

Adaptive Monitoring for Continuous Performance Model Integration

Master's Thesis of

Noureddine Dahmane

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

Reviewer:	Prof. Anne Koziolk
Second reviewer:	Prof. Ralf Reussner
Advisor:	M.Sc. Manar Mazkatli

01. August 2018 – 31. January 2019

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.

PLACE, DATE

.....
(Nouredine Dahmane)

Contents

Abstract	1
1 Introduction	3
2 Foundations	5
2.1 The Palladio Component Model	5
2.1.1 Palladio Models	5
2.2 Vitruvius	7
2.3 Java Model Parser and printer	8
2.4 Eclipse Modeling Framework	8
2.5 Automated Coevolution of Source Code and Software Architecture Models	8
2.6 Kieker Monitoring Framework	10
2.7 Source Code Model eXtractor	11
2.7.1 SoMoX SEFFs Reconstruction	12
2.7.2 Incremental SEFF Reconstruction	12
2.8 Continuous Integration of Performance Model	13
2.9 Iterative Performance Model Parameter Estimation Considering Para- metric Dependencies	14
2.10 DevOps	15
3 Problem Statement	19
4 Adaptive Monitoring for Continuous Performance Model Integration	21
5 Evaluation	23
5.1 First Section	23
6 Related Work	25
7 Conclusions and Future Work	27

List of Figures

2.1	PCM Models and transformations	6
2.2	Example of source code and SEFF	7
2.3	The steps used in co-evolution approach to keep architecture models and the source code consistent	16
2.4	Overview of the Kieker's architecture	17
2.5	Example of manual instrumentation of source code	17
2.6	Example of a Kieker Probe using AOP	17
2.7	Overview of the Incremental Reconstruction Process of SEFF	18
2.8	CIMP activities	18

Abstract

English abstract.

1 Introduction

bla bla bla

2 Foundations

2.1 The Palladio Component Model

Palladio Component Model (PCM) is a component-based software architecture approach, that can be used to predict and evaluate the performance and the reliability of component-based software systems at design stage. For performance prediction, it makes it possible to analyze the components of the software architectures before implementation and thus it can for example detects bottlenecks, predict response time and predict throughput. For software reliability, PCM specifies metrics like detecting the probability of failures on demand.

2.1.1 Palladio Models

To allow the prediction of the Performance and the reliability of a software system, PCM defines four different roles to create four different models 2.1. The Repository model, which is created by developers. Developers specify and implement components, interfaces, provided roles, required roles and signatures for the repository. In the context of PCM, a component is by definition a block of software, that can be composed, deployed and customized without requiring the understanding of its internals. Moreover, the roles in PCM specify the relation between the components and the interfaces. Developers should also specify the internal behavior of the components in term of Service Effect Specification (SEFF). The Assembly model is created by software architects. Software architects use the existing components in repository to create the software system. The allocation model is created by system deployers. Deployers are responsible for specifying the environment resource, like Servers, CPUs, HDDs and network connection. Afterwards, they decide which assembly can be deployed in which resource. The usage model is created by domain experts. Domain experts provide information about the interaction between the system and the users. Additionally, domain experts can define the relevant critical usage scenarios and the inputs parameters values.

In the context of this thesis, we will use the information provided by SEFFs. Thus, we will explore the related concepts based on [2] and [11].

2.1.1.1 SEFFs

Service Effect Specification (SEFF) were firstly presented by [11], which describes like a UML Activity Diagram the control flow of component services. For each provided service, SEFF describes how services in the required interface are called in the provided service. Moreover, as mentioned before, the components in PCM can be used

without understanding their internals and thus as a black box. However, SEFF turns a component to a gray box by describing the behavior of its provided services.

The ResourceDemandingServiceEffectSpecifaction (RDSEFF) used in PCM to predict the performance, is an extension of SEFF. RDSEFF offers the possibility to add performance inputs values associated to each activity of SEFF. Moreover, to each component provided service, developers can specify a RDSEFF in order to describe how the service uses the hardware/software resources and how it calls the component's required services.

In the rest of this thesis, we will indicate RDSEFF as SEFF in order to avoid ambiguity. The principal elements of SEFF will be explored in the following. SEFF includes the so called ExternalCallActions which present the call of a required Service within the SEFF. InternalActions are used within SEFF to abstract the internal computation of the component. Moreover, an internal action can be defined as a set of successive instruction, that do not include any external call from other components. LoopActions within SEFF are specified to indicate the number of times the sub control flow within the loop is going to be executed. BranchActions models the branches within the control flow. The execution of a branch can be decided either on an input parameter or a probability. InternalActionActions refers to the internal behavior of a component, that can be only used by the services of this component.

To have a better understanding of these concepts, we put them together in an example 2.2, which depicts on the right hand the source code and on the left hand the corresponding SEFF model. The implementation of the Service service() starts with an internal call of an inner method. Since the inner method innerMethod1() represent an internal computation and does not make any external call, it is seen as an internal action within SEFF. Within the next branch is seen as a BranchAction, because there is a call of the service service1() of the component componentB. In the second branch transition, there is loop, which makes call of the external service service2(), that's why it's represented as a LoopAction within the corresponding SEFF.

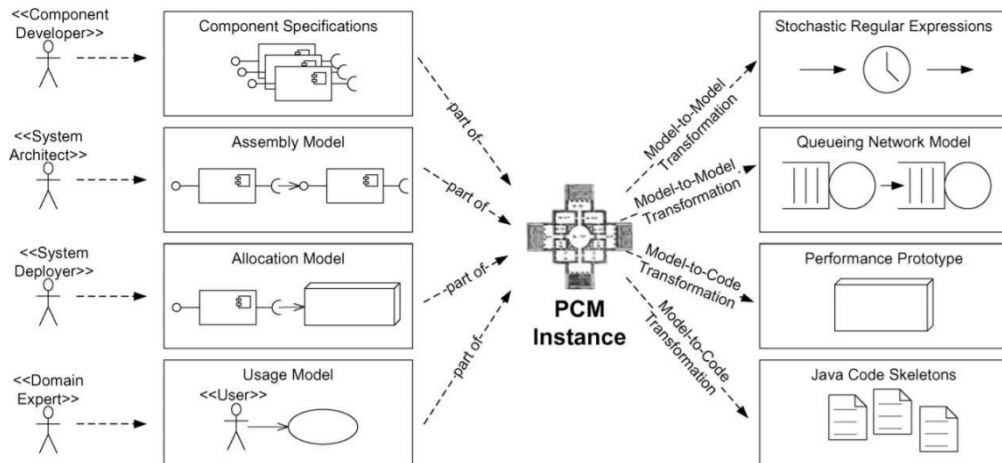


Figure 2.1: PCM Models and transformations

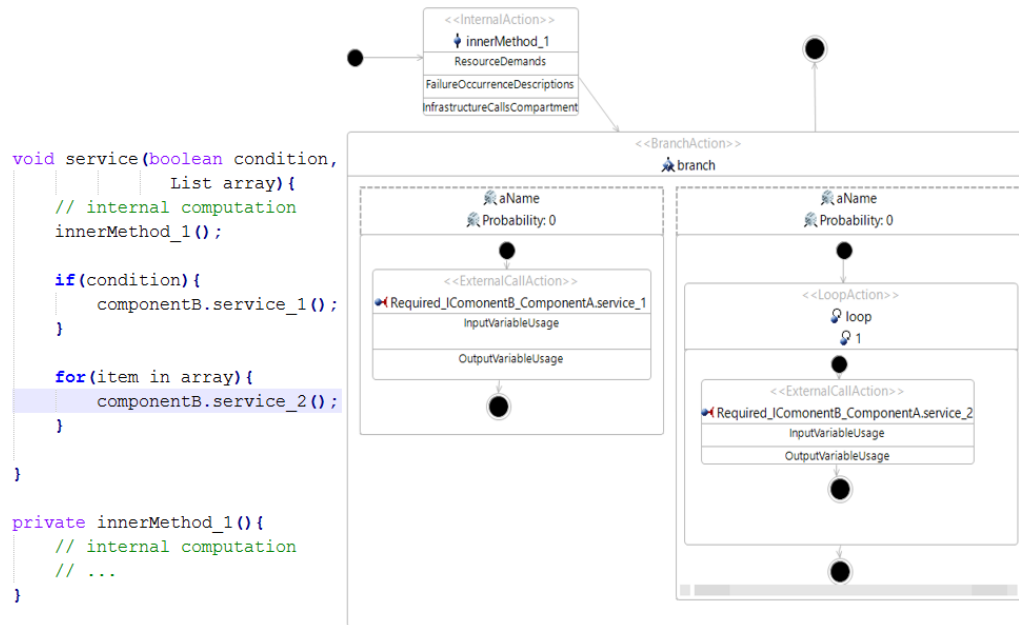


Figure 2.2: Example of source code and SEFF

2.2 Vitruvius

The Vitruvius approach is a view-based [5] engineering approach, which was introduced in [4, 12]. Vitruvius can be used to keep the models of a system consistent. In Model Driven Development (MDD) [1, 17], the whole system can be represented by different models, that describe different aspects of the system, even the source code of the system is seen as a model. These models can be changed separately and became inconsistent, for example if the source code of a system has changed, the architecture model of the system has to be also updated. In this case, architectures must capture the changes in the source code and update accordingly the associated architecture model. Therefore, Vitruvius performs this task automatically by defining consistency rules that define how the changes in a model must be transformed to another model.

Vitruvius uses views to make access to the models possible. This approach reduces the complexity of dealing with the whole models, because views present only a part of the model. Therefore, developers can focus only on the relevant parts of the system.

Vitruvius defines the so called Virtual Single Underlying Model (VSUM), which contains all information related to a system. VSUM contains the meta-models instances, that are used within a system. Moreover, Vitruvius provides the correspondence meta-model, which offers the possibility to map between correspondent elements of different model, like SEFFs in PCM and methods in Java source code.

To enable the process of keeping models consistent within Vitruvius, developers should provide and implement two concepts defined by Vitruvius, namely Domains and Application. A Vitruvius Domain represents a defined meta-model in the system and provides information for its use within the Vitruvius framework. Vitruvius Domains can be reused, for example the Domain of Java meta-model can always be used in systems that are implemented using java. Vitruvius Applications determine the relation between two Domains. They specify how changes in a Domains meta-model instance should be transformed to the instance of another Domains meta-model instance. Vitruvius Applications can be also reused in many environments, if the are needed.

2.3 Java Model Parser and printer

Java Model Parser and Printer (JaMoPP) [6] is a parser and printer for the Java language. JaMoPP defines a complete meta-model for the Java language based on the meta modeling language Ecore [18]. JaMoPP parser allows to parse a Java source code into a Java model, and the JaMoPP printer allows to print a Java model into Java source code. JaMoPP can convert Java source code into an EMF model, which can be manipulated using model driven techniques, like models transformation. Using JaMoPP, we can for example, parse a Java file, which contains Java source code, add new Java statements after or before an existing statement and print back the changes in the Java source code file. The creation of JaMoPP is based on EMFText [7], which allows to define text syntax for languages described by an Ecore meta-model.

2.4 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) is a framework, that enable the creation of meta-models based on Ecore. Moreover, EMF provides the following facilities:

1. Source code generation from meta-models.
2. Generate an instance of eclipse in order to instantiate meta-models and edit them.

2.5 Automated Coevolution of Source Code and Software Architecture Models

The approach of co-evolution of source code and component-based software architecture presented in [15, 14] is based on the platform Vitruvius present above. It gives developers and architectures the possibility to keep the architecture and the source code of a software system consistent. The Consistency in this approach is kept in both directions, that means, if developers changed the source code of a system, the

corresponding architecture will automatically be changed, and if architectures updated the architecture of the system, the source code will also automatically be updated. The co-evolution approach uses PCM models as architectures models. Therefore, it describes the behavior of the source code in term of SEFF. Hence, a special objective of the co-evolution approach is to keep incrementally the behavior of the source code consistent with the source code itself. The incremental up-to-date of the behavior model of the source consists of only the part of the SEFF that corresponds to the changes performed in the source code. For example, if the developers update a service S of Component A, only the SEFF of the service S will be reconstructed but not the whole SEFF model of the system.

The co-evolution approach uses two different concepts to preserve different modes consistent, namely model-driven engineering and change-driven engineering. Moreover, the co-evolution approach uses the concepts of Vitruvius to keep tracks of the models, that represents the system.

The co-evolution approach considers all involved artifacts as models. Hence, the model-driven engineering is used in this approach as a main concept. In model-driven engineering, all artifacts of the system are represented by models and the models are centric in the development process. That means, that the source code must be also represented within the co-evolution approach as a model. Therefore, in the case of Java, the co-evolution approach uses JaMoPP to parse the Java source code to a mode, on which the change-driven techniques can be applied, like model to model transformations.

The co-evolution approach uses change-driven engineering in order to reacts on changes, that the users perform on models, especially the architecture model and the source code model. These changes are monitored, converted in a Vitruvius change model and propagated. The propagated changes can be captured and transferred using consistency preservation rules to the target model. The consistency preservation rules are bidirectional and defined between the architecture meta-model and the source code meta-model.

The co-evolution approach was applied to the Palladio Component Model (PCM) as an architecture model and Java source code. However, the concepts presented in this approach can be applied to other component-based architecture model, like UML component-diagrams, and other object-oriented languages. In the following, we will review the application of the co-evolution approach to PCM and Java source code and the steps, that are required to keep PCM models instances and Java source code consistent.

Figure 2.3 shows the steps, that are used to keep architecture models and the source code consistent:

1. Step (0): users change either the PCM repository, the PCM system or the Java source code.

2. Step (1): monitors capture changes on the models.
3. Step (2): monitors trigger Vitruvius Framework and pass it the changes.
4. Step (3): based on these changes, the Vitruvius framework executes the consistency preservation transformations.
5. Step (4): the transformations use information from the changes and the correspondence model to execute the preservation rules.
6. Step (5): the transformations update the models.

When it comes to an automatic update of SEFF in step (5), the co-evolution approach executes an incremental SEFF reconstruction step instead of transformation.

2.6 Kieker Monitoring Framework

Kieker is an extensible Java-based application performance monitoring and dynamic software analysis framework [8]. Figure 2.4 shows the architecture of Kieker framework. It's composed from two main components, namely the monitoring component and analysis component. The analysis component can be used to read monitoring data, analyze and visualize them for a certain purpose, like generating UML sequence diagram, dependency graphs or Markov chains.

The monitoring component is responsible for source code instrumentation, data collection and data logging. The Monitoring probes are responsible for collecting the monitoring data and send them to the monitoring controller component, which instantiates a monitoring record for every probe. The Monitoring writer component receives the monitoring records from the monitoring controller and serialize them to the monitoring log/stream.

A monitoring Record represents the measurement data gathered in a single measurement. Kieker provides the possibility to store different types of records for different types of probes. Kieker offers the possibility to create customized probes. We can create new probes either manually by extending the interface `IKiekerMonitoringProbe` or automatically by using the Instrumentation Record Language (IRL) [9]. For example, in one record, we can store the signature and the response time of a method, in another record, we can persist the response time of specific number of statements inside a method.

A monitoring Probe represents the monitoring logic used to collect measurement data from the application. There are two ways to use probe within Kieker. There is the manually instrumentation, which consists of mixing the instrumentation logic with the business logic of the application. Figure 2.5 show an example, in which the monitoring probe is implemented by mixing monitoring logic with business logic,

in this example we create a probe that logs the name of the called service, the start time and end time. Kieker includes also probes based on Aspect Oriented Programming (AOP) [10], which helps to separate the instrumentation logic from the business logic. Kieker defines AOP based monitoring probes like `OperationExecutionAspectAnnotation` and `OperationExecutionAspectAnnotationServlet`. Figure 2.6 shows how the instrumentation looks like, when using AOP for probes implementation.

Using AOP to instrument the source code has the advantage of separating concerns. However, this technique has a limitation, when it comes to the monitoring of certain statements of inside a method, like monitoring the number of executions of a loop or the probability of a branch execution, because the annotation possibilities on these cases are out of box.

2.7 Source Code Model eXtractor

Source Code Model eXtractor (SoMoX) is a reverse engineering approach, which has been developed by Krogmann [13]. SoMoX is able to reverse-engineer software component architectures. Moreover, SoMoX can extract a PCM repository from source code and creates a PCM system derived from the repository. The repository created by SoMoX contains mainly components, interfaces, roles and SEFFs. Thus, the results of reverse engineering of SoMoX depend strongly on the project implementation to reverse-engineer which means that SoMoX delivers best results, if the analyzed source code followed a component-based architecture.

In the following, we will review only the features of SoMoX that are involved in the context of this thesis.

In order to create the architecture of a software system, SoMoX reverse-engineer the software system using the following steps:

1. Parse the source code into a model.
2. Detect components and interfaces using metrics.
3. Detect data types and signatures using metrics.
4. Reconstruct the SEFFs.

For the source code parsing in the first step, SoMoX uses JaMoPP to create an EMF model of the Java source code that can be manipulated. In this thesis, we will also use JaMoPP source code parsing and manipulation purpose.

For components and interfaces detection, SoMoX uses various source code metrics and combine them to determine detection strategies for architecture elements. These metrics must be given each a value between 0 and 100 by the user of SoMoX. The value of the metric tells SoMoX the impact factor, for example the value 0 of a metric

means that the impact factor is low, whereas the value 100 of the metric means that the impact factor is high.

The reconstruction of SEFFs which aims to reverse-engineer the statical behaviour of the source code is done by analysing the methods of the source code. This step has been extended by Langhammer [14] and its results are used in thesis. In the following, we will explain briefly how the reconstruction of SEFFs is done within SoMoX and how it was extended by Langhammer.

2.7.1 SoMoX SEFFs Reconstruction

For SoMoX SEFFs reconstruction which is done in the last step of the reverse-engineering process, SoMoX uses two models which were created in the first and the second steps. The first model is the Java source code model which was created in the first step and the second is the so-called Source Code Decorator Model (SCDM) which was created in the second step. The SCDM contains the information that map between the source code model elements and the reverse-engineered architectural model elements.

In order to create the SEFF of a method, SoMoX analyses the source code of the method which was detected as a provided method of a component. This analysis is performed in two steps which are described in the following.

In the first step, the SoMoX visits all the method calls within the analysed method and classified them in three categories. The first category contains the component-external method calls which are considered as required roles. The second category contains library calls which considered as calls to a third-party library like `java.lang`. the second category contains the component-internal calls which are calls to the inner methods of the component.

In the second step, SoMoX creates the SEFF for the method. To do so, SoMoX visit again the statements in the source code of the method in order to find the following SEFF elements if they exist: `ExternalCallActions`, `BranchActions`, `LoopActions` and `InternalActions`. `BranchActions` and `LoopActions` are created for branches and loops. A Branch or a Loop is considered as `BranchAction` or `LoopAction` if it has an external method call, else it will be combined with an `InternalAction`.

2.7.2 Incremental SEFF Reconstruction

Langhammer has proposed in his Co-evolution approach an incremental SEFF reconstruction approach Figure ?? which can build the SEFFs for only the changed parts of the source code. In contrast to SoMoX, the incremental SEFF reconstruction neither require the parsing of the complete project source code nor the SCDM. Moreover, the

SEFF of the smallest unit that can be currently incrementally reconstructed is the SEFF of a method.

The incremental SEFF reconstruction is integrated in the co-evolution approach that means it can use functionalities and information provides by Vitruvius. Moreover, the incremental SEFF reconstruction is done in change-driven way, that means change that happened in the source code are captured in a Vitruvius change model instance and can be treated in order to regenerate the SEFF of the method in which they belong. To classify the method calls which is necessary for SEFF reconstruction, Langhammer uses the current preservation rules and information from the Vitruvius correspondence model. More details on this step can be found in his thesis [14].

2.8 Continuous Integration of Performance Model

Continuous Integration of Performance Model (CIPM) is an approach proposed by Mazkatli and Koziolk [16] in order to extract incrementally and iteratively the Performance Model from the source code and enrich it by the Performance Model Parameters (PMPs). Furthermore, CIPM aims to keep the source code and the extracted Performance Model consistent during the system development. CIPM extends the Continuous Integration (CI) and the Continuous Deployment (CD) of the source code with a continuous integration of the performance model.

To achieve that Mazkatli and Koziolk used the Coevolution approach developed by Langhammer (Section 2.5) which uses the Palladio Performance Model as a Performance Model and the Vitruvius Platform to keep incrementally and in a change-driven way the source code and the corresponding Performance Model consistent. CIPM uses likewise a change-driven way to enrich the extracted PM by the Coevolution process with PMPs. It defines the consistency rules that minimize the monitoring of the source code execution and the analysis overhead. In addition, it specifies a self-validation process that validates the estimated PMPs. To release that, CIPM automates four activities that are executed in each iteration 2.8. In this section we will describe only the first two activities because we based our work on them, more details on the other activities can be found in [16].

The activities in CIPM are represented by the Reaction Language (RL) routines. RL is used in Vitruvius to describe the consistency rules and it's based on two concepts, namely reaction and routine. A reaction specifies changes and triggers a routine that contains the consistency rules that should be executed in reaction to these changes.

The first activity in CIPM is responsible for updating the structure of the performance model, the usage model and the probes used in the activity two. The structure of the performance model is updated based on the work of Langhammer [15]. The usage model is extracted incrementally from the test cases. For the probes generation

CIPM specifies an Instrumentation Metamodel (IMM) and added it to VSUM. IMM contains and manages the instrumentation points and the weaving information based on Aspect-oriented Programming (AOP) [10]. CIMP defines the consistency rules that keep the instrumentation model and the source code consistent. For example, a new probe can be added to the instrumentation model, when a part of the source code was added that corresponds to a new SEFF element.

The second activity uses the probes from the first activity, instruments the source code and creates the monitoring data. For the monitoring data, CIPM specifies a Measurement Meta-model (MMM) and added it to VISUM. MMM describes the data structures of the different monitored data as well as the consistency rules to keep it consistent with the IMM. For example, after the monitoring phase is done, if a SEFF element had enough monitoring data, this element has to be deactivated in the instrumentation model.

2.9 Iterative Performance Model Parameter Estimation Considering Parametric Dependencies

In his master thesis, Jägers has created an approach that can estimate Performance Model Parameters taking into account the parametric dependencies. The approach is based on the vision presented by Mazkatli and Koziolk (Section 2.8), it extends precisely the concepts introduced in the activities three and four in Figure 2.8. The approach is designed to make iteratively the estimation and thus reduce the overhead resulting from the estimation for the whole system.

Jan uses the Palladio Performance model as a Performance Model for his approach. Since the Palladio Performance Model is expressed in terms of SEFF, Jan estimates the performance model parameters for loop iteration, resource demands, branch transitions and external call arguments. He specifies diverse predictive models for estimating Performance model parameters. He uses decision trees to create a predictive model for branch transitions. For loop iterations and resource demands he uses regression analysis. The predictive models are related with service call arguments. The predictive models can be transformed into stochastic expressions in order to use them for enriching performance model.

This approach uses as inputs the monitoring data that are generated from instrumenting and executing the source code that corresponds to the performance mode for which the estimation is done. The monitoring data are structured in records that contain diverse information about diverse source code elements like loop record, branch record response time record etc. Since the execution depends strongly on the monitoring data, in order to keep the estimation iterative, the monitoring must be also done iteratively. this includes also the fine-grained, automatic, iterative instrumentation of the source code. the instrumentation has to be fine-grained because the

estimation requires specific information about the program elements like the number of loop execution, the response time of a specific source code that corresponds to an internal action. This thesis provides an approach that can generate iteratively the monitoring data for performance model. These monitoring data can be used by the approach of Jan to make the estimations.

2.10 DevOps

DevOps [3] is an agile development process which combines between development (Dev) and operations (Ops). The main goal of DevOps is to reduce the time between changes in the business process and the delivery of a solution to these changes. It achieves its goal by acting mainly on simplifying the communication between organizational and technical teams as well as introducing the automatization for the tasks that can be automated.

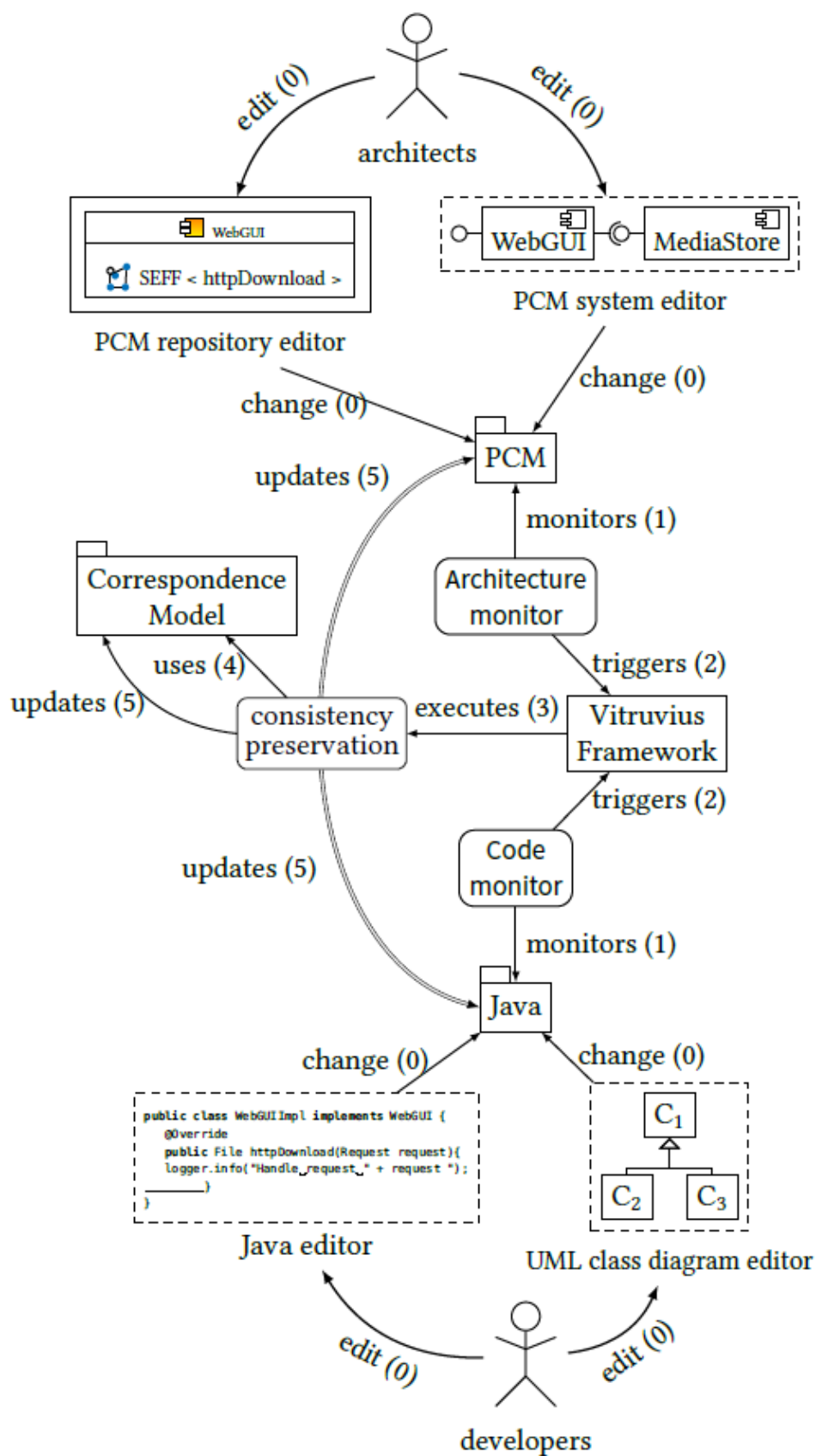


Figure 2.3: The steps used in co-evolution approach to keep architecture models and the source code consistent

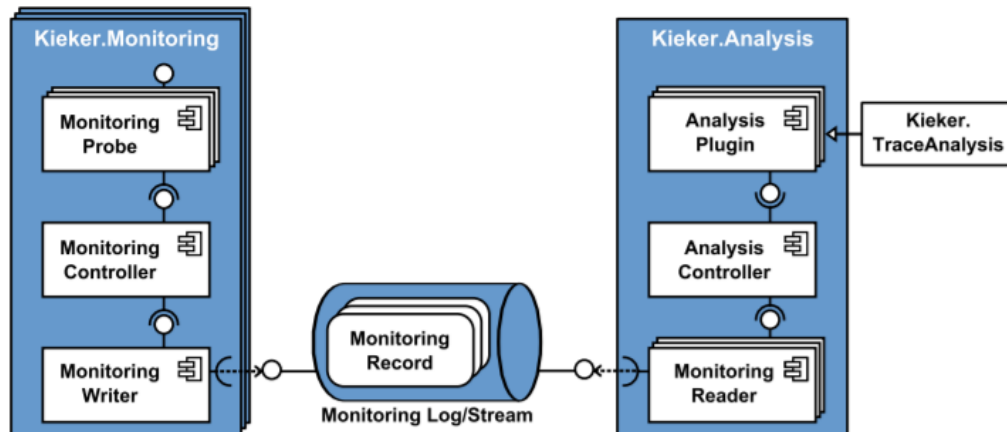


Figure 2.4: Overview of the Kieker's architecture

```

final long tin = MONITORING_CONTROLLER.getTimeSource().getTime();
// call of the service
this.serviceA();
final long tout = MONITORING_CONTROLLER.getTimeSource().getTime();
// Create a new record and set values
final MyResponseTimeRecord e = new MyResponseTimeRecord(
"serviceA", tout, tin);
// Pass the record to the monitoring controller
MONITORING_CONTROLLER.newMonitoringRecord(e);

```

Figure 2.5: Example of manual instrumentation of source code

```

@OperationExecutionMonitoringProbe
public void serviceA(){
    /*
    method content
    */
}

```

Figure 2.6: Example of a Kieker Probe using AOP

```

@OperationExecutionMonitoringProbe
public void serviceA() {
    /*
     * method content
     */
}

```

Figure 2.7: Overview of the Incremental Reconstruction Process of SEFF

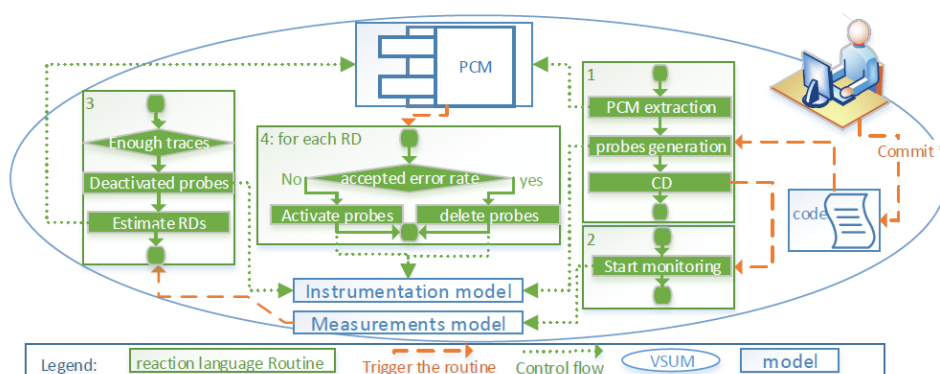


Figure 2.8: CIMP activities

3 Problem Statement

bla bla

4 Adaptive Monitoring for Continuous Performance Model Integration

5 Evaluation

5.1 First Section

6 Related Work

bla bla

7 Conclusions and Future Work

testinnnnnnnnnnnnnnnnnnnn

Bibliography

- [1] Thomas Stahl et al. “Modellgetriebene Softwareentwicklung : Techniken, Engineering, Management. 1. Aufl.” In: *1. Aufl. Heidelberg: dpunkt-Verl.* Vol. 303. 2005, pp. 23–38.
- [2] Steffen Becker, Heiko Koziolk, and Ralf Reussner. “The Palladio component model for model-driven performance prediction”. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.
- [3] Andreas Brunnert et al. “Performance-oriented DevOps: A research agenda”. In: *arXiv preprint arXiv:1508.04752* (2015).
- [4] Erik Johannes Burger. “Flexible views for view-based model-driven development”. In: *Proceedings of the 18th international doctoral symposium on Components and architecture*. ACM. 2013, pp. 25–30.
- [5] Thomas Goldschmidt, Steffen Becker, and Erik Burger. “View-based Modelling-A Tool Oriented Analysis”. In: *Proceedings of the Modellierung*. 2012.
- [6] Florian Heidenreich et al. “Closing the gap between modelling and java”. In: *International Conference on Software Language Engineering*. Springer. 2009, pp. 374–383.
- [7] Florian Heidenreich et al. “Derivation and refinement of textual syntax for models”. In: *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2009, pp. 114–129.
- [8] André van Hoorn et al. “Continuous monitoring of software services: Design and application of the Kieker framework”. In: (2009).
- [9] Reiner Jung. *An instrumentation record language for kieker*. Tech. rep. Tech. rep. Kiel University, 2013.
- [10] G Kiczales. “J. lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming”. In: *Lecture notes in Computer Science, ECOOP* (1997), pp. 220–242.
- [11] Heiko Koziolk, Jens Happe, and Steffen Becker. “Parameter dependent performance specifications of software components”. In: *International Conference on the Quality of Software Architectures*. Springer. 2006, pp. 163–179.
- [12] Max E Kramer, Erik Burger, and Michael Langhammer. “View-centric engineering with synchronized heterogeneous models”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. ACM. 2013, p. 5.

- [13] Klaus Krogmann. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis*. Vol. 4. KIT Scientific Publishing, 2012.
- [14] Michael Langhammer. “Automated Coevolution of Source Code and Software Architecture Models”. In: (2017).
- [15] 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.
- [16] Manar Mazkatli and Anne Koziolk. “Continuous Integration of Performance Model”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM. 2018, pp. 153–158.
- [17] D.C. Schmidt. “Guest Editor’s Introduction: ModelDriven Engineering”. In: *In: Computer 39.2*. Vol. 303. 2006, pp. 25–31.
- [18] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.