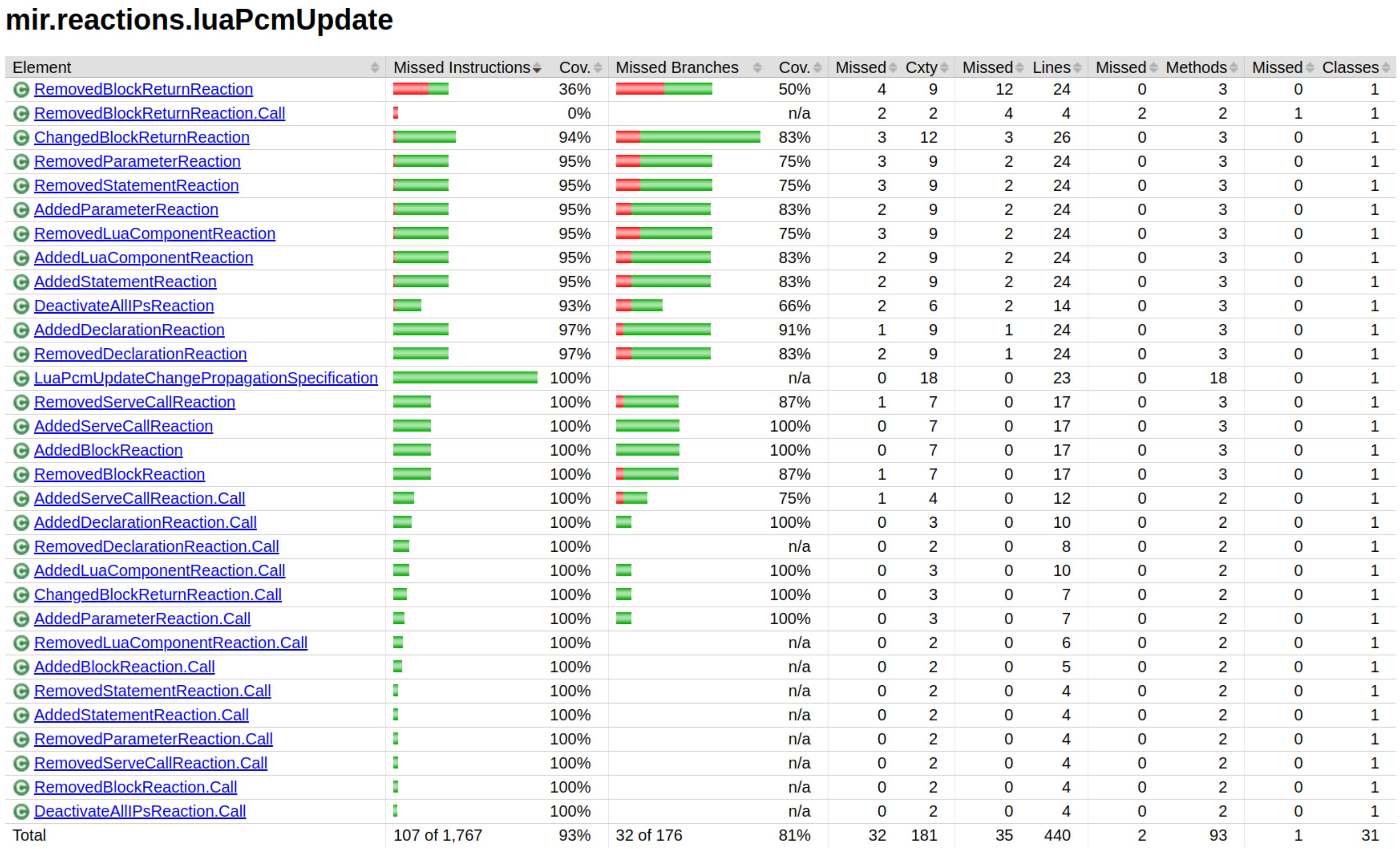
[51] *Xtext: Integration with EMF.* https://www.eclipse.org/Xtext/documentation/308_emf_integration.html. Accessed: 2022-07-19.

[52] Chong-ho Yu and Barbara Ohlund. *Threats to validity of research design.* https://creative-wisdom.com/teaching/WBI/threat.shtml. Accessed: 2023-03-31. 2010.

A. Appendix

A.1. Reaction Coverage Data

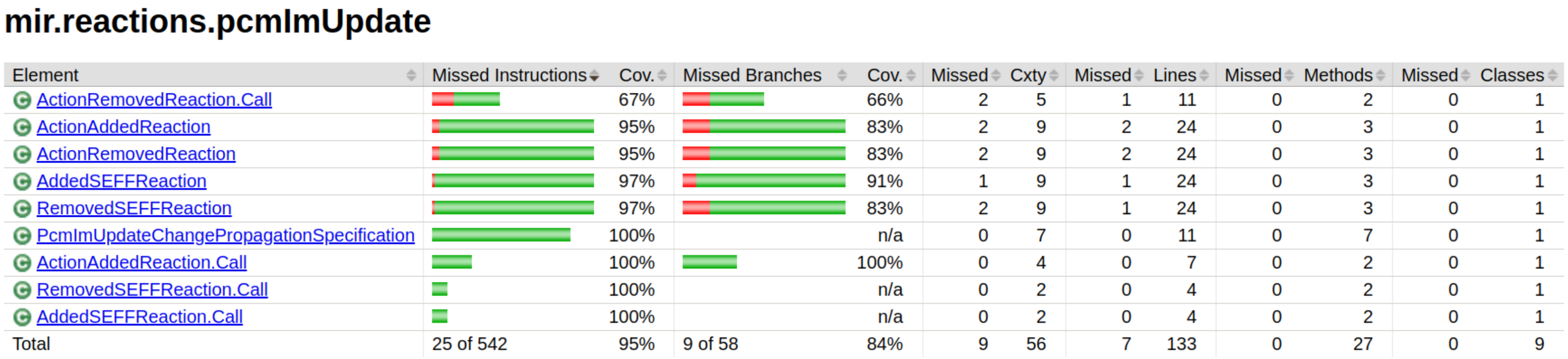
The coverage data of the reactions from the exececution of Experiment 1 can be seen in Figure A.1 and Figure A.2.



mir.reactions.luaPcmUpdate

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RemovedBlockReturnReaction		36%		50%	4	9	12	24	0	3	0	1
RemovedBlockReturnReaction.Call		0%		n/a	2	2	4	4	2	2	1	1
ChangedBlockReturnReaction		94%		83%	3	12	3	26	0	3	0	1
RemovedParameterReaction		95%		75%	3	9	2	24	0	3	0	1
RemovedStatementReaction		95%		75%	3	9	2	24	0	3	0	1
AddedParameterReaction		95%		83%	2	9	2	24	0	3	0	1
RemovedLuaComponentReaction		95%		75%	3	9	2	24	0	3	0	1
AddedLuaComponentReaction		95%		83%	2	9	2	24	0	3	0	1
AddedStatementReaction		95%		83%	2	9	2	24	0	3	0	1
DeactivateAllIPsReaction		93%		66%	2	6	2	14	0	3	0	1
AddedDeclarationReaction		97%		91%	1	9	1	24	0	3	0	1
RemovedDeclarationReaction		97%		83%	2	9	1	24	0	3	0	1
LuaPcmUpdateChangePropagationSpecification		100%		n/a	0	18	0	23	0	18	0	1
RemovedServeCallReaction		100%		87%	1	7	0	17	0	3	0	1
AddedServeCallReaction		100%		100%	0	7	0	17	0	3	0	1
AddedBlockReaction		100%		100%	0	7	0	17	0	3	0	1
RemovedBlockReaction		100%		87%	1	7	0	17	0	3	0	1
AddedServeCallReaction.Call		100%		75%	1	4	0	12	0	2	0	1
AddedDeclarationReaction.Call		100%		100%	0	3	0	10	0	2	0	1
RemovedDeclarationReaction.Call		100%		n/a	0	2	0	8	0	2	0	1
AddedLuaComponentReaction.Call		100%		100%	0	3	0	10	0	2	0	1
ChangedBlockReturnReaction.Call		100%		100%	0	3	0	7	0	2	0	1
AddedParameterReaction.Call		100%		100%	0	3	0	7	0	2	0	1
RemovedLuaComponentReaction.Call		100%		n/a	0	2	0	6	0	2	0	1
AddedBlockReaction.Call		100%		n/a	0	2	0	5	0	2	0	1
RemovedStatementReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
AddedStatementReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedParameterReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedServeCallReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedBlockReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
DeactivateAllIPsReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
Total	107 of 1,767	93%	32 of 176	81%	32	181	35	440	2	93	1	31

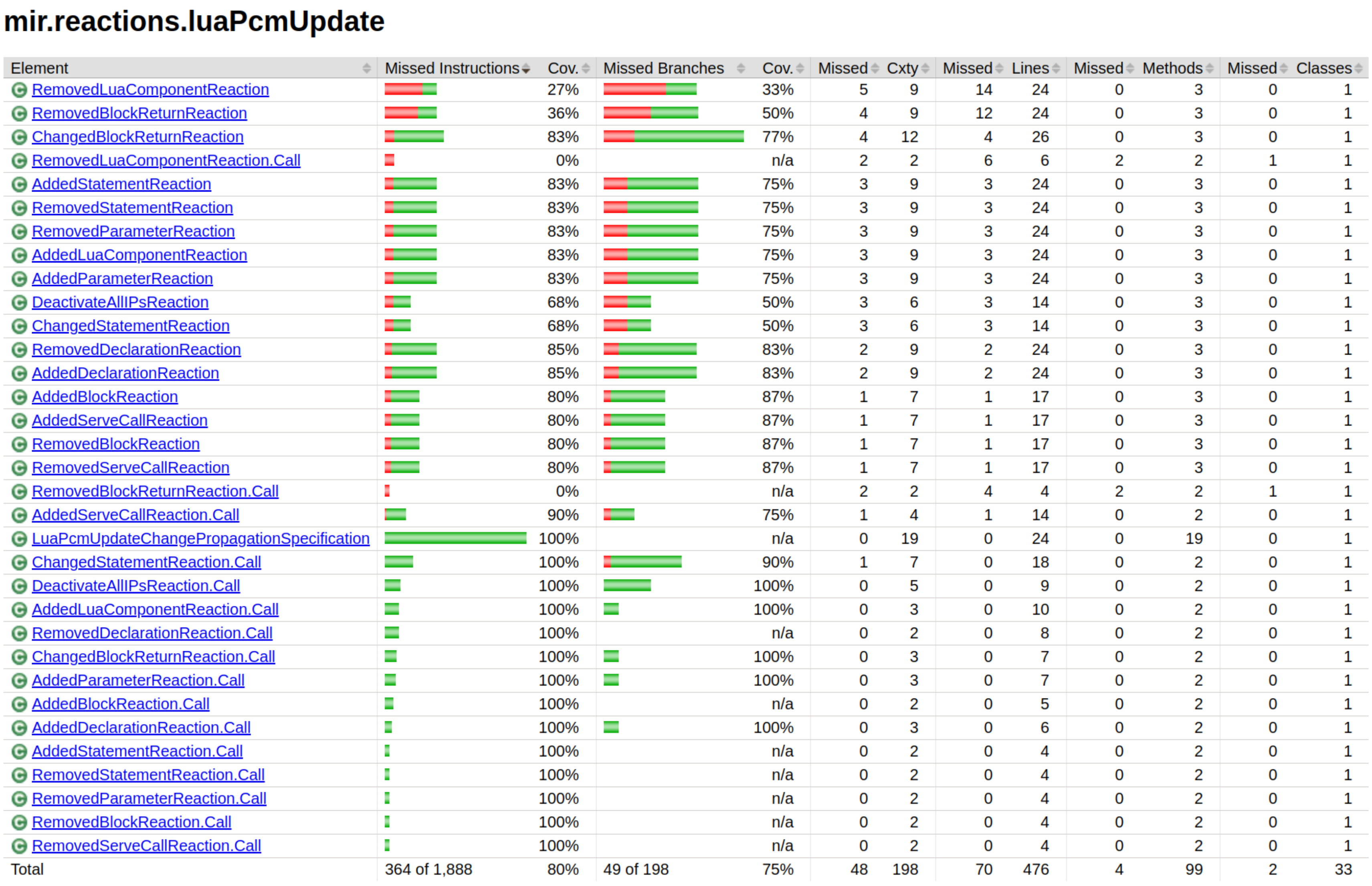
Figure A.1.: Coverage data for the reactions from the Lua code model during the execution of experiment 1.



mir.reactions.pcmImUpdate

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ActionRemovedReaction.Call	🟥🟩	67%	🟥🟩	66%	2	5	1	11	0	2	0	1
ActionAddedReaction	🟥🟩	95%	🟥🟩	83%	2	9	2	24	0	3	0	1
ActionRemovedReaction	🟥🟩	95%	🟥🟩	83%	2	9	2	24	0	3	0	1
AddedSEFFReaction	🟩	97%	🟥🟩	91%	1	9	1	24	0	3	0	1
RemovedSEFFReaction	🟩	97%	🟥🟩	83%	2	9	1	24	0	3	0	1
PcmImUpdateChangePropagationSpecification	🟩	100%		n/a	0	7	0	11	0	7	0	1
ActionAddedReaction.Call	🟩	100%	🟩	100%	0	4	0	7	0	2	0	1
RemovedSEFFReaction.Call	🟩	100%		n/a	0	2	0	4	0	2	0	1
AddedSEFFReaction.Call	🟩	100%		n/a	0	2	0	4	0	2	0	1
Total	25 of 542	95%	9 of 58	84%	9	56	7	133	0	27	0	9

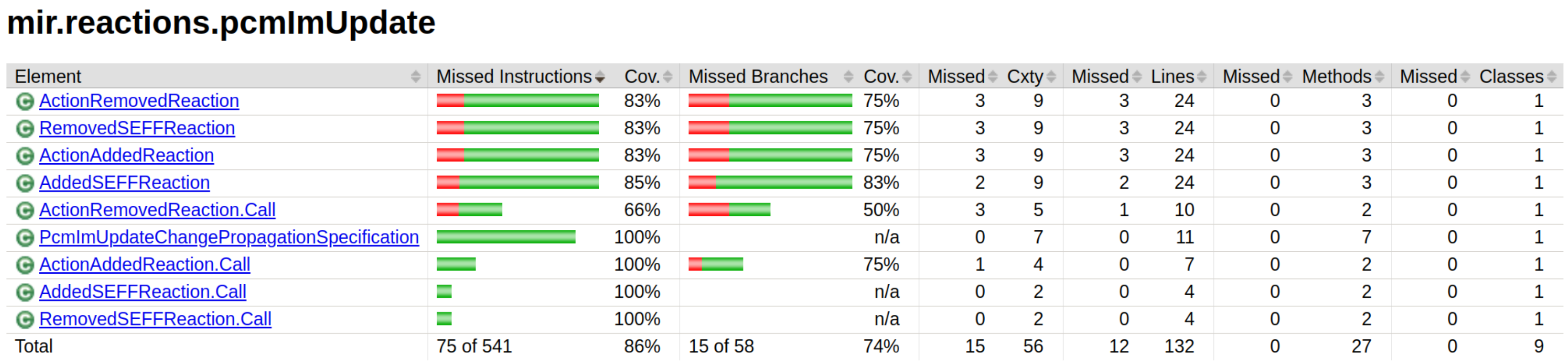
Figure A.2.: Coverage data for the reactions from the repository model of the PCM to the instrumentation model during the execution of experiment 1.



mir.reactions.luaPcmUpdate

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
RemovedLuaComponentReaction		27%		33%	5	9	14	24	0	3	0	1
RemovedBlockReturnReaction		36%		50%	4	9	12	24	0	3	0	1
ChangedBlockReturnReaction		83%		77%	4	12	4	26	0	3	0	1
RemovedLuaComponentReaction.Call		0%		n/a	2	2	6	6	2	2	1	1
AddedStatementReaction		83%		75%	3	9	3	24	0	3	0	1
RemovedStatementReaction		83%		75%	3	9	3	24	0	3	0	1
RemovedParameterReaction		83%		75%	3	9	3	24	0	3	0	1
AddedLuaComponentReaction		83%		75%	3	9	3	24	0	3	0	1
AddedParameterReaction		83%		75%	3	9	3	24	0	3	0	1
DeactivateAllIPsReaction		68%		50%	3	6	3	14	0	3	0	1
ChangedStatementReaction		68%		50%	3	6	3	14	0	3	0	1
RemovedDeclarationReaction		85%		83%	2	9	2	24	0	3	0	1
AddedDeclarationReaction		85%		83%	2	9	2	24	0	3	0	1
AddedBlockReaction		80%		87%	1	7	1	17	0	3	0	1
AddedServeCallReaction		80%		87%	1	7	1	17	0	3	0	1
RemovedBlockReaction		80%		87%	1	7	1	17	0	3	0	1
RemovedServeCallReaction		80%		87%	1	7	1	17	0	3	0	1
RemovedBlockReturnReaction.Call		0%		n/a	2	2	4	4	2	2	1	1
AddedServeCallReaction.Call		90%		75%	1	4	1	14	0	2	0	1
LuaPcmUpdateChangePropagationSpecification		100%		n/a	0	19	0	24	0	19	0	1
ChangedStatementReaction.Call		100%		90%	1	7	0	18	0	2	0	1
DeactivateAllIPsReaction.Call		100%		100%	0	5	0	9	0	2	0	1
AddedLuaComponentReaction.Call		100%		100%	0	3	0	10	0	2	0	1
RemovedDeclarationReaction.Call		100%		n/a	0	2	0	8	0	2	0	1
ChangedBlockReturnReaction.Call		100%		100%	0	3	0	7	0	2	0	1
AddedParameterReaction.Call		100%		100%	0	3	0	7	0	2	0	1
AddedBlockReaction.Call		100%		n/a	0	2	0	5	0	2	0	1
AddedDeclarationReaction.Call		100%		100%	0	3	0	6	0	2	0	1
AddedStatementReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedStatementReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedParameterReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedBlockReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
RemovedServeCallReaction.Call		100%		n/a	0	2	0	4	0	2	0	1
Total	364 of 1,888	80%	49 of 198	75%	48	198	70	476	4	99	2	33

Figure A.3.: Coverage data for the reactions from the Lua code model during the execution of experiment 2.



mir.reactions.pcmImUpdate

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ActionRemovedReaction	🟥🟩	83%	🟥🟩	75%	3	9	3	24	0	3	0	1
RemovedSEFFReaction	🟥🟩	83%	🟥🟩	75%	3	9	3	24	0	3	0	1
ActionAddedReaction	🟥🟩	83%	🟥🟩	75%	3	9	3	24	0	3	0	1
AddedSEFFReaction	🟥🟩	85%	🟥🟩	83%	2	9	2	24	0	3	0	1
ActionRemovedReaction.Call	🟥🟩	66%	🟥🟩	50%	3	5	1	10	0	2	0	1
PcmImUpdateChangePropagationSpecification	🟩	100%		n/a	0	7	0	11	0	7	0	1
ActionAddedReaction.Call	🟩	100%	🟥🟩	75%	1	4	0	7	0	2	0	1
AddedSEFFReaction.Call	🟩	100%		n/a	0	2	0	4	0	2	0	1
RemovedSEFFReaction.Call	🟩	100%		n/a	0	2	0	4	0	2	0	1
Total	75 of 541	86%	15 of 58	74%	15	56	12	132	0	27	0	9

Figure A.4.: Coverage data for the reactions from the repository model of the PCM to the instrumentation model during the execution of experiment 2.