Karlsruhe Institute of Technology

Continuous Integration of Performance Models for Lua-Based IIoT Applications

Master Thesis Proposal 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 Reussner
Advisor:	M. Sc. Manar Mazkatli
Second advisor:	M. Sc. Martin Armbruster

8. August 2022 – 8. February 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

The concept of industrial internet of things (IIoT) is steadily adopted by the manufacturing industry. Sensors, actuators and consoles for users are connected to the network. Consequently, applications with the ability to react based on all this available data can be implemented. The computing hardware used in the context is usually heterogeneous and may only have limited capabilities. It is difficult to determine if an application shows acceptable performance metrics, such as reaction time and throughput. Further, deciding which nodes execute which processing tasks can present a significant challenge. Developers may rely on trial-and-error exploration of the design space which is slow and prone to errors. The development of IIoT applications can therefore benefit significantly from a systematic approach to performance management. An example for systematic performance management is the use of performance modeling. Performance models are often created manually which makes them cumbersome to use. Additionally, models are quickly outdated, because of changes to the modeled software.

The Continuous Integration of Performance Models (CIPM) approach can be employed to continuously update models of software during development. A prototypical implementation of the CIPM approach leverages the Palladio Component Model (PCM), the Vitruvius framework, the coevolution approach, and adaptive instrumentation [2, 11, 12, 13]. It targets microservice-based web applications implemented in the Java programming language.

This proposal presents an approach to adapting the CIPM approach so it is applicable to Lua-based IIoT applications. A Code Model (CM) for Lua source code is generated by parsing the code and processing the resulting Abstract Syntax Tree (AST). The CM is integrated into the prototypical implementation by adapting existing Consistency Preservation Rules (CPRs) within the Vitruvius framework. Further, support for the instrumentation of Lua source code is added to the CIPM approach. A plan for the evaluation of the adapted prototypical implementation is presented. Real-world Lua IIoT applications from the SICK AppPool are used for the evaluation of the adapted prototypical implementation.

Contents

List of Figures

List of Tables

1. Introduction

With the popularisation of the Internet of Things (IoT) concept and industry 4.0 the manufacturing industry has begun a transition to more integrated and *smarter* production. Networked sensor nodes can continuously monitor the production and autonomous actuators react accordingly. Distributed applications are needed to realise the potential benefits of an integrated production. These industrial internet of things (IIoT) applications are comparable to traditional distributed applications found in the internet, but they are constraint by their context. Examples for these constraints are heterogeneous hardware with varying capabilities in terms of performance. Because the applications need to act on available live-data, their operation is latency sensitive. The development of IIoT applications can therefore 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 more agile software developed. AbPP can aid with considerations like

- 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?

The Continuous Integration of Performance Models (CIPM) approach can be used to make up-to-date performance models for AbPP available during the development cycle of applications [14, 13]. The approach is prototypically implemented for microservice-based web applications implemented in the Java programming language. Changes to the applications source code cause automatic updating of corresponding elements in the performance model. The Palladio Component Model (PCM) serves as a architecture-level performance model of the application. Performance model parameters (PMPs) are needed to simulate the performance of the developed software. These PMPs are obtained by instrumenting changed parts of the source code. The Vitruvius framework is used to keep the source code and the PCM consistent.

This proposal presents an approach to adapting the CIPM approach and its prototypical implementation for use with Lua-based IIoT applications. The adapted prototypical implementation is evaluated using real-world applications from the SICK AppSpace ecosystem.

This paragraph describes the structure of the proposal. The foundations are laid out in chapter 2. The approach is presented in chapter 3 and structured according to the PRICoBE principle [18]. Organizational matters like the thesis' schedule are discusses in chapter 4. Related work is presented in chapter 5.

2. Foundations

This chapter outlines the foundations of the presented approach.

The popularity of agile software development is the base motivation behind the presented approach. Agile software development is introduced in section 2.1.

2.1. Agile Software Development

Agile software development was popularized in the 2001 *Agile Manifesto* [1]. The manifestos principles aim for software development with shorter release cycles even down to a couple of weeks [1]. The principles further include “[Welcoming] changing requirements”[1]. 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 [4]. 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 [9]. 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. Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) project provides facilities for modeling various kinds of data [7]. Central to the EMF project is the *Ecore meta model*, which defines the structure of the data models. Furthermore, the Ecore meta model describes itself. Ecore is an implementation of the Object Management Groups (OMGs)’s Essential Meta-Object Facility (EMOF) [15]. EMOF itself is based on another standard by the OMG: UML2 [21]. Because EMOF is based on UML2, EMOF models and consequently Ecore can be serialized using the XML Metadata Interchange (XMI) format [15]. This is useful to persist a model instance.

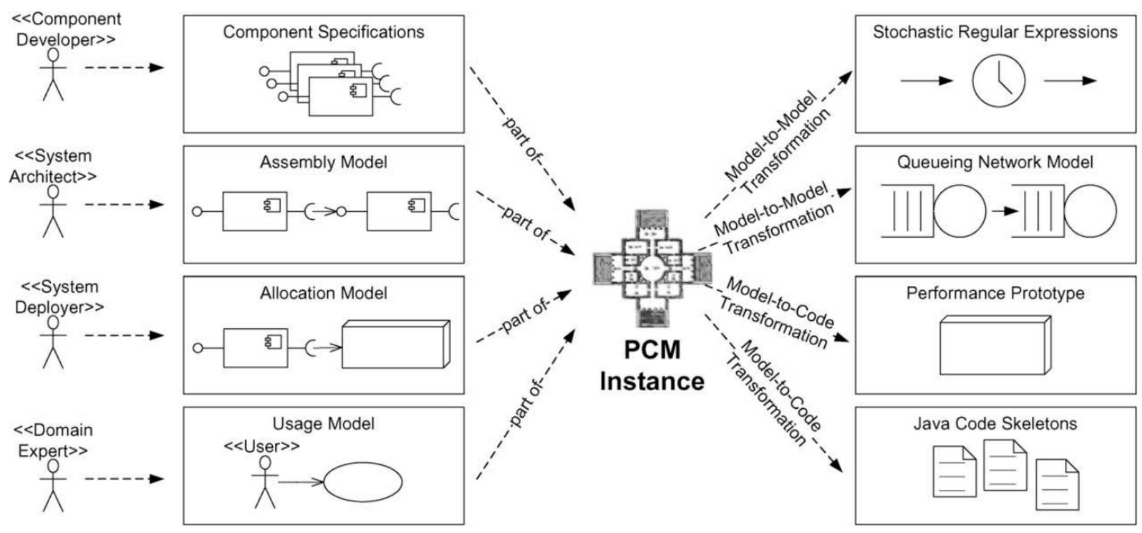


Figure 2.1.: Overview of the Palladio Component Model (PCM) process [2].

2.2.1. Xtext Framework

Xtext is a framework for the development of programming languages and Domain-Specific Languages (DSLs) [23]. Languages are defined using the so-called (Xtext) grammar language. The grammar itself is a DSL for the description of textual languages. Xtext is integrated into the Eclipse Integrated development environment (IDE). A language developed with Xtext has automatic support in Eclipse. This includes syntax highlighting, validation of code and automatic code generation. Xtext parses source code into an in-memory model. This model is sometimes also referred to as an Abstract Syntax Tree (AST) [24]. E.g. EMF models can be used as in-memory representations of parsed source code.

It is possible to infer an Ecore language model from a Xtext grammar.

2.3. Palladio Component Model

PCM is a meta model for the specification of component based software architectures [2]. The PCM was conceived as part of the Palladio approach. An overview over the complete 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.

Components, their interfaces and the actions for the interfaces are defined in the component specifications. An overview of the meta-model for the component description is displayed in Figure 2.2. The functionality exposed by the components Application Programmable Interface (API) are referred to as services. The services are specified using so-called Service Effect Specifications (SEFFs). SEFFs have a signature and can contain multiple actions. The actions can have resource demanding (RD) behaviour, e.g., an action could take 100 CPU cycles to process. External actions are actions that call a service of a different component. A pair of source code and the corresponding RDSEFF can be seen in

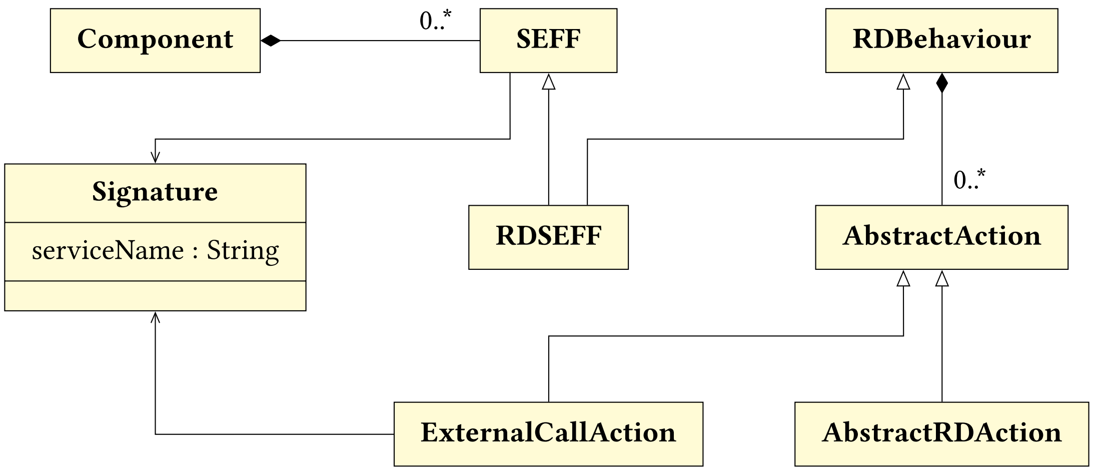


Figure 2.2.: Overview of the component specifications based on [2].
SEFF = Service Effect Specification; RD = ResourceDemanding

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

  requiredService1();

  // internal computation
  innerMethod();

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

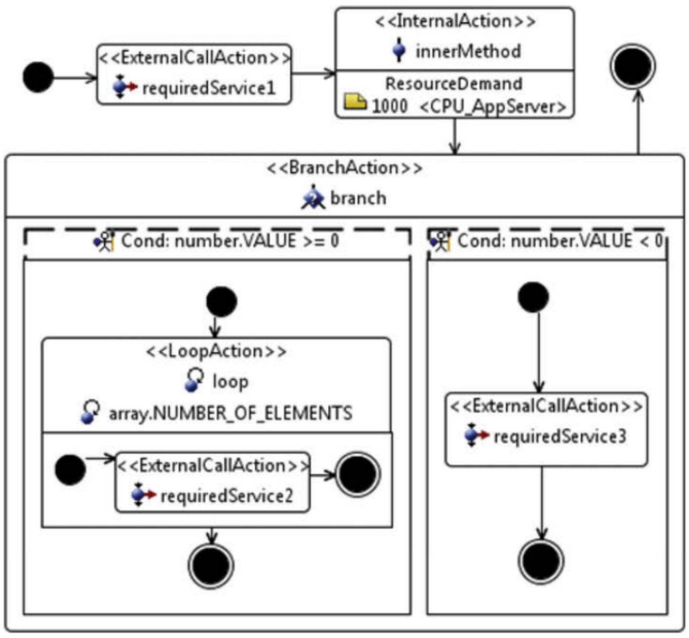


Figure 2.3.: A pair of source code and the corresponding Resource Demanding SEFF (RD-SEFF)

Figure 2.3. LoopActions and BranchActions are used to describe the control flow of the source code example.

Using model-to-code transformations it is possible to generate Java code skeletons for implementation.

The models can be used to analyse qualitative aspects of the software like performance measures. The most prominent performance measure in this context is response time.

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

Vitruvius is a framework for view based software development [11]. In the context of model-based software engineering (MBSE) software is usually described using multiple models, which describe a certain aspect of the software. It is challenge to make these various models consistent. The Vitruvius approach keeps multiple models consistent through the use of Consistency Preservation Rules (CPRs) (see subsection 2.4.1). The data of the underlying models is available through views which can aggregate the information spread to multiple models.

2.4.1. Consistency Preservation Rules

CPRs are used to keep the underlying models within Vitruvius consistent. The rules specify how consistency in retained, when models in Vitruvius are changed. CPRs are defined using two DSLs: the *mappings* language and the *reactions* language.

2.5. Continuous Integration of Performance Models

The Continuous Integration of Performance Models (CIPM) approach aims to make up-to-date architectural software models models, e.g., the PCM, available during the complete development and operation of software [13]. 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 faster updating and calibration of performance models:

- Incremental model updating inspired by and reusing parts of the Coevolution approach (see subsection 2.5.1)

- Adaptive instrumentation (see subsection 2.5.2)

An example of the performance model update workflow is presented in subsection 2.5.3. Details regarding the prototypical implementation of the CIPM approach can be found in subsection 2.5.4.

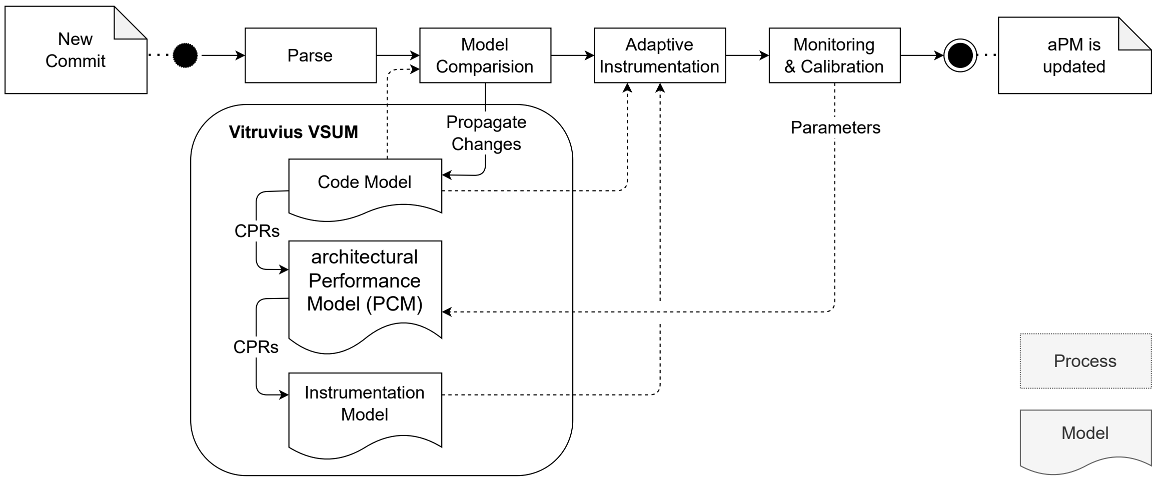


Figure 2.4.: An overview of the model update process of the CIPM approach [13]

2.5.1. The Coevolution Approach

Langhammer presented the Coevolution approach, which can be employed to automatically keep consistency between source code a PCM instance [12]. The Coevolution approach was prototypically implemented for Java and is embedded in the Vitruvius approach. JaMoPP is used to parse Java source code into an Ecore Code Model (CM) [10]. 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.

2.5.2. 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 statement can account for how often a routine is invoked.

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. Only changed parts of the software are instrumented, as up-to-date PMPs still exist for the untouched parts of the software. Further, instrumentation point can be deactivated once they were monitored for a sufficiently long time period.

2.5.3. Model Update Process

The process of updating an architectural performance model is depicted in Figure 2.4. Initially software developer makes some 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. The new CM is compared to the current CM, which resides in the Vitruvius Virtual Single Underlying Model (VSUM). The changes detected by the comparison are propagated to the current CM. The CPRs of the VSUM 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 an consequently need to be instrumented now. Using the updated CM and IM, the source code is adaptively instrumented (see again subsection 2.5.2). 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 [13]. The performance model is now calibrated and can be used for architecture-based performance prediction (AbPP) of the software.

2.5.4. Prototypical Implementation

A prototypical implementation of the CIPM approach exists for microservice-based software written in the Java programming language [13]. The prototypical implementation leverages parts of the implementation of the Coevolution approach which is described in subsection 2.5.1. The accuracy of the CIPM approach was investigated in the context of Java based microservices by previous work [16].

2.6. Industrial Internet of Things

The Internet of Things (IoT) refers to connecting various sensors, computers and possibly actuators to a common network, the internet and computing cloud. The emerging trend has found various applications including personal fitness trackers, home automation sensors and the restructuring of the manufacturing industry. In the latter case the term industrial internet of things (IIoT) is used. Networked sensors are often used to monitor production of goods or other infrastructure, like e.g. in the energy sector. They facilitate the automatic inspection for quality assurance means.

2.6.1. Differences between distributed applications and IIoT

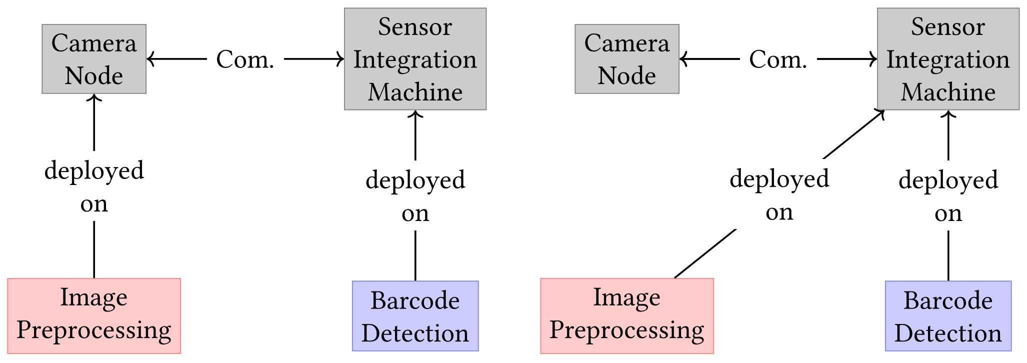
IIoT applications usually differ from traditional distributed applications in the following aspects:

Diverse hardware A networked sensor node may only have minimal hardware capabilities and use different compute architecture.

Latency sensitivity IIoT can be more latency sensitive, especially in the manufacturing industry. Sometimes they even enter he realm of real-time systems. In the latter case other network protocols like PROFIBUS may be needed.

Edge Computing Edge computing is the concept of relocating data processing or storage for to conserve network bandwidth and provide better latency.

For example bandwidth is conserved if measurements from a networked sensor node is pre-filtered by the node itself or a node in its topological proximity. On the other



(a) Distributed deployment (b) Deployment on a single node

Figure 2.5.: An application with two possible deployment configurations.
Images are taken by a camera connected to the camera node. The preprocessing can either be done directly on the camera node or on the more powerful SIM.

hand lower latency can be achieved if data is stored "closer" to the user. A common example are content delivery networks in the internet.

2.6.2. SICK AppSpace ecosystem

The SICK AppSpace is a commercial ecosystem of IIoT applications. SICK provides a wide array of sensor nodes like RFID and LASER scanners. Some sensors can be programmed using the Lua programming language applications, so-called SensorApps or *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 an *AppEngine*, which runs on the programmable sensors and SIMs. 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 name (CROWN), 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.

Complete SensorApps can be purchased through the SICK AppPool [19]. Depending on the requirements SensorApps are portable e.g. between different models of SIMs.

3. Approach

This chapter presents an overview of our approach. It is structured according to the PRICoBE principle (for Problems, Research Questions, Idea, Contributions, Benefits and Evaluation) [18]. 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 Continuous Integration of Performance Models (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 Code Model (CM) are language specific and can therefore only be used for the Java programming language.
- **P1.2** The Consistency Preservation Rules (CPRs) used in the CIPM approach are specific to the Java CM.
  - **P1.2.1** The Service Effect Specification (SEFF) reconstruction method used in the CIPM approach is designed for the Java CM.
- **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.

P3 Evaluating the extra-functional aspects of varying software component allocations of distributed industrial internet of things (IIoT) applications is difficult. An example can be seen in Figure 2.5.

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 that is not object-oriented like the Lua programming language?

- **RQ1.1** How can a CM that is usable in the CIPM approach be obtained for Lua IIoT applications?
- **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 for the software components of a Lua IIoT application?
- **RQ1.3** How can the method of code instrumentation be adapted to a different programming language?
  The following research question is of interest but will not be systematically investigated in the thesis: Is the structure of instrumentation model sufficient to support the instrumentation of a different programming language?

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

RQ3 How can the CIPM approach aid the evaluation of varying software component allocation schemes of Lua IIoT applications?

3.3. Idea

The idea of the presented approach is adapting the CIPM approach for use in the context of IIoT (sensor-)applications. The SICK AppSpace ecosystem (see subsection 2.6.2) will serve as an example to evaluate and demonstrate the process of adapting the CIPM approach. The AppSpace ecosystem uses distributed IIoT applications implemented in the Lua programming language. These applications can be distributed between programmable sensor nodes, Sensor Integration Machines (SIMs) and the cloud. In order to apply the CIPM approach to this different environment a significant amount of components of the prototypical implementation of the CIPM approach need to be adapted. Suitable parts of the existing approach should be reused and missing components must be implemented from scratch. The adapted approach will be evaluated using real-world and close-to-real-world example applications.

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 IIoT applications. The adapted prototypical implementation is evaluated.

- **C1.1** Parsing of Lua source code is integrated into the CIPM approach.
- **C1.2** The CPRs used in the prototypical implementation are adapted to the new Lua CM.
  - **C1.2.1** A SEFF reconstruction method is adapted for the new Lua CM.
- **C1.3** Support for instrumentation of Lua applications is added to the CIPM approach.

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

C3 The CIPM approach is adapted and evaluated with regard to evaluating varying software component allocations of Lua IIoT applications.

3.5. Benefits

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

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

- **B1.1** CMs for Lua IIoT applications can be generated. It is integrated into the CIPM approach.
- **B1.2** Changes to the Lua CM can be correctly propagated to the corresponding Palladio Component Model (PCM) instance.
  - **B1.2.1** SEFF can be reconstructed for software components of Lua IIoT applications.
- **B1.3** More programming languages can be instrumented for use in the context of the CIPM approach.

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

B3 Varying software component allocation schemes can be analysed using the CIPM approach.

3.6. Evaluation

This section discusses the planned evaluation of the approach presented in the previous sections of this chapter. The evaluation follows the Goal Question Metric (GQM) paradigm [22]: 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 subsection 3.6.1. The requirements for the case study are discussed in subsubsection 3.6.2.1. Based on the GQM plan and the requirements of the GQM plan, subjects for case studies are discussed in subsubsection 3.6.2.2.

3.6.1. GQM Plan

This section presents the GQM plan 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-based IIoT applications.

- **Q1-1** Does the prototypical implementation support Lua code?
  - **M1-1-1** Satisfaction of the sub-goals **G1.1**, **G1.2** and **G1.3**.
- **Q1-2** Whats the difference between the predicted response times of an up-to-date PCM instance and real-world monitoring data in the context of Lua IIoT applications?

  A similar evaluation was previously executed for microservice-based Java web applications [16]. The following metrics consequently reproduce the previously used metrics.
  - **M1-2-1** Difference of statistical measures (e.g. average, min. and max.)
  - **M1-2-2** Kolmogorov-Smirnov (KS) test
  - **M1-2-3** Wasserstein distance

G1.1 Lua source of IIoT applications can be parsed and a CM for use with the CIPM approach can be generated.

- **Q1.1-1** Can a CM of the Lua source code of IIoT applications be generated?
  - **M1.1-1-1** Equality of:
    - The Abstract Syntax Tree (AST) of the original source code.
    - The AST of the generated CM printed as source code.
  - **M1.1-1-2** Ratio of passing unit tests which were manually created for an instance of the CM.
- **Q1.1-2** Can architectural CM changes be propagated to the Vitruvius Virtual Single Underlying Model (VSUM)?
  - **M1.1-2-1** Ratio of correctly propagated changes to the VSUM.

G1.2 The existing CPRs from the CM to the PCM are adapted for the new CM.

- **Q1.2-1** Are architectural changes to the CM correctly propagated to the models of the PCM?

M1.2-1-1 Difference between the PCM instance before and after the change propagation.

M1.2-1-2 Difference between an automatically updated PCM instance and a manually updated PCM instance.

G1.2.1 SEFFs can be reconstructed using the new Lua CM.

Q1.2.1-1 Can the adapted prototypical implementation reconstruct SEFFs of a CM instance?

M1.2.1-1-1 Ratio of correctly reconstructed SEFFs

G1.3 Lua source code can be instrumented using an Instrumentation Model (IM) which is integrated into the Vitruvius VSUM.

Q1.3-1 Can the Lua source code be correctly instrumented?

M1.3-1-1 Ratio of correctly instrumented instrumentation points from the corresponding IM.

Q1.3-2 Does an IM for the instrumentation exist?

M1.3-2-1 Number of existing IM which are integrated into the Vitruvius VSUM.

G2 Components of IIoT applications can be discovered automatically.

Q2-1 Does the adapted prototypical implementation discover all components of a Lua IIoT application?

M2-1-1 $\frac{\text{\# discovered components}}{\text{\# existing components}}$

M2-1-2 $\frac{\text{\# correctly mapped components}}{\text{\# existing components}}$

G3 The approach can accurately simulate the performance of a Lua IIoT application given a set of predefined component allocations.

Q3-1 Are all component allocations simulated?

M3-1-1 Number of simulated component allocations

Q3-2 Is the simulation of the allocations accurate? What is the difference between the response time predicted by an up-to-date PCM instance and the actual response time?

M3-2-1 Difference of statistical measures (e.g. average, min. and max.)

M3-2-2 KS test

M3-2-3 Wasserstein distance

3.6.2. Case Study

This section discusses the case studies that are planned for the evaluation.

3.6.2.1. Requirements

REQ1 The case study is an application implemented in the 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 components which can be deployed allocations on the hardware nodes.

3.6.2.2. Application Concept

This section discusses the concepts of two applications for the case study based on the requirements listed in subsubsection 3.6.2.1. When the satisfaction of a requirement by the application concept is contentious, additional discussion is provided.

CSAC1 Realistic application using AppSpace samples
SICK provides a large number of small application samples to the public [20]. 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 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 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 2.3 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 [19]. These applications can be used by SICKs customers without customization and are in use in industrial settings. The applications from the 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.

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

Table 3.1.: Requirements fulfilled by the case study concepts. Fulfilled requirements are ticked a check mark. Requirements in brackets are separately discussed in subsubsection 3.6.2.2.

An overview over the application concepts in terms of the requirements is depicted in Table 3.1. Technically, both case study concepts satisfy all the requirements. Checkmarks in brackets denote contentious requirements. These are discussed in the previous listing.

3.6.2.3. Applications

This section lists the example applications with which the adapted approach will be evaluated.

CS1 Using **CSAC1** and simulated hardware The extent of the application built using AppSpace Samples is left intentionally open, so the app can be a better fit to the implementation. The approach is still evaluated using the real world application (see the next list item).

CS2 Using **CSAC2** and a hardware demonstrator

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. During the refactoring / reimplementation (see **CSAC2**) these components can be separated into multiple components. With a sufficient communication setup between the components, it is possible to distribute the components between the hardware SIM and e.g. an instance of the AppEngine that runs in the cloud. This would allow the evaluation of the impact of the allocation on the overall performance.

In addition to the allocation another variable than can be evaluated is the image frequency that can be ingested with different configurations.

Evaluating both applications is intended but ambitious because of the time constraints put on this thesis. Especially the case study “modifications” like reverse engineering a real-world application takes a significant amount of time.

3.6.3. Experiments

This section discusses possible experiments to acquire the metrics listed in the GQM plan, see subsection 3.6.1.

E1 for Q1-2 The accuracy of on up-to-date instance of the PCM can be measured as follows:

1. Simulate the execution speed of multiple services of an application using the up-to-date PCM instance.
2. Instrument the system interface of corresponding to the SEFFs.
3. Execute the instrumented application on hardware corresponding to the simulated component allocation.
4. Compare the monitoring results with the results of the simulation and calculate the statistical metrics.

E2 for Q1.2-1 The change propagation can be evaluated as follows.

Let $C := \{c_1, c_2, \ldots, c_n\}$ be the suitably long development history of the application. Let $c_i \in C$ further be a commit that introduced architectural changes to the application.

1. Create new PCM instances p_i and p_n for c_i and c_n respectively. This can be achieved using:
   - Automatic generation of the PCM instance, which has methodical drawbacks discussed in **T3**.
   - Manual creation of the PCM instance, which might not be feasible depending on its extents
2. Propagate the changes $\{c_{i+1}, c_{i+2}, \ldots, c_n\}$ to p_i creating $p'_{i+1}, p'_{i+2}, \ldots, p'_n$.
3. The change propagation works correctly if $p'_n \equiv p_n$ now holds.

3.6.4. Constraints of the evaluation

This section discusses constraints of the evaluation. The constraints are the cause of the limited time of the thesis.

Component Communication Components can directly call other components on the same computing node. For remote components a HTTP / REST Application Programmable Interface (API) may be used for component interaction. Direct local invocation of another components API incurs less overhead than the REST call. It may be necessary to constrain the evaluation to REST calls, even for other local components. This would simplify the simulation of multiple component allocation schemes,

Automatic Generation of Component Allocations It is conceivable that the CIPM approach could be extended to automatically generate all possible component allocations for the developer. Because of time constraints, this is out-of-scope for this thesis. The approach will only simulate a set of predefined component allocations.

Pareto Optimum of Component Allocations The CIPM approach will be adapted to simulate multiple component allocations. This includes the simulation of component allocations using different hardware nodes (e.g. different SIMs). Different hardware nodes (or cloud resources) have varying compute capabilities and therefore the application will experience faster execution on faster nodes. More capable hardware is usually more cost intensive. It is conceivable that the approach could be extended to automatically generate component allocations on varying hardware nodes, simulate the execution using the allocation scheme and produce a pareto front of achievable performance compared to the hardware cost. As this would require an in-detail accounting of cost factors including e.g. the cost of communication with remote resources. Because of time and subject constraints, this notion is out-of-scope for this thesis.

3.6.5. Threats to Validity

This section lists threats to the validity of the evaluation.

T1 The evaluation does only consider applications from the SICK AppSpace ecosystem. Consequently, the evaluation only considers Lua applications. This threatens the generalisability of the evaluation on other IIoT applications. Future work could investigate different IIoT applications and different programming languages to mitigate this threat.

T2 The case studies were not designed for variable allocation of components and therefore need to be adapted. This may indicate that performance prediction of varying allocation strategies is not relevant for the actual case studies.

On the contrary the requirements of the application examples have changed in the context of the case study. The evaluated applications are more flexible and could be used in a wider range of usecases when compared the original applications.

The impact of selecting different compute node hardware could still be evaluated without adapting the applications.

T3 One aspect of the evaluation is the correct updating a performance model instance when source code changes. In previous work this was achieved by comparing an automatically updated model instance with a pre-existing reference model instance. As there are no such reference models available the evaluation needs to fall back to comparing the updated model to an automatically generated reference model. This methodical weakness could be mitigated by manually creating a reference model in future work.

4. Organizational Matters

This chapter will outline organizational matters of the proposed thesis. The proposed schedule for the thesis can be found in section 4.1. A risk assessment regarding the approach and schedule is presented in section 4.2.

4.1. Schedule

The proposed schedule for the thesis is depicted in Figure 4.1. The schedule is designed for incremental evaluation of implemented thesis milestones. The application for **CS1** is implemented first, so other components can be evaluated against the application. Furthermore, the thesis is written continuously.

4.2. Risk Assessment

This section discusses the risks of the approach and the schedule. The execution of the approach has the following risks:

Code Model Obtaining a usable code model from the source code of the SensorApps is paramount to the approach. If the intended approach is not feasible, a different approach needs to implemented, which may cost significantly more time than anticipated.

Low CPR reusability It is expected that some of the existing Consistency Preservation Rules (CPRs) can be reused for the adapted prototypical implementation. Should this assumption be incorrect, these CPR need to be implemented from scratch, which will require more time.

Long implementation times The implementation can potentially take significantly longer than anticipated. This is due to he extent of the approach.

Mitigations for these risks are as follows:

Time Buffers Multiple time buffers are introduced in the schedule so unexpected time overruns can be compensated for.

Reduction of Case Study Extents It is possible to only evaluate one of the planned applications for the case study. This would significantly reduce the time needed for the evaluation. Time could also be conserved restricting the evaluation to simulated hardware and not using the hardware demonstrator.

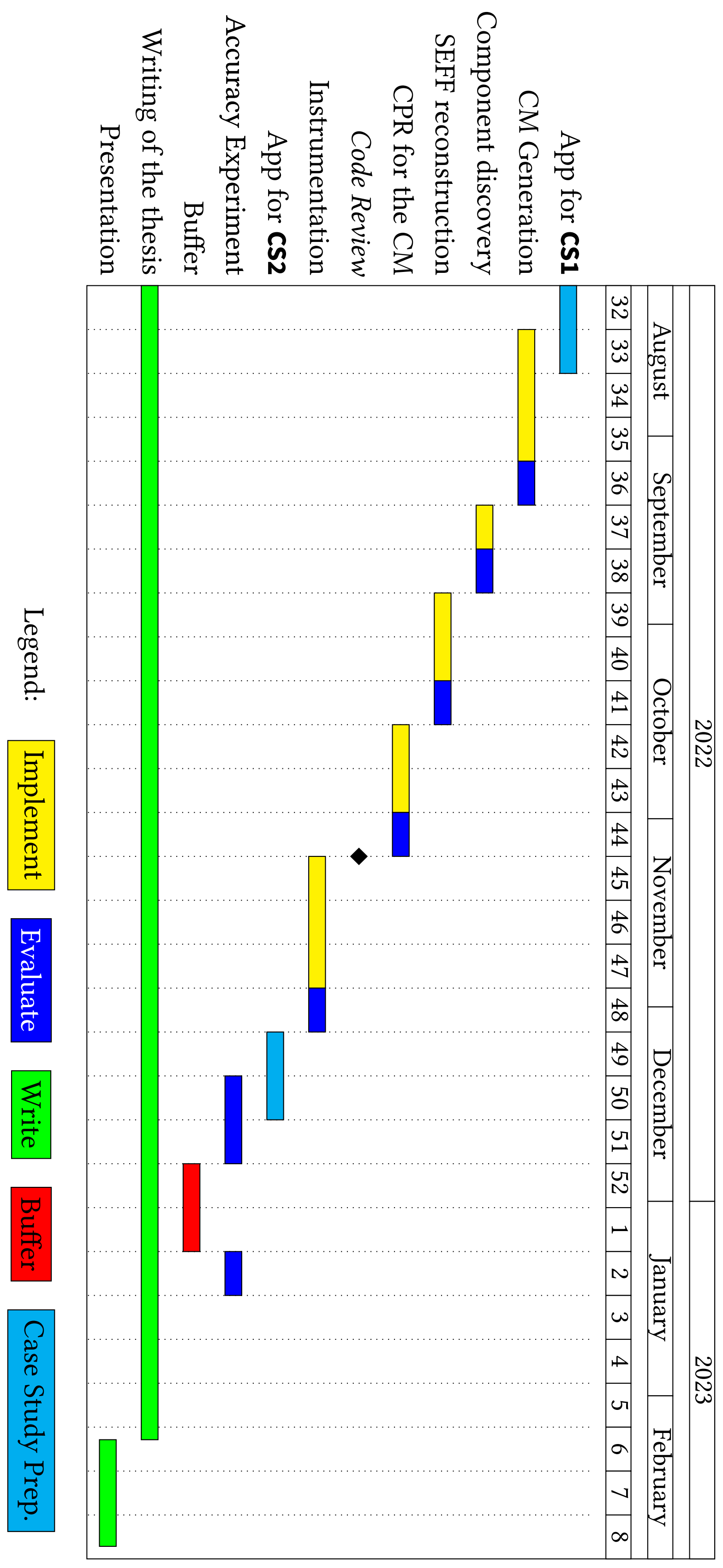


Figure 4.1.: The proposed schedule of the thesis.

Constraint Evaluation Corresponding to subsection 3.6.4 the structure of applications can be made more consistent by only using HTTP transport for inter component interaction.

Shifting accuracy evaluations to further work The accuracy of the Continuous Integration of Performance Models (CIPM) approach was previously evaluated [16]. The reevaluation of the accuracy of the approach is not strictly needed to demonstrate this approach.

5. Related Work

This chapter outlines related work in the context of Quality-of-Service (QoS) of industrial internet of things (IIoT) applications. QoS is a general term for some of the non-functional characteristics of network-based or distributed applications. Examples for QoS parameters are

- Response time: the time the application takes to respond to requests or Application Programmable Interface (API) calls. Response time can also be simulated by the Palladio Component Model (PCM) and is an important output of architecture-based performance prediction (AbPP).
- Throughput: The amount of requests that can be processed per unit of time.
- Reliability: The ratio of time for which the service is available

The following sections present three different approaches for the measurement or devel

5.1. Monitoring

Eyhab presented an approach for a QoS aware microservice IIoT architecture: *mQoSm* [8]. A goal of the approach is to select existing microservices for IIoT applications based on the observed QoS parameters. The approach comprises a monitoring framework for the measurement of QoS parameters as previously listed. The framework periodically measures the QoS parameters by employing a controller. Further, the controller can discover new microservices on the network. Because the measurements are taken periodically, the approach can account for fluctuations of the QoS.

Because mQoSm operates at runtime of the microservice application [8]. The approach can therefore not be used for a-priori considerations of application performance, in contrast to the Continuous Integration of Performance Models (CIPM) approach [13].

5.2. Network Queueing Model

Duttagupta et al. presented an approach for predicting the performance of Internet of Things (IoT) applications for their proprietary IoT platform [6]. With rising adoption of the platform, more and more sensors are deployed and inject more data into the backend of the platform. The authors want to predict QoS parameters (e.g. throughput) and performance bottlenecks of the applications. Further a-priori considerations include how the application would behave should a different kind of storage backend be used.

The authors model the application performance on the request and response level using the Java Modelling Tools (JMT) framework [6, 3]. They build a Queueing Network Model (QNM) for relevant components of the IoT application. Using the QNM they can predict the scalability of the application. The predictions are calibrated using performance tests of the respective APIs. The authors further present an approach to make the prediction results transferable to similar deployments of an application.

In contrast to the CIPM approach the approach by Duttagupta et al. does not simulate the application performance on the software architecture level [13, 6].

5.3. Model-Based development of IIoT Applications

Muthukumar et al. presented a model-based software engineering (MBSE) approach to designing and verifying a cloud-based IIoT application [17]. An IIoT gateway serves as an interface between the "Automation Cloud" and "Plant-level Control". It enables the authors to use commercial-of-the-shelf IoT devices on the plant level, which may not be compatible with the cloud. The authors use multi-view modelling for the IIoT application. Examples for the views of the domain are devices, architecture and behaviour. Muthukumar et al. employ the Automation Markup Language (AutomationML) to integrate the different views and act as a meta-model for the IIoT application [5, 17].

In contrast to the CIPM approach, Muthukumar et al. leverage auto coders to produce the actual source code [13, 17]. Because of their top-down approach, the approach is not compatible with pre-existing code. This is not the case with the CIPM approach.

Bibliography

[1] Kent L. Beck et al. “Manifesto for Agile Software Development”. In: 2013.

[2] 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.

[3] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. “JMT: Performance Engineering Tools for System Modeling”. In: *SIGMETRICS Perform. Eval. Rev.* 36.4 (Mar. 2009), pp. 10–15. ISSN: 0163-5999. DOI: 10.1145/1530873.1530877. URL: https://doi.org/10.1145/1530873.1530877.

[4] 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.

[5] Rainer Drath et al. “AutomationML - the glue for seamless automation engineering”. In: *2008 IEEE International Conference on Emerging Technologies and Factory Automation*. 2008, pp. 616–623. DOI: 10.1109/ETFA.2008.4638461.

[6] Subhasri Duttagupta et al. “Performance Prediction of IoT Application: An Experimental Analysis”. In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT’16. Stuttgart, Germany: Association for Computing Machinery, 2016, pp. 43–51. ISBN: 9781450348140. DOI: 10.1145/2991561.2991572. URL: https://doi.org/10.1145/2991561.2991572.

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

[8] Al-Masri Eyhab. “QoS-Aware IIoT Microservices Architecture”. In: *2018 IEEE International Conference on Industrial Internet (ICII)*. 2018, pp. 171–172. DOI: 10.1109/ICII.2018.00030.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] Manar Mazkatli and Anne Koziolek. “Continuous Integration of Performance Model”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.* ICPE ’18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 153–158. ISBN: 9781450356299. DOI: 10.1145/3185768.3186285. URL: https://doi.org/10.1145/3185768.3186285.

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

[16] 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.

[17] N. Muthukumar et al. “A model-based approach for design and verification of Industrial Internet of Things”. In: *Future Generation Computer Systems* 95 (2019), pp. 354–363. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.12.012. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18321137.

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

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

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

[21] Object Management Group (OMG). *OMG® Unified Modeling Language® (OMG UML®).* Version 2.5.1. 2017.

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

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

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

A. Appendix

Glossary

microservice The microservice architecture is similar to component-based architectures. The application is split into multiple microservices, which focus on a certain aspect of the application. The microservices usually communicate via specified REST interfaces.

Acronyms

AbPP	architecture-based performance prediction
API	Application Programmable Interface
AST	Abstract Syntax Tree
AutomationML	Automation Markup Language
CD	Continuous Delivery
CI	Continuous Integration
CIPM	Continuous Integration of Performance Models
CM	Code Model
CPR	Consistency Preservation Rule
CROWN	common reusable objects wired by name
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
GQM	Goal Question Metric
IDE	Integrated development environment

IIoT industrial internet of things

IM Instrumentation Model

IoT Internet of Things

JMT Java Modelling Tools

KS Kolmogorov-Smirnov

MBSE model-based software engineering

OMG Object Management Group

PCM Palladio Component Model

PMP performance model parameter

QNM Queueing Network Model

QoS Quality-of-Service

RDSEFF Resource Demanding SEFF

SEFF Service Effect Specification

SIM Sensor Integration Machine

VCS Version Control System

VSUM Virtual Single Underlying Model

XMI XML Metadata Interchange