Karlsruhe Institute of Technology

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. Dr.-Ing. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf H. Reussner
Advisor:	M.Sc. Manar Mazkatli
Second advisor:	M.Sc. Dominik Werle

01. August 2018 – 14. March 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

. . . . . .

(Noureddine Dahmane)

Abstract

Model-driven performance prediction can be used in order to predict software performance at the design stage. Applying it in the agile development process can reduce time and effort spent on monitoring. However, building such a model manually is impractical in the agile development process because it tends to generate quick releases in shot cycles. Existing approaches that extract automatically Performance Model (PM) are based on reverse-engineering and/or measurement techniques. However, these approaches require to monitor and analyze the whole system after each iteration, which will cause a monitoring overhead.

The approaches that extract incrementally the PM from the source code and keep them consistent need a monitoring approach that provide them with the required monitoring information. Existing approaches either cannot provide fine-grained monitoring information or the instrumentation of the source code has to be done manually which is unsuitable for developers.

To overcome these problems, this thesis introduces an approach that can monitor the system in an adaptive way and provide PMs with the fine-grained monitoring information. Moreover, our approach is part of the continuous integration of PMs in the agile development process. We achieved this by introducing an automatic adaptive instrumentation approach of the source code. Furthermore, our instrumentation approach can monitor changes in the source code during system development and instrument the changed parts of the source code accordingly and automatically. During the system development, we collect automatically the monitoring probes that are needed to instrument the source code. When changes in the source code are committed, we automatically instrument the source code based on the monitoring probes that we've generated during the system development. The benefit of our approach will be to reduce the high monitoring overhead and to support the continuous integration of performance models.

In particular, our approach can be integrated with other approaches that keep the source code and the PM consistent. While these approaches tend to keep the source code and its behavior in terms of PM consistent, our approach will provide them with the needed monitoring data to estimate the PM parameters (like Resource Demands RDs, loop execution number, the probability of selecting a branch).

We evaluated our approach based on a case study. we showed that we could instrument incrementally and automatically the source code. Moreover, our evaluation showed that we could provide monitoring information that can be used for the calibration of performance model parameters considering parametric dependencies. Furthermore, using our approach and the case study we showed that we could reduce the monitoring overhead to 50%.

Zusammenfassung

Modellgetriebene Leistungsvorhersagen können zur frühzeitigen Abschätzung der Softwareperformanz verwendet werden. Der Einsatz solcher Werkzeuge im agilen Entwicklungsprozess kann den in die Überwachung investierten Aufwand reduzieren. Allerdings ist das bauen eines solchen Modells während des agilen Entwicklungsprozesses unpraktisch, da es oft dazu führt, Releases in kurzen Intervallen zu produzieren. Bisherige Ansätze, welche automatisch ein Performance-Modell (PM) generieren, arbeiten auf Grundlage von Reverse Engineering und/oder Messverfahren. Allerdings überwachen und analysieren diese Ansätze das gesamte System nach jeder Iteration, was einen Überwachungsaufwand erzeugt.

Die Ansätze, welche inkrementell das PM aus dem Quellcode extrahieren und sie konsistent halten, sind auf einen Überwachungsansatz angewiesen, welcher sie mit den benötigten Informationen versorgt. Vorhandene Ansätze können entweder bei der Überwachung keine feinkörnigen Informationen liefern oder die Instrumentierung des Quellcodes muss manuelle vorgenommen werden, was den Entwicklern nicht zumutbar ist.

Um diese Probleme zu überwinden, stellt diese Thesis einen Ansatz zur adaptiven Systemüberwachung vor, welches PMen feinkörnige Überwachungsinformationen zur Verfügung stellt. Zudem ist unser Ansatz Teil der kontinuierlichen Integration von PMen in den agilen Entwicklungsprozess. Das haben wir durch die Einführung eines automatischen adaptiven Instrumentierungsprozesses des Quellcodes erreicht. Außerdem kann unser Instrumentierungsansatz Quellcodeänderungen während der Systementwicklung überwachen und die veränderten Teile des Quellcodes entsprechend automatisch instrumentieren. Während der Systementwicklung sammeln wir automatisch die Überwachungsproben, welche zur Instrumentierung des Quellcodes benötigt werden. Wenn Änderungen im Quellcode vorgenommen werden, instrumentieren wir den Quellcode automatisch basierend auf den Überwachungsproben, welche wir während der Systementwicklung generiert haben. Der Vorteil unseres Ansatzes wird die Reduzierung des hohen Überwachungsaufwandes und die Unterstützung der kontinuierlichen Integration der PMe sein.

Insbesondere kann unser Ansatz mit anderen Ansätzen, welche den Quellcode und das PM konsistent halten, zusammengeschlossen werden. Während jene Ansätze tendenziell den Quellcode und ihr Verhalten bezüglich des PMs konsistent halten, wird unser Ansatz die notwendigen Überwachungsdaten liefern, um die PM Parameter zu schätzen (wie Ressourcenbeanspruchung, Anzahl von Schleifenausführungen, Wahrscheinlichkeit einer Programmierverzweigung).

Wir haben unseren Ansatz anhand einer Fallstudie evaluiert. Wir haben gezeigt, dass wir den Quellcode inkrementell und automatisch instrumentieren konnten. Außerdem hat unsere Evaluierung gezeigt, dass wir Überwachungsinformation zur Verfügung

stellen konnten, welche für die Kalibrierung der PM Parameter benutzt werden können, unter Berücksichtigung von parametrischen Abhängigkeiten. Weiterhin haben wir mit unserem Ansatz und der Fallstudie gezeigt, dass wir den Überwachungsaufwand auf 50% reduzieren konnten.

Contents

List of Figures

1 Introduction

Software performance is a quality attribute, which determines whether a given system has the ability to perform the functional requirements in terms of acceptable scalability and responsiveness. Many applications crash or can not be applied because the performance problems have been not taken into account during development.

In order to avoid performance issues, performance attributes must be captured and treated during the system development. Performance tests can be used to decide if the system meet its requirements. However, using it in a huge application can be too expensive because we need to allocate resources and generate measurements of an application, which creates a massive load.

Performance tests can be altered by performance model prediction, which can reduce the costs of performance tests. A performance model represents an abstraction of the behavior as well as the performance-relevant aspects of the system. Moreover, performance model prediction solves performance problems by using performance models and simulation engines. Performance simulation requires only a fraction of resources to evaluate the performance of a system. Another advantage of model-based performance prediction is the capability of identifying performance issues before implementation, which then helps to correct design problems as early as possible.

Model-based performance prediction can be used in modern agile development process to support design decisions and reduce the costs of measurement-based performance evaluation. However, Existing approaches have two shortcomings. (1) they do not extract incrementally the performance model, which means if they will be used iteratively, they will extract the performance model of the whole system in each iteration. This can guide to a high monitoring overhead in case of huge systems. Moreover, extracting the performance model in a such way will not keep the old modifications of the performance in each iteration. (2) the existing techniques do not come up with performance model parameters (like Resource Demands, loop execution number, the probability of selecting a branch) and their dependencies with impacting factors like input data.

There exist approaches that extract incrementally the performance model from the source code and keep them consistent. For example, Langhammer introduced an approach [18] that keep the source code and architecture consistent include the performance model in term of Service Effect Specification (SEFF) 2.1.2. Furthermore, Mazkatli and Koziolek have extended his approach by introducing an approach that uses the Continuous Integration (CI) and Continuous Deployment of the source code and extend them with the Continuous Integration of a Performance Model (CIMP) [20]. Both presented approaches require an adaptive monitoring approach that provide them with the required monitoring information and can be integrated in the continuous integration of the performance model.

In this thesis, we present an approach for adaptive monitoring for continuous performance model integration. Our approach can be used incrementally to provide monitoring information for continuous performance model integration. In particular, we used the performance model of the Palladio Component Model (PCM)(Section 2.1), which describes the behavior and the performance aspects of the system in term of SEFF. Furthermore, our approach can provide the monitoring information that can be used to estimate the SEFF model parameters. We provide monitoring information of computation response time, loop execution number, the probability of selecting a branch and the parameters of service calls, which can be used to consider the parametric dependencies in SEFF model parameters estimation.

In order to achieve an incremental instrumentation of the source code, our approach uses the change-driven framework Vitruvius (Section 2.2) and we extended the Coevolution approach [18] of Langhammer. Vitruvius provides the possibility to keep model instances consistent via consistency preservation rules can must be defined by the user. Moreover, Langhammer used Vitruvius to generate incrementally SEFF models, but he did not provide an adaptive monitoring approach to provide SEFF Models with the required monitoring information. Moreover, we contributed in Vitruvius by presenting an Instrumentation Model (IM) (Section 5.2), which describes the instrumentation points, which are needed for the source code instrumentation. Then, we defined the consistency preservation rules that keep the source code model and IM consistent. Using Vitruvius, we could in each iteration capture changes in the source code and create accordingly the instrumentation points. Therefore, we could instrument the source code incrementally.

Our approach is part of the vision presented by Mazkatli and Koziolek [20]. They extended the agile development process by introducing an approach that automatically and continuously integrate the performance model.

In chapter 2 we explain the foundation of our approach. In chapter 3 we define our contributions. In chapter 4 we describe our approach. In chapter 5 we a case study to evaluate out approach. In chapter 5 we discuss the related work. Chapter 7 gives a summary of this thesis and feature work.

2 Foundations

In this section, we introduce the necessary foundations for this thesis. In section 2.1, we present the Palladio Component Model (PCM) and it's models. In section 2.1.2, we describe briefly the Service Effect Specification (SEFF) model of PCM. In section 2.2, we present the Vitruvius Framework. In section 2.3, we present Java Model Parser and Printer (JaMoPP). In section 2.4, we introduce the Coevolution approach [19]. In section 2.5, we present the Kieker Monitoring Framework. In section 2.6, we give an overview of the reverse-engineering of the approach Source Code Model eXtractor (SoMoX). In section 2.8, we bring in the approach Continuous Integration of Performance Model (CIPM). In section 2.8.1, we summarize the master thesis of Jägers. In the last section 2.7, we introduce the development process DevOps.

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 the 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, predicts response time and 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 relationship 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 the 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. Afterward, 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 [14].

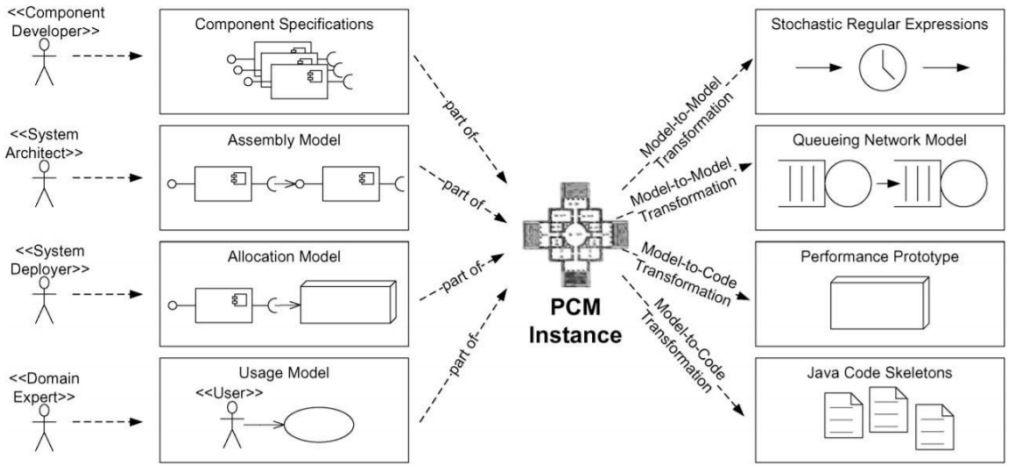

Figure 2.1: PCM Models and transformations [21]

2.1.2 SEFFs

Service Effect Specification (SEFF) were firstly presented by [14], 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 with 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 does 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() represents an internal computation and does not make any external call, it is seen 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 a loop, which calls the external service service2(), that's why it's represented as a LoopAction within the corresponding SEFF.

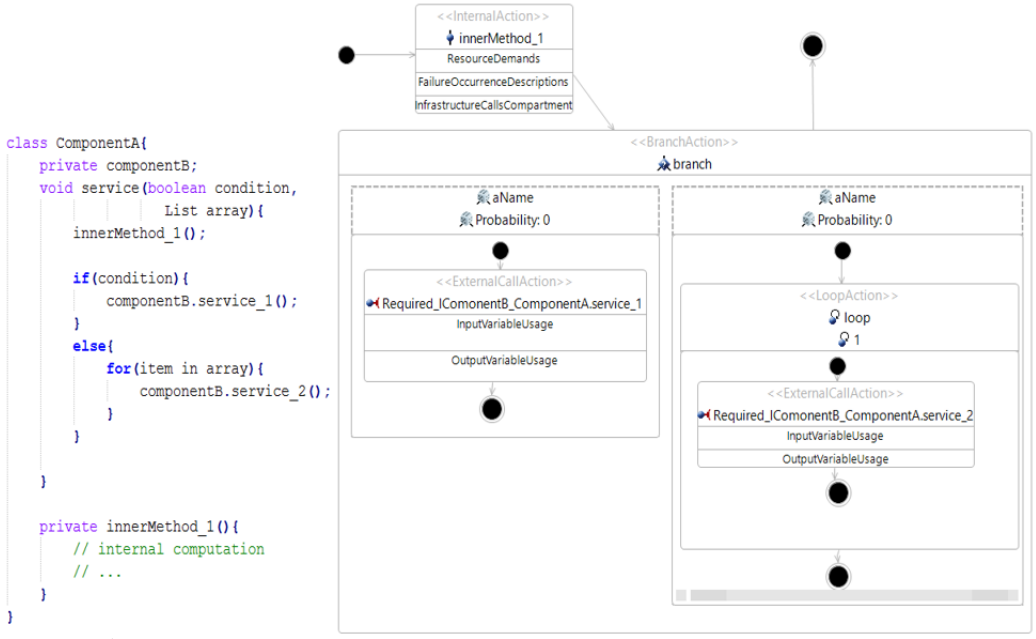

Figure 2.2: Example of source code and SEFF

2.2 Vitruvius

The Vitruvius approach is a view-based [6] engineering approach, which was introduced in [5, 15]. Vitruvius can be used to keep the models of a system consistent. In Model Driven Engineering (MDE) [1], 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 into 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 models, 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 they are needed.

2.2.1 Vitruvius VSUM

In order to keep models instances consistent, Vitruvius approach uses the so-called Virtual Single Underlying Model (VSUM) which contains all information that represents the system Figure 2.3. The models in VSUM are accessible using views. Views are instances of view types and they can be used to manipulate model instances. Moreover, there are two kinds of view types, namely projectional view types or combining view types [5]. Projectional view types allow to show information from one meta-model solely. Combining view types can be used to show information from diverse meta-models.

Models consistency in Vitruvius can be preserved using Consistency preservation rules. They describe how changes in one model should be transferred into changes in another model. Moreover, the Consistency Preservation Process uses these rules to create the model's transformation that preserves the consistency.

2.2.2 Correspondence Meta-model

The Correspondence Metamodel (CMM) is used within Vitruvius in order to describe the corresponding elements of two meta-models. Moreover, Vitruvius one instance of CMM can be used for each Vitruvius application. A Vitruvius application can have one or many meta-models instances. In our approach, we use one CMM instance which has been created and used in the Coevolution approach. Furthermore, for the process of probes generation, we will use the existing information in this instance that have been saved the Coevolution approach and add new information to it by extending the Coevolution approach.

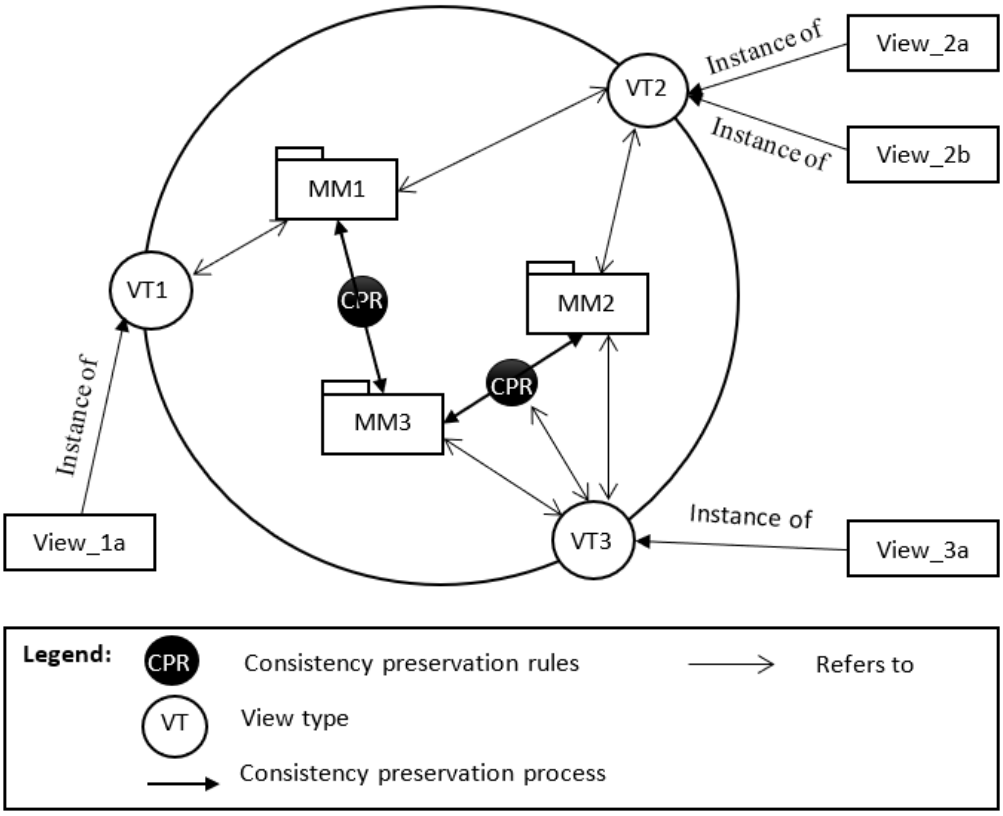

Figure 2.3: Overview of the Vitruvius VSUM and how model instances can be kept consistent

Figure 2.4 shows the Vitruvius Correspondence Meta-model. It's basically composed of two classes. The root class Correspondences contains the list of Correspondence. The class Correspondence is composed of two lists of identifier references. The first list contains identifiers that reference models in one meta-model, while the second list contains identifiers that reference models in the other meta-model.

CMM is generic and can be used for diverse meta-models. Therefore, in order to identify an element, the reference in the Correspondence has to be unique because only one concrete element needs to be identified for a given ID. For this purpose, the CMM uses the so-called Temporarily Unique Identifier (TUID) mechanism. TUID is a string that can identify an element. It has to be calculated based on the properties of the element that it represents. The TUID is also used when an element needs to be retrieved from the correspondence model.

2.3 Java Model Parser and printer

Java Model Parser and Printer (JaMoPP) [7] 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 [23]. JaMoPP parser allows to parse a Java source

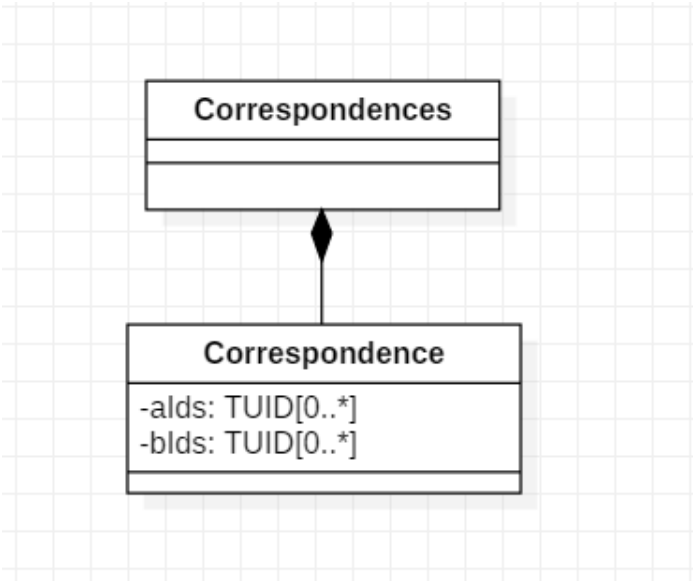

Figure 2.4: The Correspondence Meta-model of the Vitruvius Framework

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 [8], which allows defining text syntax for languages described by an Ecore meta-model.

2.4 Automated Coevolution of Source Code and Software Architecture Models

The approach of co-evolution of source code and component-based software architecture presented in [19, 18] 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 the 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 a model to model transformations.

The co-evolution approach uses change-driven engineering in order to react 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 another 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 2.5 shows the steps, that are used to keep architecture models and the source code consistent:

- Step (0): users change either the PCM repository, the PCM system or the Java source code.
- Step (1): monitors capture changes on the models.
- Step (2): monitors trigger Vitruvius Framework and pass it the changes.
- Step (3): based on these changes, the Vitruvius framework executes the consistency preservation transformations.
- Step (4): the transformations use information from the changes and the correspondence model to execute the preservation rules.
- 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.

2.4.1 Incremental SEFF Reconstruction

Langhammer has proposed in his Co-evolution approach an incremental SEFF reconstruction approach Figure 2.6 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 into the Coevolution approach that means it can use functionalities and information provides by Vitruvius. Moreover, the incremental SEFF reconstruction is done in a change-driven way, that means to 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 [18].

2.5 Kieker Monitoring Framework

Kieker is an extensible Java-based application performance monitoring and dynamic software analysis framework [9]. Figure 2.7 shows the architecture of Kieker framework. It's composed of 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 serializes 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) [11]. 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 a 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 manual instrumentation, which consists of mixing the instrumentation logic with the business logic of the application. Figure 2.1 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) [12], which helps to separates the instrumentation logic from the business logic. Kieker defines AOP based monitoring probes like OperationExecutionAspectAnnotation and OperationExecutionAspectAnnotationServlet. Figure 2.2 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 the box.

```
1  final long tin = MONITORING_CONTROLLER.getTimeSource()
       .getTime();
2  // call of the service
3  this.serviceA();
4  final long tout = MONITORING_CONTROLLER.getTimeSource
       ().getTime();
5   // Create a new record and set values
6  final MyResponseTimeRecord e = new
       MyResponseTimeRecord(
7  "serviceA", tout, tin);
8  // Pass the record to the monitoring controller
9  MONITORING_CONTROLLER.newMonitoringRecord(e);
```

Listing 2.1: Example of manual instrumentation of source code

```
1  @OperationExecutionMonitoringProbe
2  public void serviceA(){
3     /*
4      method content
5     */
6  }
```

Listing 2.2: Example of a Kieker Probe using AOP

2.6 Source Code Model eXtractor

Source Code Model eXtractor (SoMoX) is a reverse engineering approach, which has been developed by Krogmann [16]. 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 main 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:

- Parse the source code into a model.
- Detect components and interfaces using metrics.
- Detect data types and signatures using metrics.
- 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 behavior of the source code is done by analyzing the methods of the source code. This step has been extended by Langhammer [18] and its results are used in the thesis. In the following, we will explain briefly how the reconstruction of SEFFs is done within SoMoX and how it was extended by Langhammer.

2.6.1 Source Code Decorator Model

The Source Code Decorator Model (SCDM) has been presented within the SoMoX approach (Section 2.6). SoMoX reverse-engineers the source code and can extract the architecture models from it. Moreover, SoMoX can be used to extract the Palladio Component Model (PCM) from the Java Source Code. The SCDM is used to create trace links at the model level. The linking between the source code elements and the architecture model elements are needed in the reverse-engineering process. Therefore, the SCDM creates links between the source code elements and the PCM elements. Furthermore, due to the use of SCDM, the reverse-engineering process can be done without mixing the linking concerns with the domain specific language of the source code model and the architecture model.

The SCDM was also used in the Coevolution approach (Section 2.4) which keeps automatically the Java Source Code and the PCM models consistent during the system development. The SCDM was used to contain the information, which source code element is reverse-engineered into which architectural model element. For example, it contains the information, which classes are mapped into which component.

In our approach, we will need the SCDM in order to map between the SEFF elements and the source code elements or the statements. As mentioned before,

the instrumentation points in our approach are represented by the SEFF elements. Moreover, in order to insert the instrumentation code in the right location in the source code of the system, we need to know which SEFF elements correspond to which source code statements. The SCDM offers this possibility by mapping between each SEFF element and the corresponding JaMoPP statements.

In order to map between the SEFF elements and the JaMoPP statements, we do not need to implement this feature because it's done already in the Incremental SEFF Reconstruction Process (section 2.4.1) in the Coevolution approach. This process can monitor the changes within the source code and reconstructs incrementally the SEFF models. Moreover, the execution of this process requires the execution of the SoMoX in order to reverse-engineer the changed parts of the source code. The execution of SoMoX creates the SCDM which maps between the SEFF elements (like Branch Actions, Internal Actions, etc.) and the JaMoPP statements Figure 2.8.

2.6.2 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 maps 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 analyzing 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 visits 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 DevOps

DevOps [4] 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.

2.8 Continuous Integration of Performance Model

Continuous Integration of Performance Model (CIPM) is an approach proposed by Mazkatli and Koziolek [20] 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 continuous integration of the performance model.

To achieve that Mazkatli and Koziolek used the Coevolution approach developed by Langhammer (Section 2.4) 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 2.9. 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 [20].

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 activity two. The structure of the performance model is updated based on the work of Langhammer [19]. 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) [12]. 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 VSUM. 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.

2.8.1 Iterative Performance Model Parameter Estimation Considering Parametric Dependencies

Jägers has created his master thesis [10] 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 Koziolek (Section 2.8), it extends precisely the concepts introduced in the activities three and four in Figure 2.9. 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 to 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 the performance model. These monitoring data can be used by the approach of Jan to make the estimations.

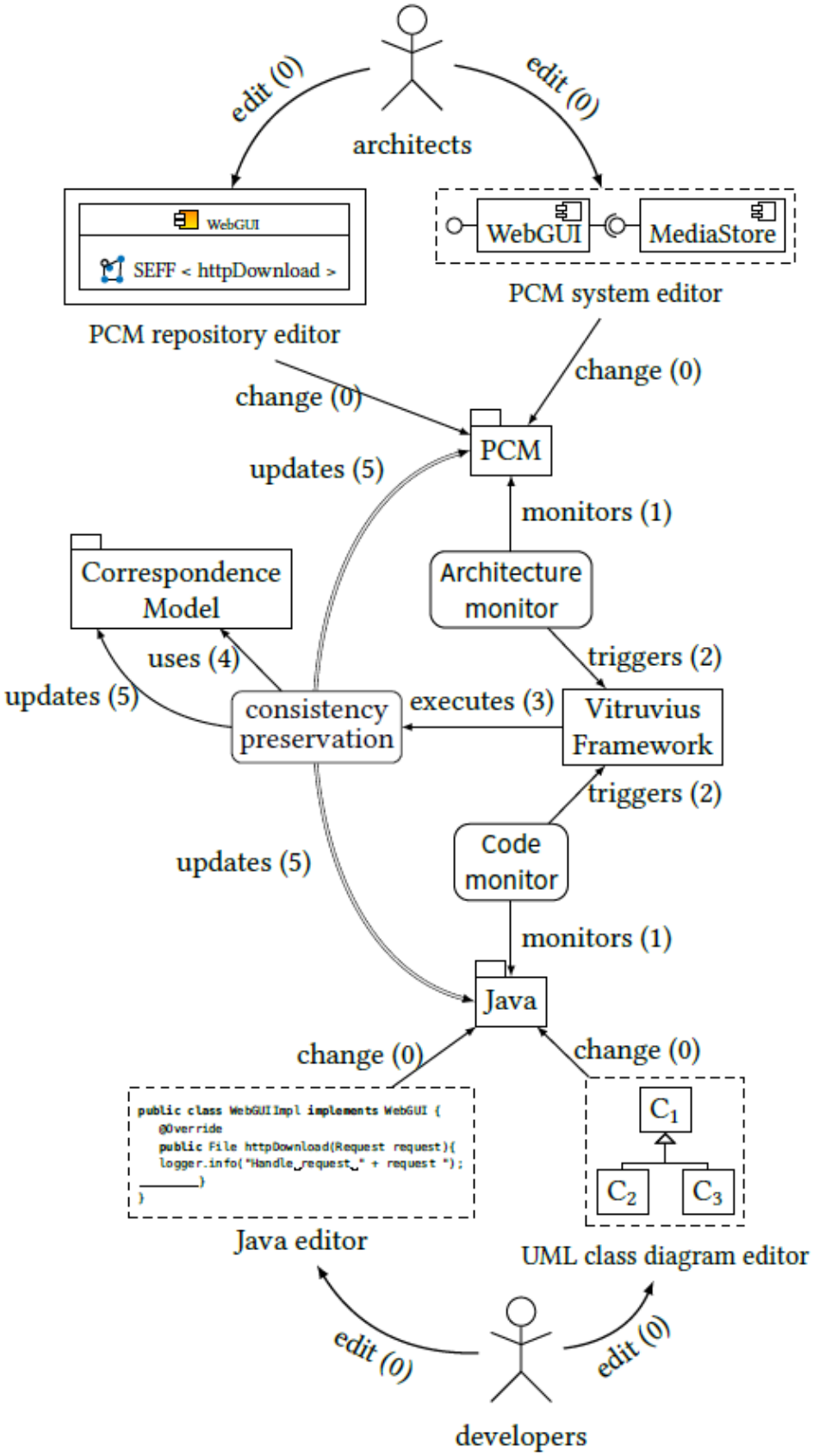

Figure 2.5: The steps used in co-evolution approach to keep architecture models and the source code consistent [19]

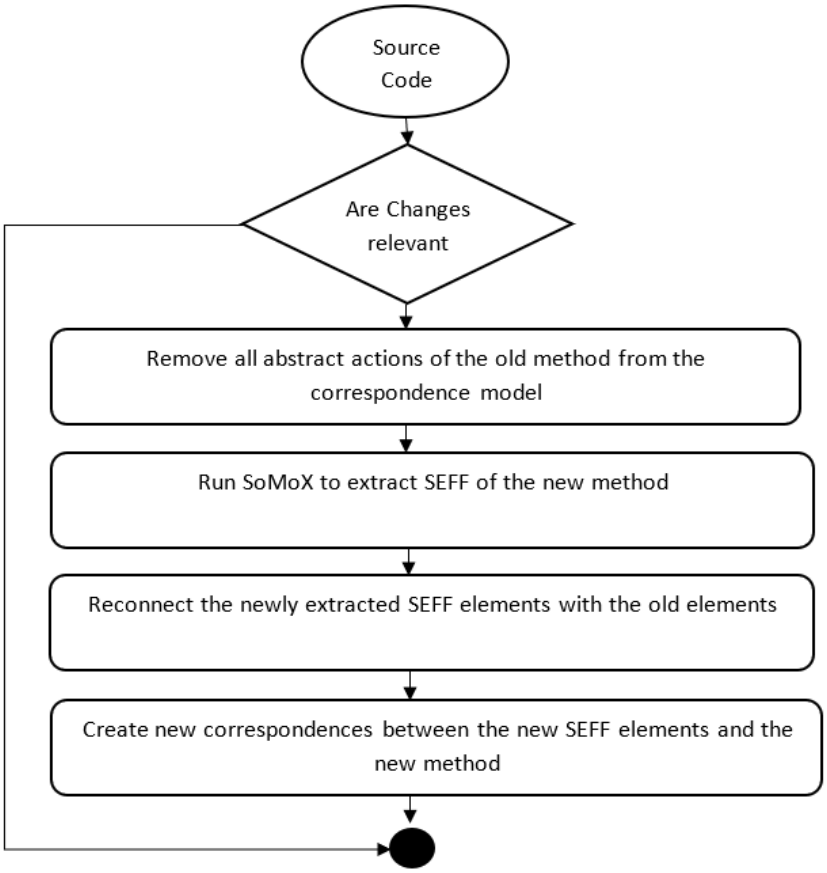

Figure 2.6: Overview of the incremental reconstruction process of SEFF

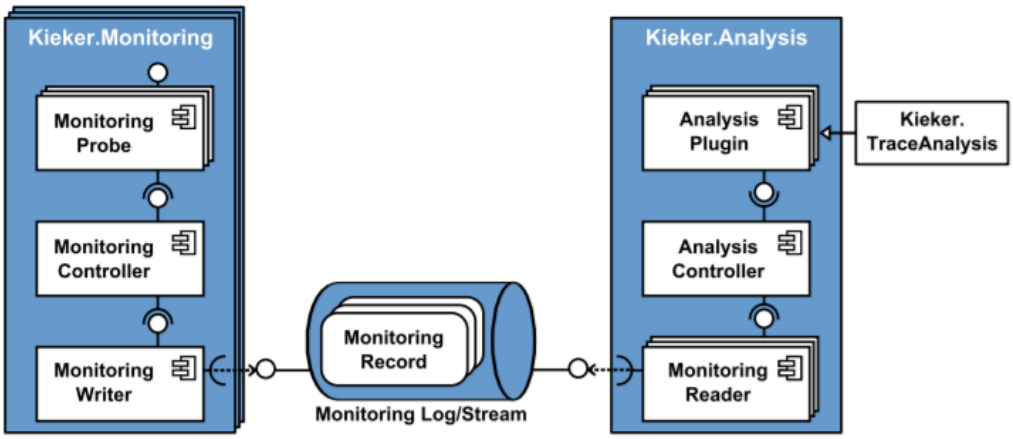

Figure 2.7: Overview of the Kieker's architecture [9]

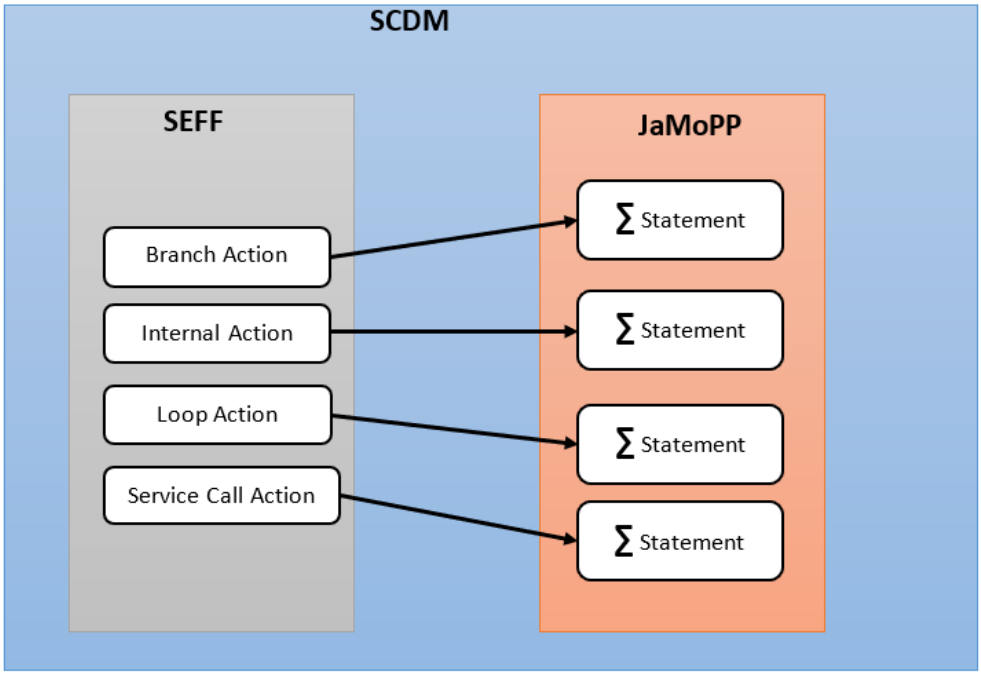

Figure 2.8: Source Code Decorator Model (SCDM) maps between the SEFF elements and the corresponding JaMoPP statements

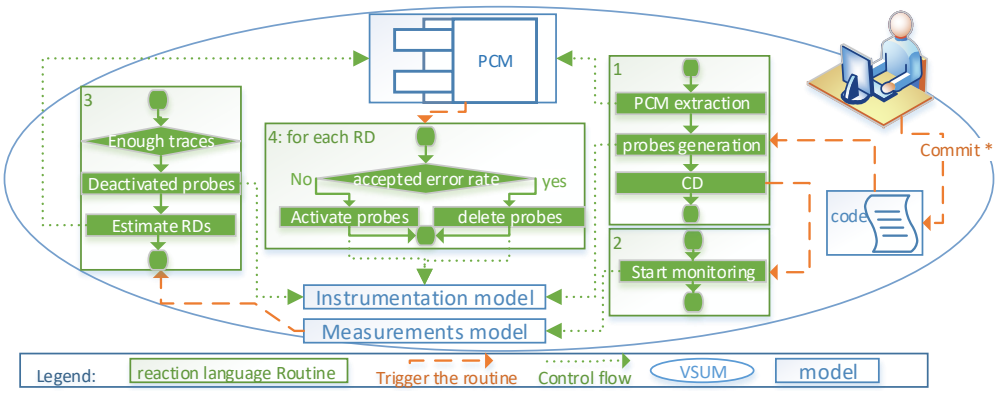

Figure 2.9: CIMP activities [20]

3 Thesis Statement

In this section, we will introduce our contribution to the subject of adaptive monitoring for performance model integration as well as the involved scientific challenges that must be solved in order to achieve the goals of our contribution.

3.1 Contribution

The contribution of this thesis is an extension of the contribution presented in the CIPM approach [20]. Precisely, we focus on our contribution on the first and the second activity of the CIPM approach.

In order to contribute to the subject of adaptive monitoring for performance model integration, we've created an approach that takes into account two properties. The first property is the adaptive monitoring which means that the probes which are responsible for logging the monitoring data are capable to be activated and deactivated during the monitoring phase. The meaning is that once we have collected enough monitoring data of a probe, the probe must be deactivated because more monitoring data is not necessary and can influence the performance of the monitoring. Furthermore, we adopted our approach to fine-grained monitoring. This is required, when the used performance model needs performance model parameters (like Resource Demands RDs, loop execution number, the probability of selecting a branch). Therefore, we adapted our approach to monitor the response time of computation, the number of execution of loops, the executions of branches and the parameters of a method calls.

The second property is the monitoring for continuous integration of performance model. The continuous integration of performance model in this context means the incremental update of the performance model in each iteration. The incremental update of the performance model includes the possibility to keep first of all the status of the performance model in the previous iteration and then update it with the information provided within the current iteration. The opposite of that will be to extract the whole performance model in each iteration which will lead to two issues. The first issue is that the extracted performance model in an iteration will not be saved in the next iteration which will cause the second issue. Since the iterations do not consider the modification done in the previous ones, the whole system will have to monitored in every iteration in order to provide the required monitoring data for the new extracted performance model. Hence, in the case of huge Enterprise Applications (EAs) monitoring the whole system in every iteration will cause a monitoring overhead

which is unfeasible and that is the second issue.

To guarantee the monitoring for performance model integration, we've made the following decisions:

- We used the Palladio Performance Model as a performance model.
- We used Vitruvius to support the model-driven development in our approach.
- We used the Coevolution approach for the continuous creation of performance model.
- We created an instrumentation model to persist the generated probes.

3.2 Scientific challenges

In this section, we will identify the scientific challenges that we will resolve in order to achieve the goals of this thesis.

- *what are the steps necessary to generate the probes and instrument automatically the source code for monitoring?*
  In order to monitor the source code of a system, the system must be firstly instrumented using the needed probes. The instrumentation in our approach is done automatically based on the generated probes during the system development. Moreover, the processes of probes generation and the source code instrumentation can be executed independently. Therefore, we need to define the necessary steps to execute these processes and how they change information between them.

- *what are the probes that must be generated for the monitoring?*
  The definition of probes for monitoring is based on the monitoring data needed by the performance model. Since we used Palladio Performance Model, we have to define probes that provide monitoring data that can be used by the Palladio Performance Model.

- *What are the models required to support the iterative and adaptive monitoring?*
  In order to enable the iterative and the adaptive monitoring for continuous performance model integration. As mentioned before, we extend the coevolution approach which uses models like PCM, Java and Correspondence Model. Therefore, we need to use the information provided by the existing models and define new models for persisting the information that is not provided by the existing model.

- *What are the alternatives of the source code instrumentation for monitoring?*
  The best way to instrument the source code of a system is to use Aspect-oriented Programming (AOP) because it separates the instrumentation logic from the business logic which simplifies the application maintenance. However, AOP

is based on annotations which cannot be used for all kind of probes. For example, we can not use AOP annotation to monitor the number of executions of a loop. Therefore, we need to find alternatives that enable the source code instrumentation without changing the original source code of the system. Otherwise, mixing the original source code with instrumentation code will make the readability of the source code infeasible for the user.

- *What level of incremented update of performance model can we reach?*
  Here, we define two levels of incrementation. In the first level, the performance model will be updated based on the changed services. In the second level, it will be updated based on the changed parts of the service. The difference between the first and the second level is that on the first level if a service has changed the SEFF of the service will newly recreate. Therefore, the old elements of this SEFF will be lost with their parameters. In the second level, the unchanged old elements in SEFF will be kept, the new elements will be added and the deleted elements will be deleted. The Coevolution approach has reached the first level. Thus, we will extend to reach the second level of incrementation.

4 Adaptive Monitoring for Continuous Performance Model Integration

In this chapter, we will introduce our approach for adaptive monitoring for continuous performance model integration. In section **??**, we bring in the context of our approach. In section 4.2.1, we define the monitoring probes that we need for our monitoring approach. In section 4.2.2, we define the monitoring records that we monitor. In section 5.2, we clarify the adaptive instrumentation concept in our approach. In section 4.5, we define an instrumentation model for saving the monitoring probes. In section 4.9.5, we give an overview of our approach and its activities. In section 4.6, we introduce the process that we used to generate iteratively the monitoring probes. In section 4.7, we present our approach for adaptive instrumentation of the source code.

4.1 Context of our Approach

As mentioned before, our approach is part of the CIPM vision (Section 2.8) which extends the agile and DevOps process and provides them with iterative and incremental Performance Model. Moreover, the Performance Models in CIPM are enriched with Performance Model Parameters. In the following, we will briefly depict and describe the process and the context in which our approach takes place.

Figure 4.1 shows the process in which our approach takes place. this process is based on Vitruvius (Section 2.2) which means its elements are either models or transformations. The Java code in 4.1 is represented by a JaMoPP model. When the developer commits changes on this model, two transformations will be triggered. The first one is the Coevolution process of Langhammer (Section 2.4) which keeps the Source Code and the models in the Palladio Component Model consistent, mainly the repository and the SEFF Model. The second Transformation is our Transformation which is specified for keeping the Source Code and the Instrumentation Model consistent. We proposed the Instrumentation Model in order to persist the Probes that will be required for instrumenting the Source Code.

When the system under development is deployed, our Instrumentation Process will be triggered. This process receives as inputs the Probes from the Instrumentation Model and the Source Code of the System. It delivers afterward the instrumented Source Code as a JaMoPP Model. After the instrumentation process has been finished, the instrumented Source code will be executed and monitored. The information provided by monitoring is encapsulated in a Measurement Model which describes the

needed monitoring records. After the monitoring has been finished the Parameters Estimation Process (Section 2.8.1) can be triggered. it uses the information in the Measurement Model to estimate the Performance Model Parameters and updates accordingly the SEFF Model in the Palladio Component Model. Afterward, the user can use the updated Performance Model to simulate and evaluate the performance of the system.

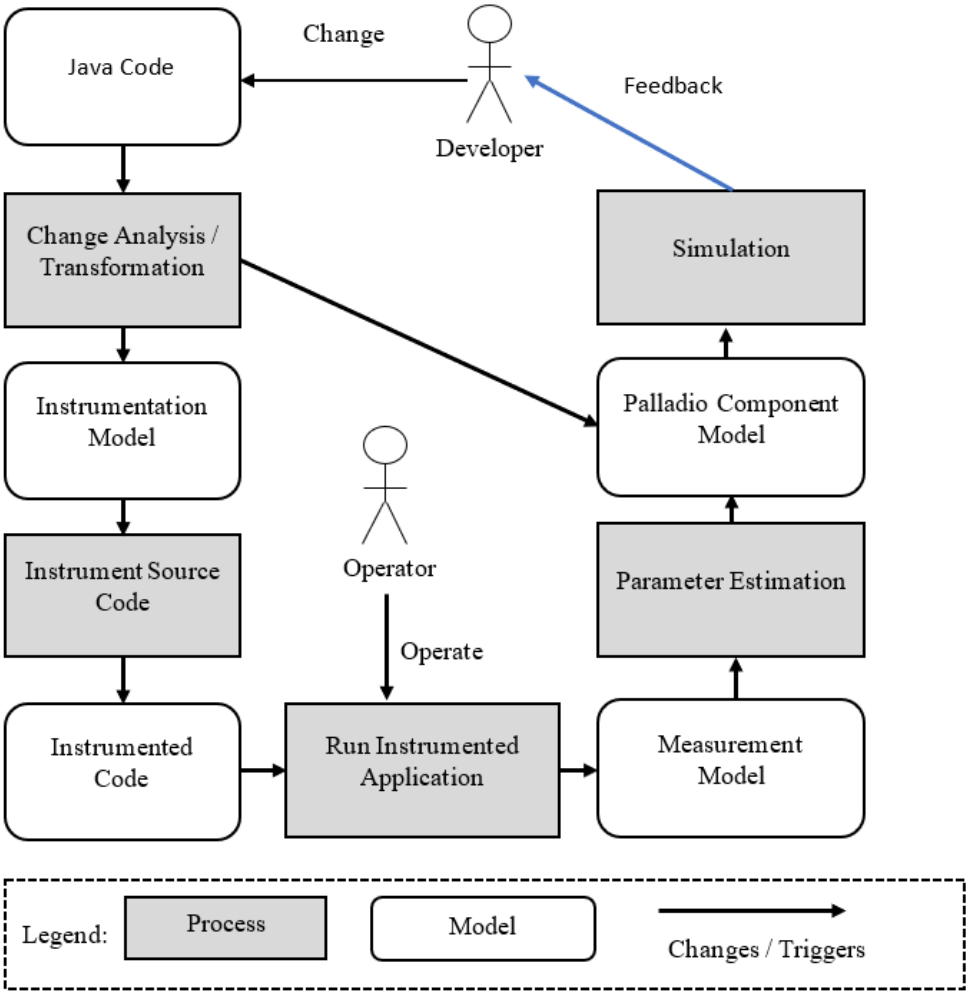

Figure 4.1: Context of our approach

4.2 Terminology

In this section, we introduce the terminologies we used in our approach. In section 4.2.1, we define the probes used in our approach. In section 4.2.2, we define the monitoring records that we should generate during system operation. In section 5.2, we give an overview of the concept of adaptive instrumentation in our approach. In section 5.3, we define the adaptive monitoring in the context of this thesis.

4.2.1 Monitoring Probes

Monitoring Probes are responsible for collecting monitoring information from the system. Furthermore, they can be specified based on the needed monitoring information. For example, if we need to monitor the response time of a service and the number of execution of loops, we can specify two monitoring probes, one probe for the response time and the other for loops execution number.

In our approach, we want to provide monitoring information for Palladio Performance Model which are described in terms of SEFF (Section 2.1.2). SEFF Model is composed of four main elements which inherit from the so-called Abstract Action, namely Internal Action, Branch Action, Loop Action, and Service Call Action. In other words, we should specify probes that provide these elements with the needed monitoring information. Therefore, we defined a monitoring probe for each SEFF element Figure 4.2. The monitoring information that we need to produce for SEFF elements is described in (Section 4.2.2) under monitoring records.

For the monitoring purpose, we used the Kieker Monitoring Framework (Section 2.5) which offers the possibility to define new monitoring probes based on the needed monitoring records.

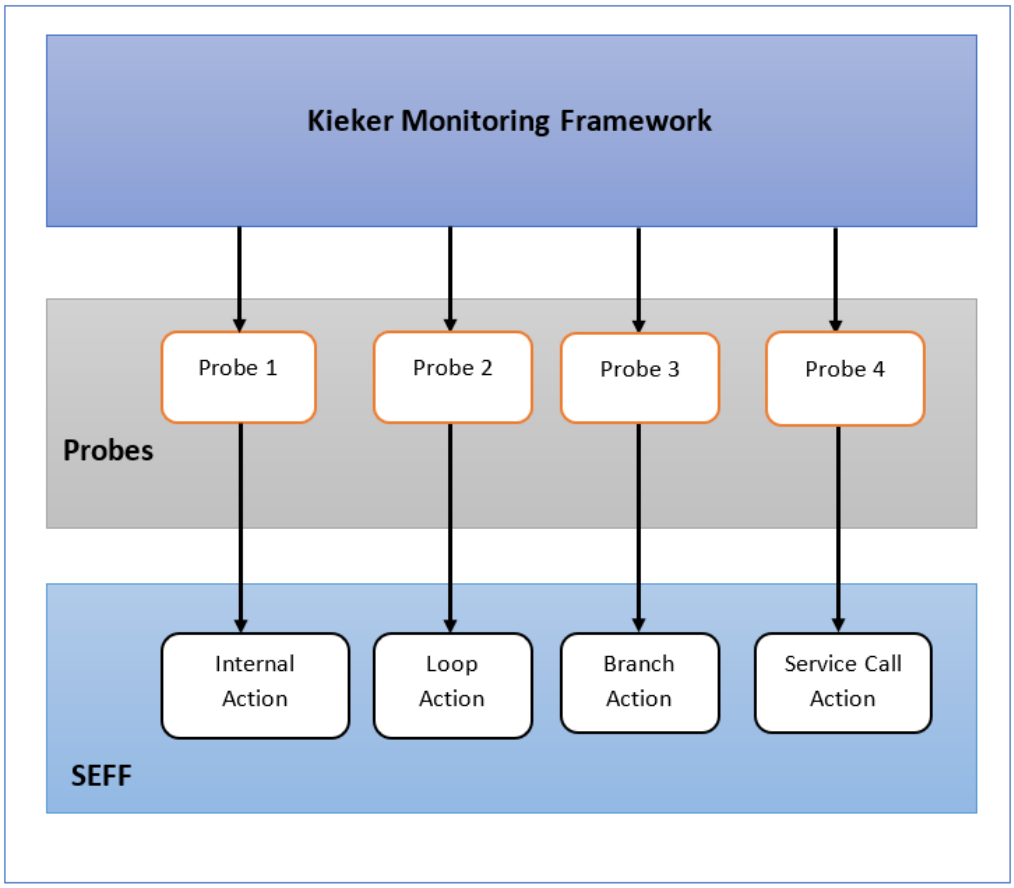

Figure 4.2: Specified monitoring probes in our approach

4.2.2 Monitoring Records

In order to create monitoring probes (Section 4.2.1), one must define first of all the needed monitoring information. in our case, we defined monitoring probes for SEFF model which are based on the monitoring records in Figure 4.3.

Figure 4.3 shows a UML Class Diagram that describe the monitoring records. It provides extra information that is needed for the purpose of performance model parameters estimations. The abstract class RecordWithSession specifies a sessionID attribute that helps to identify the monitoring information based on sessions. The abstract class ServiceContextRecord adds a serviceExecutionID attribute that helps to reference a ServiceCallRecord.

All records are identified via their ids. The id of each record can be provided by the used monitoring framework. However, in order to be able to use these records for SEFF models, we've used for each record the id of the corresponding SEFF element.

For internal actions which express an internal computation, we want to know the time consumed by them. Therefore, we need to log the id of the internal action, start time and stop time.

For Loops, we need to recognize the number of executions of a loop. The same thing for branches, we log the id of the executed branch in order to realize if the branch was executed or not.

As to service calls, we need to locate them in which service is called, we need also to provide the response time and the parameters they are given. The parameters of services are given in a JSON format.

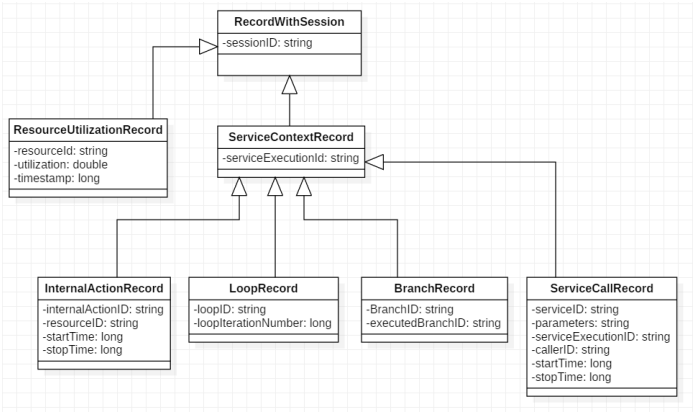

Figure 4.3: UML class diagram that shows the monitoring information required by SEFF models

4.2.3 Adaptive Instrumentation

In this section, we will introduce the meaning of Adaptive Instrumentation in our approach as well as the main concepts used in order to achieve that.

As mentioned before, our approach is addressed to iterative development processes like DevOps. In this context, Adaptive Instrumentation means that only the parts of the source code that have been changed in the current iteration will be monitored.

Based on this definition, we should be able to generate instrumentation points during the development phase. In order to do this, we decided to use two main concepts which are model-driven engineering and change-driven engineering. Moreover, our approach is based on the Coevolution approach which itself uses these two concepts to keep the source code and the architecture models consistent. Precisely, the Coevolution approach uses change-driven consistency preservation in order to keep the Java source code and SEFF models consistent. Changes in the source code model are monitored and transformed to changes in the SEFF model based on defined consistency-preservation rules.

In order to achieve adaptive instrumentation in our approach, we've defined and Instrumentation Model which contains the instrumentation points. Moreover, we defined a transformation that monitored changes in the source code model and creates the corresponding instrumentation points in the instrumentation model.

In order to keep the source code and the instrumentation model consistent, we've used the Vitruvius Framework which makes it possible to keep models instances consistent based on models changes.

4.2.4 Adaptive Monitoring

Adaptive Monitoring means that the monitoring probes can be activated and deactivated based on the existing monitoring information. this is needed when we've collected enough monitoring information for some probes but they still can log monitoring information which is not needed and which can lead to monitoring and performance model parameters estimations overhead. Therefore, adaptive monitoring can help to reduce the monitoring overhead by reducing the number of monitoring probes.

In order to achieve Adaptive Monitoring in our approach, we've added an attribute for the monitoring probes that defines their activeness. That means, monitoring probes are checked during the monitoring phase and they can log monitoring information only if they are activated. The deactivation of the probes can be done for example, when we've realized that the current monitoring information for these probes is enough for the performance model parameters estimation.

4.3 Approach

In this section, we will briefly introduce our approach as well as the context in which our activities will be executed. Further details on these activities will be presented in the next sections.

our approach is developed in the context of the iterative development process DevOps. Therefore, Figure 4.4 gives an overview of the activities of our approach and in which DevOps phase they can be executed. The green color indicates processes or model in which our algorithms are executed.

In our approach, we've defined to main processes. The first one is responsible for collecting the instrumentation points or the probes. It must be executed during the development phase. It uses information from Vitruvius and it's based on the Coevolution approach. The Coevolution approach keeps the source code and the SEFF model consistent and provides us with information that helps to gather the probes. Vitruvius is used to keep the source code model and the Instrumentation Model consistent. The generated probes are saved and managed in the Instrumentation Model. For more details on the probes generation process, look at the section 4.6.

The second process is the instrumentation process which can be executed at any time. Once it's executed, it takes the probes from the Instrumentation Model and insert the instrumentation source code in the source code of the system based on the types of the probes (Section 4.2.1). However, if the monitoring will be firstly done in the monitoring phase of DevOps, this process can be automatically triggered at the Continuous Deployment phase. For more details on our instrumentation process look at Section 4.7.

In the monitoring phase, the instrumented system can be executed in order to log monitoring information. Moreover, in order to achieve adaptive monitoring, the monitoring probes execute a self-checking for their activeness. Therefore, the monitoring code uses information from the Instrumentation Model in order to check if probes are activated or deactivated and thus if they can log or not.

4.4 The Vitruvius VSUM of our Approach

In this section, we will present the VSUM of our approach in order to generate the monitoring probes. We will precisely extend the VSUM Figure (Section 2.5) defined by the Coevolution approach.

As mentioned above, our approach extends the Coevolution approach, which keeps the source code and the architecture consistent. The VSUM in the Coevolution approach is composed of the following models:

- Palladio Component Model: it represents the architecture models.
- JaMoPP Model: it represents the source code of the system.
- Correspondence Model: it links between diverse model elements.

Moreover, the steps that are used to keep these models consistent are described in (Section 2.4).

In order to use Vitruvius for the adaptive monitoring purpose, we added an Instrumentation Meta-model (Section 4.5) that saves the monitoring probes and keeps them consistent with the source code. Figure 4.5 shows our VSUM which is extended from the VSUM of the Coevolution approach. The steps described in (Section 2.4) are kept the same. However, we added a transformation (Section 4.6.2) in the step (5) which keeps the Java source code and the instrumentation model consistent.

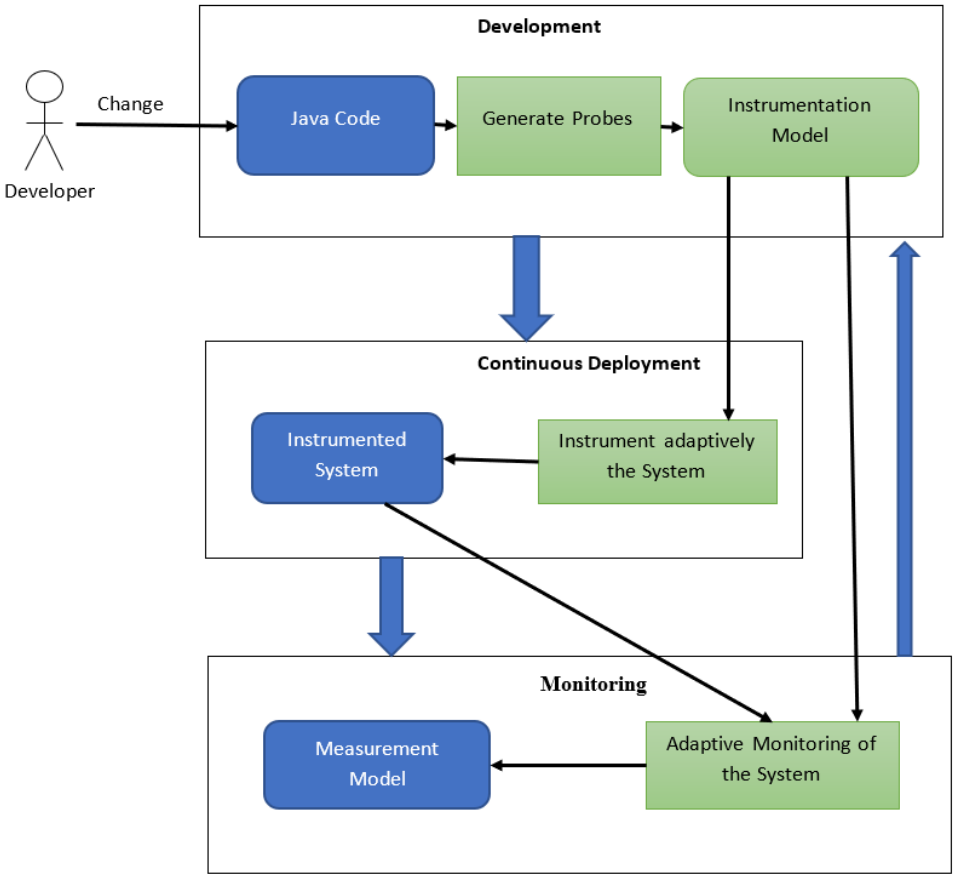

Figure 4.4: Overview of our approach activities in DevOps context

4.5 Instrumentation Meta-model

The Instrumentation Meta-model was originally presented in the approach of Manar and Koziolek [20] on which we based our approach.

The Instrumentation Meta-model(IMM) Figure 4.6 is one of the Contributions of this thesis in Vitruvius. IMM is responsible for describing and managing the instrumentation points. Moreover, we've defined IMM in order to achieve the Adaptive Instrumentation (Section 5.2) and Adaptive Monitoring (Section 5.3).

IM is composed of two elements, AppProbes which represents the model root and the element Probe which represents an instrumentation point. The element Probe corresponds to a SEFF abstract action which can be an Internal Action, a Branch Action, a Loop Action or a Service Call Action.

As mentioned before, we've used Vitruvius in order to keep the source code and the IMM consistent. We've also extended the Coevolution approach which uses also Vitruvius to keep the Java source code and the architecture models consistent. The Coevolution approach uses an instance of the SEFF model in which our probes are stored. Therefore, in order to avoid redundant information within Vitruvius, we've decided to store only the Ids of these probes which give us the possibility to return them

when they are needed. Moreover, since the probes represent concretely the four above mentioned abstract actions of SEFF, the id of a probe is named abstractActionID. Furthermore, we've added a boolean attribute to the monitoring probes in order to enable adaptive monitoring.

4.6 Probes Generation

In this section, we will introduce the process that we used to create the monitoring probes based on the Vitruvius Framework. This process is divided into two steps. The first step consists of collecting the information that is useful for our transformation that keeps the source code and the Instrumentation Model consistent. the second step is the execution of this transformation.

4.6.1 Collecting Information for the Transformation

In this section, we present the process used to collect information. this information is can be used in the Instrumentation Process and for executing the transformation that generates the monitoring probes and keeps the source code and the instrumentation model consistent.

In order to execute our transformation, we need to know the new or the updated probes (SEFF elements). For the Instrumentation Process, we need correspondence between the SEFF elements and corresponding JaMoPP statements. This information should be provided in each change of the source code.

In order to get this information, we used the information provided by the Incremental Reconstruction Process of SEFF (Section 2.4.1). This process reconstructs incrementally the SEFF of service if its source code has been changed. Moreover, the SEFF elements are linked with the corresponding services in the Correspondence Model and can be retrieved from our transformation.

The second information that we need is the correspondence between the SEFF elements and source code. this information can be also provided by the SEFF reconstruction process. As described in (Section 2.4.1), the SoMoX is executed to reverse-engineer the changes service and creates its SEFF. SoMoX creates also the Source Code Decorator Model (SCDM) (Section 2.6.1) which contains links to the SEFF elements of the service and the corresponding JaMoPP statements. Moreover, in order to make this information useful in our instrumentation process, we should make it accessible from outside of the SEFF reconstruction application. Therefore, we extended the SEFF reconstruction process with a new step that transforms the linking between the SEFF elements and the JaMoPP statements from SCDM into correspondences in the correspondence model Figure 4.7.

4.6.2 Keeping the Source Code and the Instrumentation Consistent

In this section, we present our transformation that keeps the source code and the Instrumentation Model (Section 4.5) consistent.

In order to generate the monitoring probes in our approach, we should preserve the consistency of the source code and the instrumentation model. Therefore, we created a transformation that transfers changes in the source code into changes in the instrumentation model. This transformation is executed when the source code has been changed.

In order to execute our transformation, we need to know the old service and its SEFF elements as well as the new service and its SEFF elements. This information can be obtained from the Correspondence Model after the Coevolution has been executed. Therefore, the Coevolution approach and our transformation are executed asynchronously. That means, we execute firstly the Coevolution approach in order to collect the information for our transformation then we execute our transformation based on this information.

The consistency preservation rules in our transformation are simple. When the source code has been changed and the changes are relevant, we delete the old probes of the old service from the instrumentation model and we add the new probes of the new service in the instrumentation model. Moreover, the newly added probes are by default activated (Section 4.2.1).

4.7 Adaptive Instrumentation Process

As we described in (Section 4.9.5), our approach is composed of two main activities. the first activity is responsible for collecting the monitoring probes. The second activity is the instrumentation process, which uses the collected monitoring probes from the first activity to instrument the source code. In this section, we define our Instrumentation Process for adaptive instrumentation of the source code.

4.7.1 Design

In this section, we give an overview from outside of our adaptive instrumentation process. In section 4.7.2, we define the inputs for this process. In section 4.7.3, we discuss the alternative Outputs of our process and prove our decision.

4.7.2 Inputs

Figure 4.8 shows an overview of the inputs and the output of our Instrumentation Process. It receives three Inputs. First of all, the source code of the system which we want to monitor. secondly, The Correspondence Model which contains correspondences between the services and their SEFFs as well as correspondences between the SEFF elements and the corresponding JaMoPP statements. Finally, the Instrumentation Model which contains the instrumentation points that we should use to instrument the source code.

4.7.3 Output

The output of the Instrumentation process is an instrumented version of the source code of the system. This instrumentation is based on the monitoring probes defined in (Section 4.2.1). Moreover, there are two ways to instrument the source code. The first alternative consists of using the Aspect-Oriented Programming which separate the instrumentation code from the source code of the system. The second alternative consists of mixing the instrumentation code with the source code of the system. Furthermore, we showed in (Section 2.5) that the first alternative cannot be used in our approach because of the fine-grained monitoring probes that we defined in our approach (Section 4.2.1). Therefore, we decided to use the second alternative in our instrumentation approach.

Alternative of the instrumentation process outputs
As mentioned above, in order to do the source code instrumentation, we decided to mix between the source code and the instrumentation logic. Here, we distinguish also between two alternatives for instrumenting the source code of the system. The first alternative consists of instrumenting the original source code of the system. The second alternative consists of making a copy of the original source code of the system and instrument it. We decided to use the second alternative in our instrumentation process.

Instrumentation of a copy of the source code
The reason why we decided to use the second alternative to instrument the source code was due to the readability of the original source code. Moreover, if we instrumented the original source code of the system, we will reduce its readability and make the maintenance more difficult in the feature for programmers. Figure 4.1 shows an instrumented version of the source code in Figure 2.2. As we can see, the readability of the source code has become difficult and the maintenance will be also hard to do. Therefore, we decided the instrument a copy of the original system in order to receive the monitoring information.

```
1  class ComponentA{
2        private componentB;
3
4        void service(boolean condition,
5                     List array){
6            ServiceParametersFactory
                serviceParametersFactory = new
                ServiceParametersFactoryImp();
7      ServiceParameters __serviceParameters7b0c60ab =
          serviceParametersFactory
8    .getServiceParameters(new Object[] { condition, array
       }, new String[] { "condition", "array" });
9      ThreadMonitoringController.getInstance().
          enterService("_sPEbUDkyEembGJ6iNCoQDQ",
10   __serviceParameters7b0c60ab);
```

```
11
12              // internal computation
13              final long __tin_f7087098_bbc8 =
                    ThreadMonitoringController.getInstance().
                    getTime();
14              innerMethod_1();
15              ThreadMonitoringController.getInstance().
                    logResponseTime("_tWldADkyEembGJ6iNCoQDQ",
                    "_oro4gG3fEdy4YaaT-RYrLQ",
16      __tin_f7087098_bbc8);
17
18              // Branch Action
19              String __executedBranch_91862b17 = null;
20              if(condition){
21                  __executedBranch_91862b17 = "
                        _tXW5EDkyEembGJ6iNCoQDQ";
22                  ThreadMonitoringController.getInstance().
                        setCurrentCallerId("
                        _tXeN0DkyEembGJ6iNCoQDQ");
23                  componentB.service_1();
24              }
25              else{
26                  __executedBranch_91862b17 = "
                        _tXW5EDkyEembGJ6iNCoQDQ";
27                  long __counter_346b00f9_23d0 = 0;
28                  for(item in array){
29                      __counter_346b00f9_23d0++;
30                      ThreadMonitoringController.getInstance
                            ().setCurrentCallerId("
                            _nXeN0DkyHdnbGJ6iNCoQDW");
31                      componentB.service_2();
32                  }
33                  ThreadMonitoringController.getInstance().
                        logLoopIterationCount("
                        _tXbKgDkyEembGJ6iNCoQDQ",
34       __counter_346b00f9_23d0);
35              }
36              ThreadMonitoringController.getInstance().
                    logBranchExecution("_tXVD4DkyEembGJ6iNCoQDQ
                    ",
37      __executedBranch_91862b17);
38              ThreadMonitoringController.getInstance().
                    exitService();
39          }
40
```

```
41          private innerMethod_1(){
42              // internal computation
43              // ...
44          }
45
46      }
```

Listing 4.1: Instrumented version of the source code in Figure 2.2, the instrumentation is the result of the execution of our approach

4.7.4 Architecture

Figure 4.9 describes the architecture of our Instrumentation approach. The component diagram shows the dependencies between the components that we used to implement our approach.

Two Main Concerns
We distinguish between two main concerns, which are executed in diverse contexts. The first concern is related to the instrumentation of the source code. The second concern consists of the logic used to extract the parameters of the services and to log the monitoring information.

The Source Code Instrumentation Concern
In order to instrument the source code, we provided an instrumentation process which is implemented in the component *Source Code Instrumentation*. This component depends on the component *Probes Provider*, which provides the fine-grained probes (Section 4.2.1) we defined for our instrumentation approach. The component *Source Code Instrumentation* uses these probes in combination with information from the Correspondence Model(Section 2.2.2)to accomplish the instrumentation of the source code. we provide more details on the instrumentation process can be found in (Section 4.7.5.

The Monitoring Source Code Concern
This concern is related to the source code that has to be executed to log the monitoring information. it can be divided into two tasks. The first one consists of the extraction of the service parameters. This task is implemented by the components *Service Parameters Extractor* and *Service Parameters Factory*. The first component is responsible for extracting the service parameters values and their names and pass them to the second component. The second component receives the parameters of the services and put them in a defined format, currently, it produces a JSON file, which contains the service parameters names as the keys that their values as the values of the keys. The second task consists of the logic used to collect and log the monitoring information, which is implemented in the component *Monitoring*. The component *Source Code of the System* represents the source code that we want to instrument, it has dependencies to the components *Monitoring* and *Service Parameters Factory*.

4.7.5 Call Sequence of the Instrumentation Process

Figure 4.10 depicts the steps used in the Instrumentation Process to instrument the source code. we will describe each step in the following.

Step (1): Execute the Instrumentation Process
This step consists of starting the instrumentation process. Developers can execute this process at any time. Once the process is executed, it takes the probes from the Instrumentation Model (Section 4.5) and accomplishes the instrumentation of the source code.

Step (2): Copying the Source Code of the System
Here, we create a copy of the source code, which we will instrument. As we showed in (Section 4.7.3), we decided to instrument a copy of the source but not the original source code. Moreover, since we used Eclipse for our development purpose, we created functionality that clones the original project of the system with its dependencies and properties. Therefore, the instrumentation will find a place in the cloned project.

Step (4): Parse the Copied Source Code via JaMoPP
In this step, we create the Java model of the source code that we want to instrument, which is the copied source code. For parsing the Java source code, we used JaMoPP (Section 2.3). Moreover, the parsing of the source code will give us the possibility to manipulate it, like referencing the corresponding statements of a probe or the insertion of the instrumentation code in a defined position in the source code.

Step (5): Returning the Monitoring Probes
In this step, the Instrumentation Process calls the component Probes Provider to receive the probes that we want for the instrumentation.

Step (6): Finding the Probes Statements
As we mentioned before, we will instrument a copy of the source code, which means we will need to find the statements in the copied source code that correspond to our probes. However, the inputs that we have for the Instrumentation Process at this stage are the probes and the corresponding statements in the original source code. That means, in order to instrument the copied source code, we will need to search the corresponding statements of the probes in the copied source code. To do this, we compared the statements of the probes in the original source code with the statements of the copied source code.

In order to optimize the searching for corresponding statements of the probes in the copied source code, we used three parameters to identify equal statements. The first parameter is the service in which the statement is written. The second parameter is the class in which the statement find a place. The third parameter is the location of the statement in the source code. The combination of these parameters is unique for every statement in the source code. Therefore, it can be used to search for the probes state-

ments in the copied source code based on probes statements in the original source code.

Step (7): Instrument the Source Code
In this step, the copied source code will be instrumented based on the probes and their corresponding statements that have been mapped in the previous step. Moreover, the probes are injected into the source code based on their types (Section 4.2.1). Here, we distinguish between two kinds of instrumentation, namely coarse-grained instrumentation and fine-grained instrumentation. Fine-grained instrumentation is based on the probes that are represented by the SEFF elements that we defined in (Section 4.2.1). Coarse-grained instrumentation consists of instrumenting the whole service without taking into account its fine-grained probes. In this step, we execute both fine-grained and coarse-grained instrumentation.

Step (8): Coarse-grained Instrumentation
This step consists of the coarse-grained instrumentation of all services in the source code that have been not instrumented in the previous step. This step is required because of the adaptive instrumentation approach. As we mentioned before, in adaptive instrumentation, we collect probes only for the changed parts of the source code in the current iteration. That means services that have been not changed in the current iteration will not be related to any probes. Thus, they will not be instrumented in the previous step. However, even though they have been not changed in the current iteration, if they are called in the changed parts of the source code, we will need their monitoring information. Therefore, we finalized our instrumentation process by the coarse-grained instrumentation of the rest of the services of the system.

4.7.6 Service Parameters Extraction

Manar and Koziolek showed in their approach [20] that Performance Model parameters can strongly depend on the parameters of the services. Therefore, we decided to enrich our monitoring information by extracting the service parameters and their values.

Figure 4.11 shows our approach for extracting the service parameters. We limited our approach to five parameter types, namely the type Map, Collection, Array, Primitive and Data Types. Data Types are the types that are defined by the user and are given as parameters to a service. For example, a DataType can be Java Bean Class named FileType, which contains the attributes file name of type string and file content of type Array of bytes.

For service parameters extraction, we return for each parameter type only the property that has an impact on the performance model. For collections and maps, we extract their size. The same thing for arrays, we return their length. As for primitive types, we return the value of the parameter. Furthermore, Data types are handled via Reflection. In Data types we use reflection to search for attributes of the types Map, Collection, Array or Primitive in order to extract their defined properties.

4.8 Limitation

In this section, we will present the limitation of our process for the generation the monitoring probes.

Limitation (1): adaptive Instrumentation is limited to Service level
This limitation is due to the implementation of the SEFF Reconstruction Process (Section 2.4.1) which recreates the whole SEFF of a service if a part of its source code has changed. That means, if only one element of the SEFF of the changed service has been changed, the process creates a new SEFF with new elements and replaces the old SEFF of the service. This will guide to delete the old elements of the SEFF and loose their estimated parameters, even though they were not touched by the last changes. This limitation is reflected also on the adaptive instrumentation, the adaptive monitoring and the parameters estimation of SEFF models.

Another level that can be achieved to prevent this limitation is the statement level. In this level, only the SEFF elements that were touched by the source code changes have to be updated. In such a way, we can reduce the monitoring probes which reduces the overhead of the monitoring and parameters estimation of the performance model. This can be achieved for example by comparing the SEFF of the old service and the SEFF of the new service based on the source code statements. Using this comparison, we can deduce, which element of the old SEFF has been modified, newly added or deleted.

Limitation (2): probes with small response time must be ignored
This limitation is related to service calls and internal actions. Basically, probes with a response time that do not influence the performance model must be ignored. For example, an internal action that corresponds solely to a variable declaration will clearly log response time that does not affect the performance model.

We identify basically two approaches to find out if a probe with a given response time can be accepted or ignored. The First approach consists of analyzing the statements of the source code in order to decide if the probe can affect the performance model. This approach is efficient in terms of reducing the computation resources for the instrumentation, the monitoring, and the parameters estimation. However, we do not know if it's possible to apply it in all cases.

The second approach, which we've implemented in our approach is the filtering of the monitoring records under a defined response time value. However, this approach is applied after the monitoring phase and is less efficient than the first approach because we apply the computation resources in the monitoring of probes that are not relevant.

If the first approach can not cover all cases, it can be used in combination with the second approach in order to improve the efficiency of the monitoring. In the first step, the first approach can be used to remove probes that have a defined number and types of statements. In the second step, the second approach will remove the records with a defined response time value, which could not be judged by the first approach.

4.9 Extending the Incremental SEFF Reconstruction Process

In this section, we will introduce our contribution in the Incremental SEFF Reconstruction Process (Section 2.4.1). The objective of our contribution is to increase the level of incrementation of the SEFF reconstruction process. Moreover, our contribution aims to reduce the high overhead caused by the monitoring of large systems.

Our motivation for introducing this contribution was due to the shortcoming of the Incremental SEFF Reconstruction Process that we highlighted in limitation (2) (Section 4.8). Furthermore, the incremental SEFF generation of the services is limited to the service level. That means, if the source code that corresponds to one element of the SEFF elements of the service has changed, the process will create a new SEFF for the service and then remove its old SEFF. This includes removing the old elements of the SEFF even though they did not change. As a consequence, we will need to repeat the monitoring for all elements of the SEFF. In respect to this, the overhead of the monitoring will increase.

4.9.1 Objective

In our contribution to the Incremental SEFF Reconstruction Process, we will propose an approach that resolves the limitation presented above. We will precisely extend this process by updating only the SEFF elements that have been affected by the service changes. This will prevent the recreation of the whole SEFF of the service and spare untouched SEFF elements. As results, we can minimize the overhead of the monitoring.

4.9.2 Method

In order to achieve the objective presented above, we will base our approach to comparing the old and the new SEFF of the service. The old SEFF of the service can be obtained from Correspondence Model. The new SEFF of the service corresponds to the new source code of the service. The new SEFF of the service can be returned by executing the SoMoX on the new source code of the service.

When the source code of the service changes, we return its old SEFF and we run the SoMoX on it in order to receive the SEFF of the new source code of the service. The comparison of the old SEFF and the new SEFF of the service will provide a difference that can be analyzed in order to update the old SEFF of the service.

4.9.3 Scientific Challenges

In this section, we introduce the challenges that we need to solve in order to achieve the objective (Section 4.9.1) of our contribution.

- How can we compare the old and the new SEFF?

- How can we calculate the difference between the old and the new SEFF?
- How can we update the old SEFF based on the difference between the old and the new SEFF?

4.9.4 SEFF Comparison

As mentioned above, our approach is based on the comparison of the old and the new SEFF of the service. In order to compare the SEFF elements, we proposed to use their corresponding JaMoPP (Section 2.3) statements in the source code. Moreover, each SEFF element is represented by a set of statements in the source code, which means, in order to compare two SEFF elements we can simply compare their statements.

SPLevo: Diffing JaMoPP Cartridge In order to compare the JaMoPP statements, we used the software development tool SPLevo [13]. SPLevo provides the Diffing JaMoPP functionality, which can compare between JaMoPP models. We used this functionality to compare between JaMoPP statements, which we need, in order to compare SEFFs.

4.9.5 Approach

Figure 4.13 shows our contribution to the Incremental SEFF Reconstruction Process. The original process is described in (Section 2.4.1). In the following, we will describe the activities that we added to the process in order to achieve our objectives (Section 4.9.1). Furthermore, the activities in the blue color represent our contribution.

Step (1): Remove all abstract actions of the old method from the correspondence model
This activity is responsible for clearing the correspondence model from the old method and its correspondences. This is required because the old method will be replaced by the new method and we will need to create new correspondences for the new method.

Step (2): Return the Old SEFF
In this step, we use the old method reference to return its SEFF from the correspondence model.

Step (3): Run SoMoX to find the SEFF
In this step, we run the SoMoX on the new method in order to return its SEFF.

Step (4): Find the Matching of the New and the Old Resource Demanding Behavior
In this step, we compute the matching between the old SEFF and the new SEFF. The matching possibilities are represented by the UML class diagram in Figure 4.12. The output of this step is a list of the matching between the old SEFF elements and the new SEFF elements. The enumeration *MatchingType* describes the matching types,

which define if two SEFF elements are totally equal or not. Two SEFF elements are total equal if their statements match completely.

A matching of type modified means that the source code of the old SEFF element has changed but it has a matching in the new SEFF. This type of matching will inform us if a SEFF element has been modified.

Step (5): Find the difference between the new and the old SEFF
The difference between the old and the new SEFF is computed based on the matching returned from Step (4) and the elements of the old and the new SEFF. the difference is composed of two lists. (1) the first list contains the newly added SEFF elements that do not exist in the old SEFF but do exist in the new SEFF. The list of the newly added SEFF elements can be returned by searching in the elements of the new SEFF that do not exist in the matching list. (2) The second list contains the elements that must be deleted from the old SEFF. These elements do exist in the old SEFF but not in the SEFF. This list can be returned by searching the elements in the old SEFF that do not have any matching in the matching list.

Step (6): Update the old SEFF
In this step, we update the old SEFF based on the difference between the new and the old SEFF that we computed in the Step (5). we identify three cases of updating the old SEFF:

- Case (1): SEFF elements that have to be deleted
  we use the difference of the new and the old SEFF to return the SEFF elements that have to be deleted.

- Case (2): SEFF elements that their corresponding code has changed
  these elements have no effect on the SEFF structure and they will not be added or deleted. However, the corresponding source code has to instrumented and monitoring in order to provide these elements with the monitoring of the changed source code.

- Case (3): SEFF elements that we need to add
  the SEFF elements that we need to add in the old SEFF do not exist in the old SEFF but in the new SEFF. Moreover, in order to add these elements in the old SEFF, we need to find the location in which we should add them. Here, we identify two cases. (1) if the SEFF element that we want to add in the old SEFF has no predecessor in the new SEFF that has a matching in the old SEFF, the SEFF element that we want to add, has to be added in the second position of the old SEFF. (2) else, we add the SEFF element that we want to add, after the SEFF element in the old SEFF that matches with the first predecessor of the SEFF element that we want to add.

Step (7): Create new correspondences between the new method and the SEFF elements

In this step, we create a correspondence in the Correspondence Model between the new method and actual old SEFF elements.

Step (8): Create correspondences between the SEFF elements and the corresponding JaMoPP statements
In this step, we create the correspondence in the Correspondence Model between the actual old SEFF elements and the corresponding JaMoPP statements. These correspondences will be needed in probe generation.

4.9.6 Implementation Limitation

We implemented the steps of the approach that we presented to reconstruct the SEFF incrementally and based on the statement level. We used the software development tool SPLevo [13] to compare different JaMoPP statements. Moreover, our testing has succeeded in updating the Internal Actions and Branch Actions. However, we could not update incrementally all the elements of SEFF. In particular, we could not compare correctly the source code statements that correspond to Loop Actions in SEFF. Furthermore, because of the time limit of this thesis, we could not improve this approach. Therefore, we suggest improving this approach in a future job.

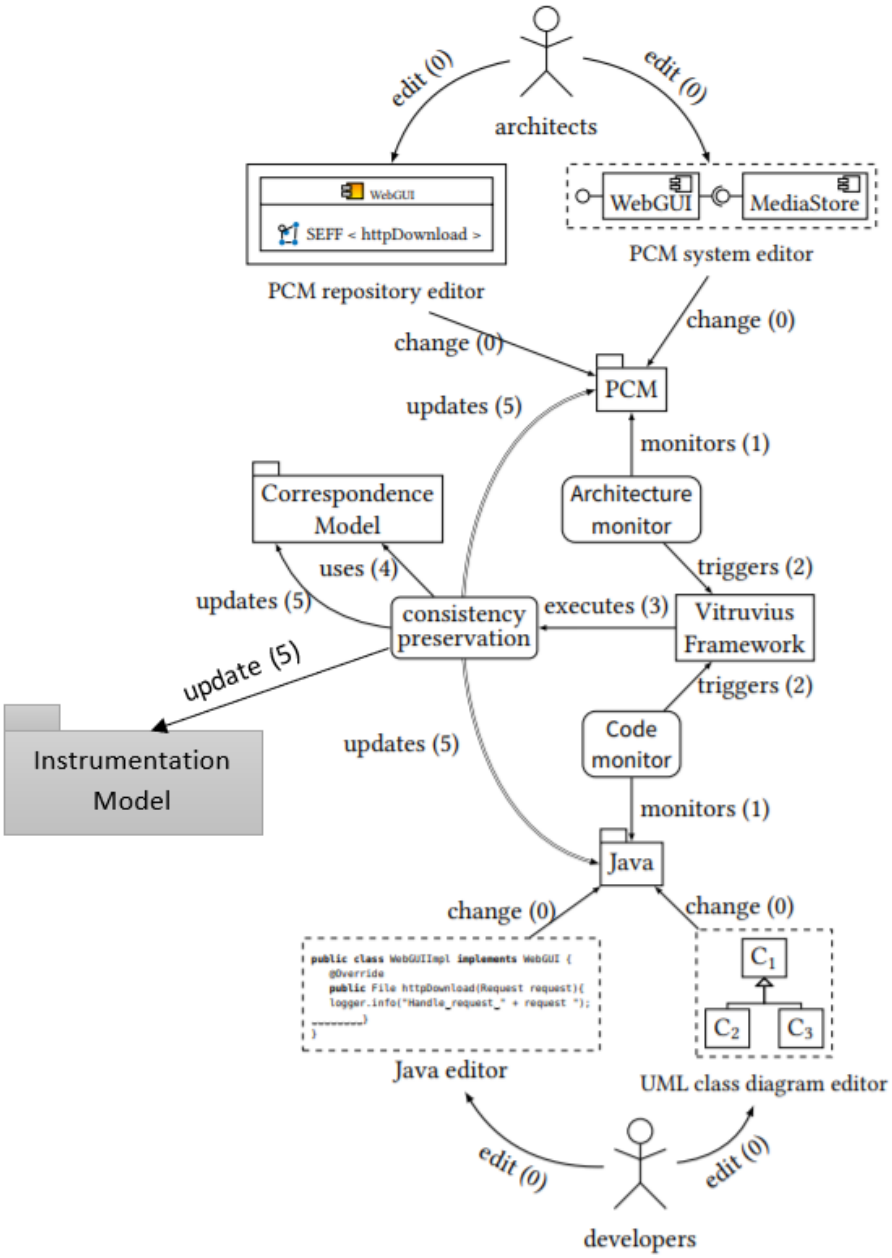

Figure 4.5: Overview of the Vitruvius VSUM of our approach, we extended the VSUM of the Coevolution approach

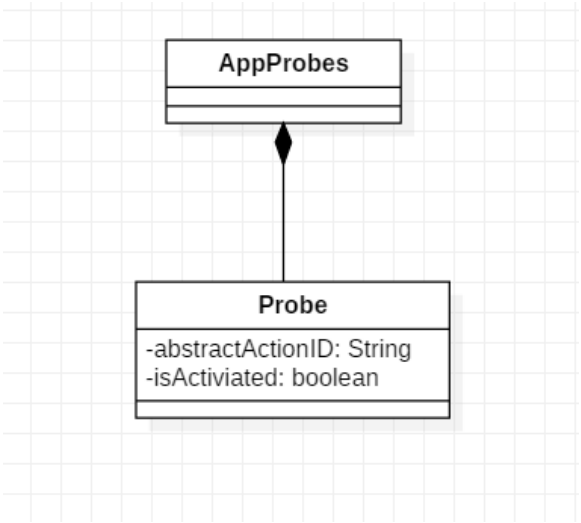

Figure 4.6: UML Class Diagram that represents the Instrumentation Meta-model (IMM)

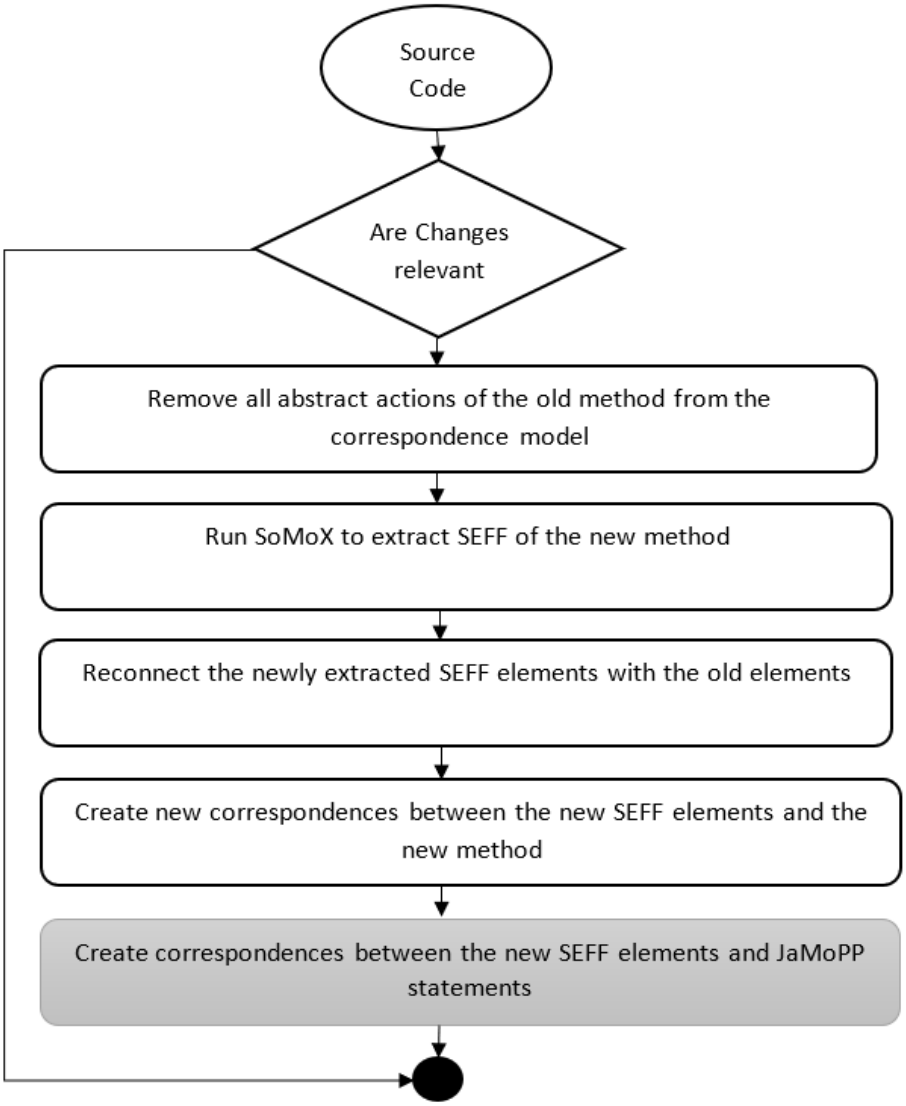

Figure 4.7: The extended Incremental Reconstruction Process of SEFF

Figure 4.8: The inputs and the output of the instrumentation process, the output is an instrumented copy of the original source code of the system

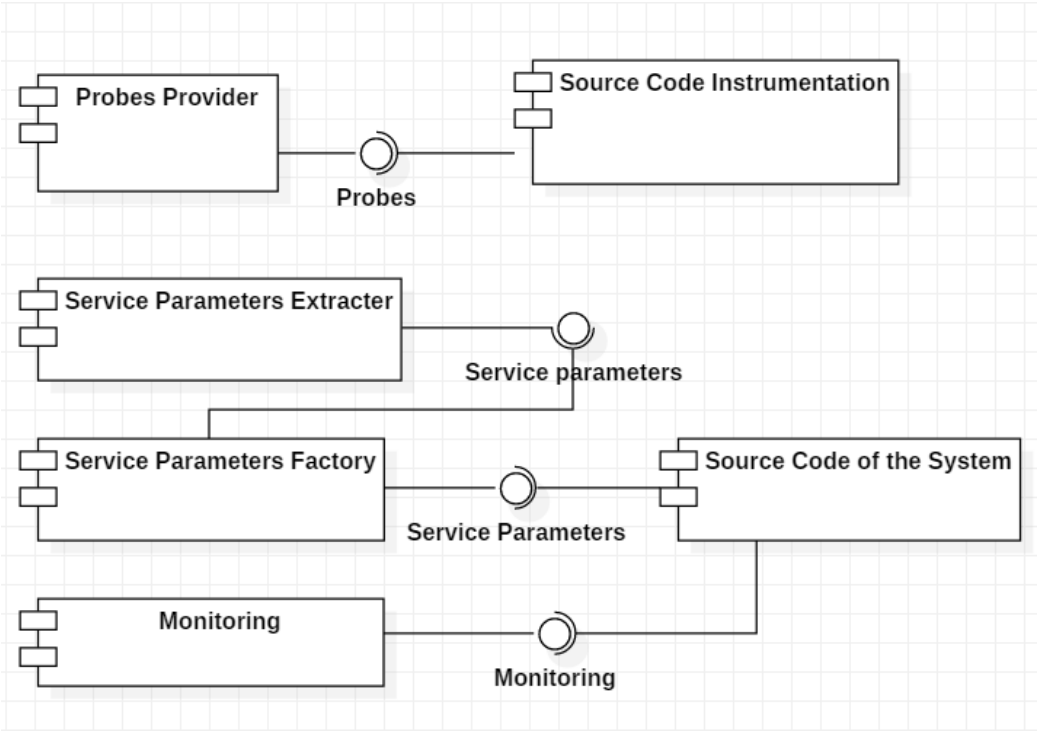

Figure 4.9: The Component Diagrams shows the dependencies between the components we used to implement our approach

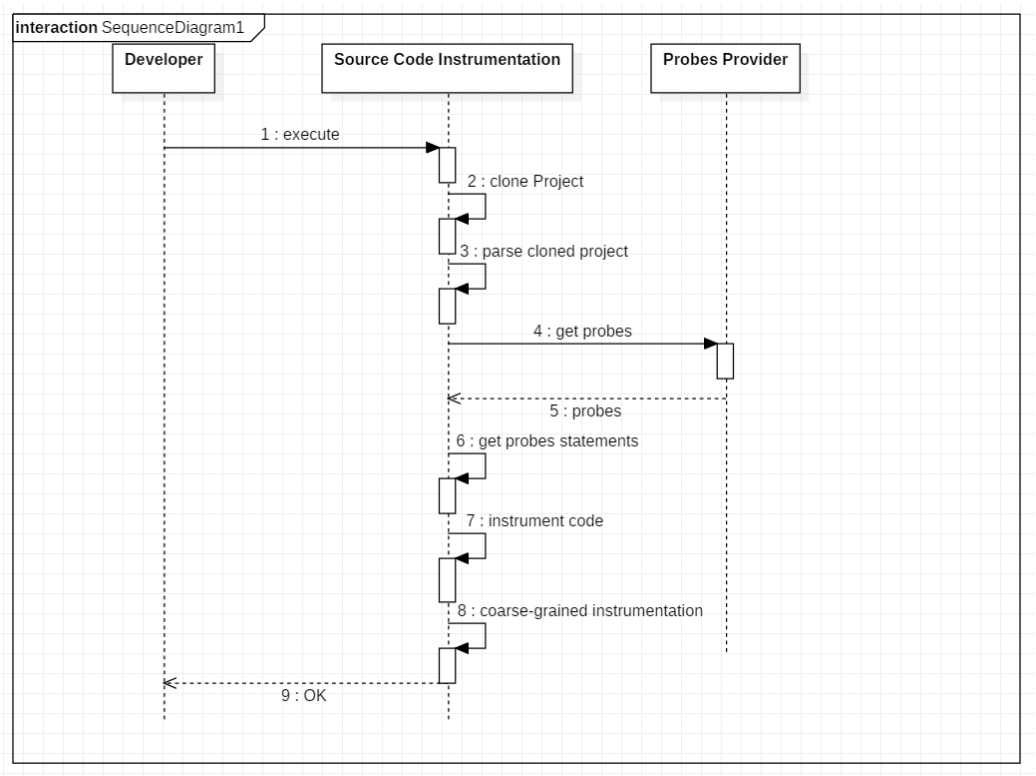

Figure 4.10: Sequence Diagram that illustrates the activities of the Instrumentation Process

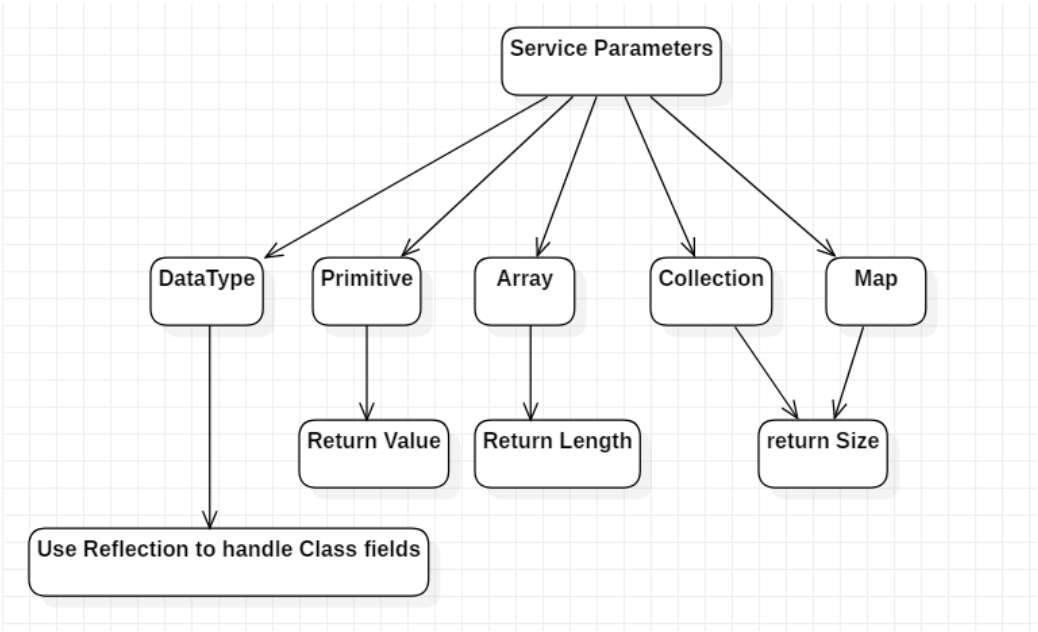

Figure 4.11: Service parameters types and how they are extracted

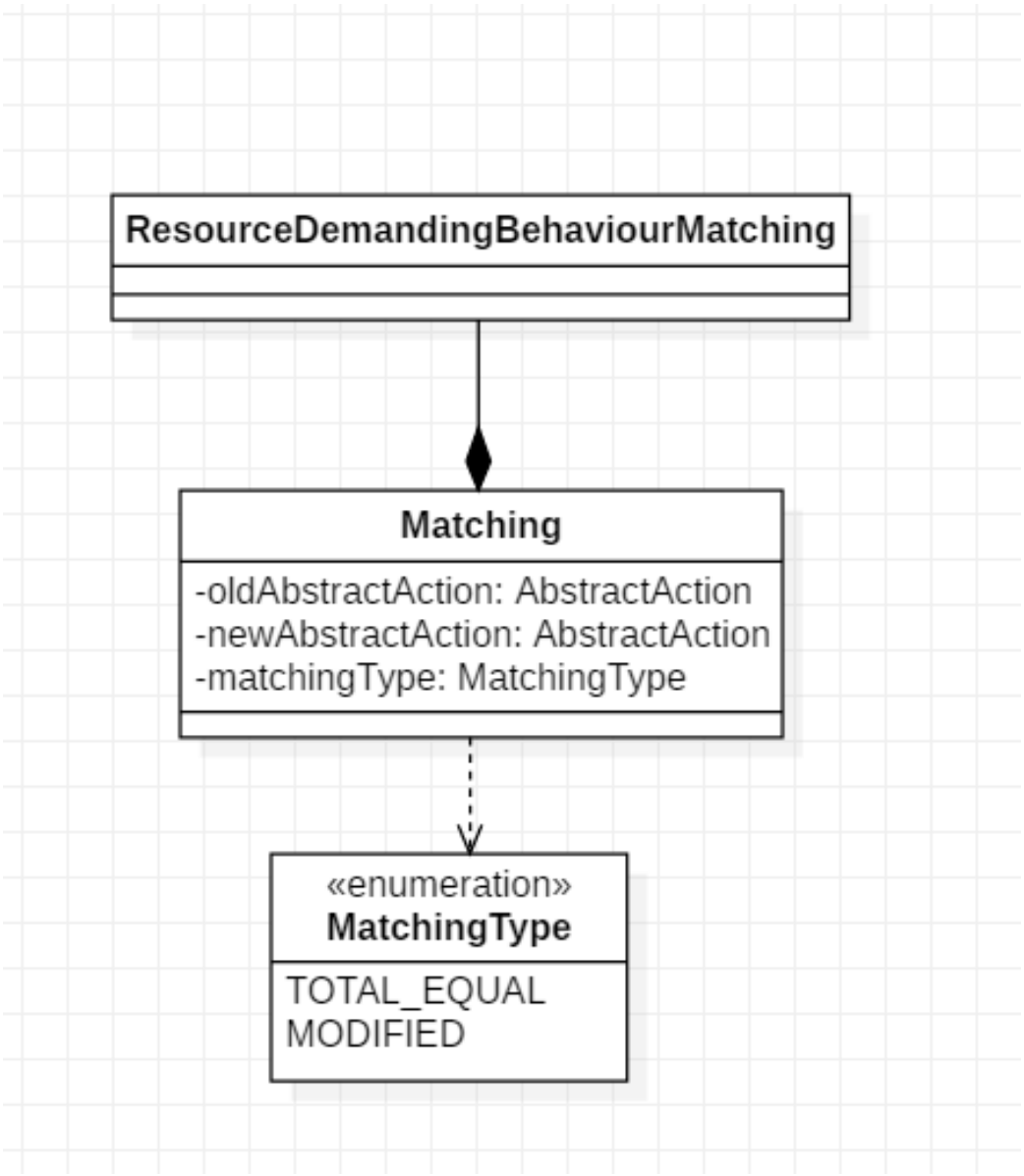

Figure 4.12: UML class diagram that models the matching between the old and the new SEFF

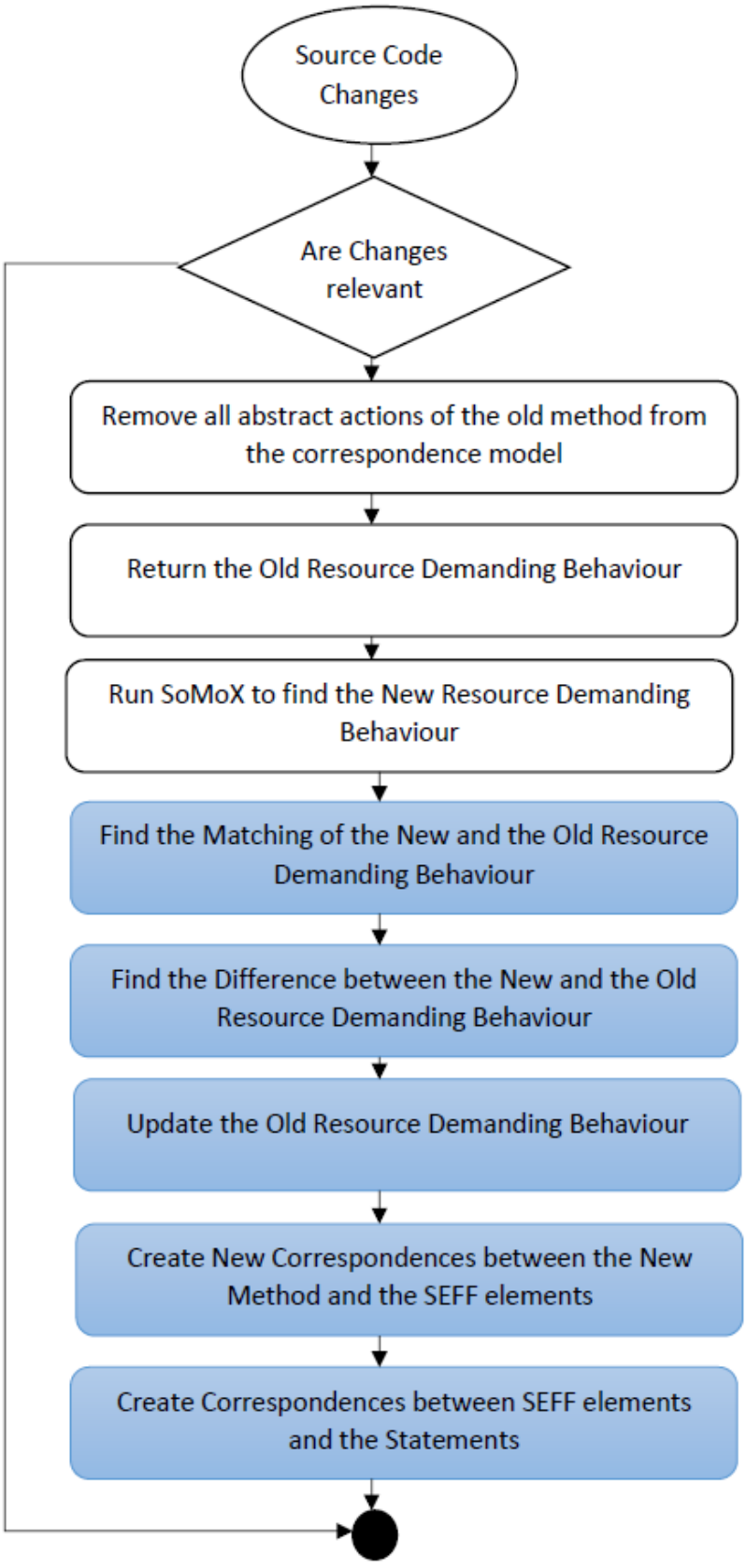

Figure 4.13: The activities in blue color shows our contribution to the Incremental SEFF Reconstruction Process 2.4.1

5 Evaluation

In this section, we introduce the evaluation of our approach. We divided our evaluation into levels. In the first level, we evaluate the adaptive instrumentation. In the second level, we evaluate the monitoring information that we created to make sure that we've created the right monitoring information. In the third level, we evaluate our approach in term o reducing the monitoring overhead.
In the following we defined the goals (G) and research questions (Q) for our evaluation:

G1: Adaptive Instrumentation

Q1.1: Is our instrumentation of the source code incremental?

M1.1.1: the difference between the probes in the instrumentation model and the generated probes in the instrumented source code.

Q1.2: did we generate the right monitoring information?

M1.2.1: Number of acceptances of our monitoring information from performance model parameters estimation approach.

G2: Adaptive Monitoring

Q2.1: How much monitoring overhead could we reduce?

M2.1.1: number of generated monitoring records.

5.1 Case Study

In order to evaluate our approach, we performed a case study that is represented by the component diagram in Figure 5.1. We created two Java interfaces and we implemented them. The first interface *ISearchingAlgo* contains two functions. The service *sequentianSearch*, which receives an Array of integers and a value of type integer, it returns True if the given value exists in the given array. This service is based on the sequential searching algorithm. The second service *binarySearch* perform receives also an Array of integers and an integer value, it must then tell us if the given could be found in the given array. The service *binarySearch* is implemented based on the binary searching algorithm. The binary searching algorithm is more efficient then the sequential searching algorithm, we choose these algorithms to provide the possibility of having different response times and service calls parameters.

The second interface *GuiSearchingAlgo* provides also two services and requires services from the interface *ISearchingAlgo*. The service *guiSeuentialSearch* receives two arrays of integers, the first array represents the array in which we want to perform the searching, the second array represents the values we want to search

for. This service iterates over the array of values that we want to search for and calls the service *sequentialSeach* from the interface *ISearchingAlgo*, which tells if the current searched value could be found. Figure 5.3 shows the corresponding SEFF of the *guiSequentialSearch* service. The service *guiBinarySearch* in the interface *GuiSearchingAlgo* does the same thing as the service *guiSequentialSearch* but it calls the service *binarySearch* instead of *sequentialSeach* service.

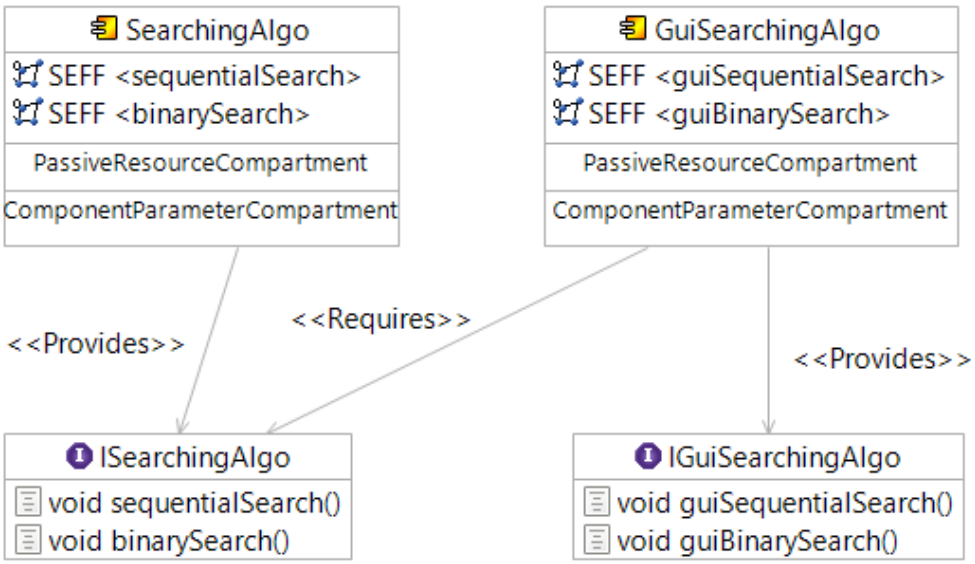

Figure 5.1: Component diagram of our case study

5.2 Adaptive Instrumentation

In this section, we use our case study to evaluate the incremental instrumentation and monitoring information types. We evaluate the incremental instrumentation to make sure that we could instrument the source code incrementally in our approach. We evaluate the monitoring information types to make sure that we've generated the right information that can be used by performance model parameters estimation approaches.

5.2.1 Incremental Instrumentation

In the first level of our evaluation, we evaluate the incremental instrumentation of our approach. As a metric for this evaluation, we decided to compute the difference between the probes in the instrumentation model and the monitoring probes in the instrumented source code in each iteration. The reason we used this metric is because of the incremental generation of probes in our instrumentation model. That means,

in each iteration, we collect only the probes of the changed parts of the sour code and save them in the instrumentation model. Moreover, if we instrumented the source code incrementally, all the probes in the instrumented source code must be found in the instrumentation model. That means also that the difference between the probes in the logged monitoring information and in the instrumentation model must be empty. However, if this difference is not empty, that means, there exist probes in the instrumented source code, even though they do exist in the instrumentation model. That implied that the instrumentation has been not done incrementally.

In order to return the probes in the instrumented source code, we run it to receive the monitoring records. Each monitoring record corresponds to a monitoring probe in the source code. Moreover, monitoring records are can be identified via the id of their monitoring probes. Therefore, we can use the monitoring records after the execution of the instrumented source code to extract the monitoring probes in the source code.

Moreover, the monitoring probes in our instrumentation model are saved via their ids. Furthermore, since the monitoring records have the same id as their monitoring probes, we can then compute the difference between the existing ids of the probes in the instrumentation model and ids of the monitoring records in the monitoring information.

As mentioned above, in order to evaluate the incremental instrumentation, we calculate the difference between the probes in the monitoring information and in the instrumented source code. Therefore, we created two iterations Figure 5.2. In the first iteration, we changed the source code of our case study by implementing the services *sequentialSearch* and *guiSquentialSearch*. During the source code changes, we collected the probes of the changed parts of the source code, then we instrument the source code based on these probes. Then, we executed the instrumented source code to generate the monitoring information.

In the second iteration, we changed the services *binarySearch* and *guiBinarySearch*. Then, we followed the same steps in the first iteration to generate the monitoring information for the second iteration.

Figure 5.2, shows that the difference between the probes in the instrumented source code and the probes in the instrumentation model is empty in the first and in the second iteration, which means our instrumentation has executed incrementally.

5.2.2 Monitoring Information

In the second level of our evaluation, we evaluate the monitoring information that we log using our incremental instrumentation. The source code that we instrument has to provide monitoring information that can be used to estimate parameters for the Palladio performance model.

Our monitoring information is defined in records (Section 4.2.2), which are four types, namely response time record, loop record, branch record, and service call record.

In order to make sure that we generate the right records, we've instrumented our case study and executed the instrumented source code to generate the monitoring records for all types of records that we defined in our approach. Then, we used our

Iterations	IDs of the Probes in IM (ids)	IDs of the Probes in the instrumented code	Diff
Iteration-0	_cnPYcDD7EemNS-Dzmbs5Uw _c1sEIDD7EemNS-Dzmbs5Uw _c2c5IDD7EemNS-Dzmbs5Uw _c2XZkDD7EemNS-Dzmbs5Uw	_cnPYcDD7EemNS-Dzmbs5Uw _c1sEIDD7EemNS-Dzmbs5Uw _c2c5IDD7EemNS-Dzmbs5Uw _c2XZkDD7EemNS-Dzmbs5Uw	0
Iteration-1	_s4bhYDkyEembGJ6iNCoQDQ _tIFG8DkyEembGJ6iNCoQDQ _tXbKgDkyEembGJ6iNCoQDQ _tXVD4DkyEembGJ6iNCoQDQ	_s4bhYDkyEembGJ6iNCoQDQ _tIFG8DkyEembGJ6iNCoQDQ _tXbKgDkyEembGJ6iNCoQDQ _tXVD4DkyEembGJ6iNCoQDQ	0

Figure 5.2: difference between the probes in the instrumentation mode and in the monitoring, information is empty in the first and the second iteration, which means our instrumentation for the case study has been done incrementally

monitoring records and the performance model parameters approach of Jägers [10] to estimate the parameters for the Palladio Performance Model.

Figure 5.3 shows the SEFF that corresponds to the service *guiSequetialSearch*. The estimated parameters of this model have been done based on our monitoring records. Moreover, the case study contains all the record types that we defined, which gives us the possibility make sure that all the records that we defined can be used in the palladio performance model parameters estimation.

5.3 Adaptive Monitoring

In the third level of our evaluation, we answer the question, how much monitoring overhead can we reduce using our approach.

Our experiment is based on the adaptive instrumentation of the source code in (Section 5.2). After we run the instrumented source code in the iteration-0, we counted 6700 records for one load test. For the monitoring of the instrumented source code in the iteration-1, we counted 6700 records for one load test.

In we instrument completely the source code in the iteration-1, we count 13400 records for one load test. If we instrument incrementally the source code in the iteration-1, we receive 6700 records, which is about 50% the amount we count if we did not instrument the source code incrementally.

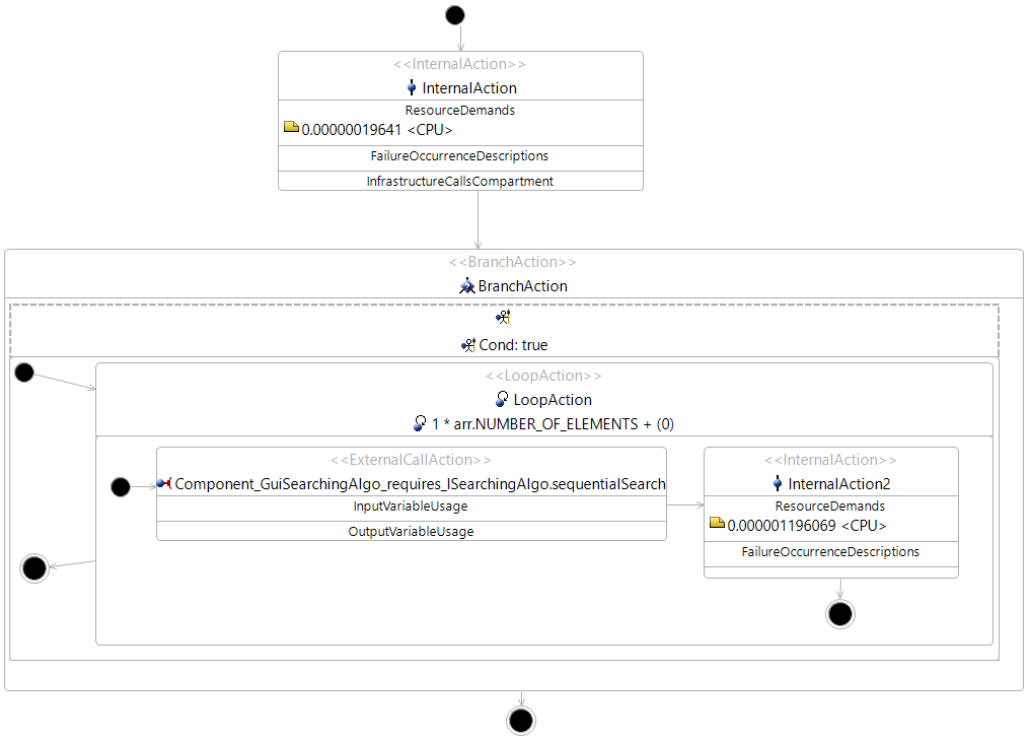

Figure 5.3: shows the SEFF of the service *guiSequentialSearch* and its estimated parameters based on our records that received from monitoring this service

6 Related Work

None of the existing approaches that deal with the automatic extraction of performance models considers the parametric dependencies of performance models, the adaptive monitoring and the incremental execution of the whole process at the same time. Moreover, existing approaches do not support the adaptive monitoring for performance model continuous integration.

Walter et al. [24] provided an expandable approach for automated extraction of the architectural performance model. He presents a framework for the extraction of the architectural performance models based on monitoring files generalizing over the target architectural model language. This framework can be used to create an extraction tool for a specific modeling formalism. He implemented his approach in so-called Performance Model Extractor (PMX) tool and provided it as a web service.

Krogmann et al. [16, 17] used genetic programming to estimate parametric dependencies between inputs and the number of execution for each byte code instruction. Depending on input data he could find control flow dependencies between required and provided services. However, for receiving monitoring information, he instruments the whole system in each iteration, which causes a monitoring overhead.

Spinner et al. [22] predicted the system performance at run time under varying workloads and system configuration. He proposed an approach for obtaining architectural-level model performance in a virtualized environment.

Langhammer [18] introduced two approaches to extract the static and dynamic behavior of the source code. Furthermore, he proposed an approach that keeps the source code and the architecture consistent during development. Furthermore, he could generate performance model incrementally. However, he did not use an adaptive monitoring approach, which means, he instruments the whole system in each iteration. Therefore, the monitoring overhead could not be reduced.

Brosig et. al [3] introduce an approach that the Palladio Component Models of Java EE applications based on the monitoring data collected during operation. However, he did take into account the adaptive monitoring, which means he monitor the whole source code in each iteration. Therefore, monitoring huge Java EE applications will lead to monitoring overhead.

Jägers has also introduced in his master thesis [10] an approach that estimates incrementally the performance model parameters considering parametric dependencies. Moreover, he used fine-grained monitoring information for the estimation. However, he instrumented the source code manually.

7 Conclusions and Future Work

This chapter presents a conclusion of our approach and introduces suggestions for future work.

7.0.1 Summary

We presented an approach that monitors the source code adaptively and provides performance models with the needed monitoring information for estimating the performance models parameters. Our approach is part of the continuous integration of performance model, presented by Mazkatli and Koziolek. we can provide fine-grained monitoring information that is needed for estimating Palladio Component Model like the number of loop execution, the probability of selecting a branch and resource demands. Moreover, we can provide the external calls arguments, which can be used to estimate performance model parameters considering parametric dependencies.

We extended the Vitruvius VSUM with an Instrumentation Meta-model that can be used to describe and manage probes for source code instrumentation. The generation of these probes is based on source code changes. Moreover, we created a transformation that keeps the source code and the Instrumentation model consistent.

We provided an adaptive and automatic instrumentation approach that instruments the source code iteratively and automatically. We based our instrumentation on the Kiekeker Monitoring Framework. We extended Kieker with new monitoring records that contain the monitoring information needed for performance model parameters estimation.

We provided an adaptive monitoring approach, which is based on the activeness of the probes. That means, the probes can be activated and deactivated during the monitoring phase, so that only the activated probes will be monitored.

We evaluate our approach on three levels. In the first level, we showed that we could instrument the source code incrementally. In the second level, we showed that the monitoring information that we generated can be used for performance model parameters estimation. In the third level, we showed that we could reduce the monitoring overhead to 50% in our case study.

7.0.2 Future Work

We identify two disadvantages of our approach. The first one is related to the filtering of the probes that have no effect on the performance model. These probes are for example internal actions that contain only one variable declaration, which has less response time then logic used to instrument them. In our approach. we filter these probes after the monitoring phase, which helps to reduce the monitoring

records number and thus reduce the time used for the performance model parameters estimation. Therefore, we suggest creating in future work, an approach that identifies these probes based on a static analysis of the source code. Such an approach will also reduce the monitoring overhead.

The second disadvantage is related to the level of adaptive monitoring in our approach. As we mentioned above, our incremental monitoring is limited service level changes. That means, if one part of the source code of service has changed, we instrument all the probes in the service. We proposed an approach (Section 4.9) that resolves this limitation. However, we could not evaluate it at the planned time of this thesis.

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 Koziolek, 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] Fabian Brosig, Samuel Kounev, and Klaus Krogmann. “Automated extraction of palladio component models from running enterprise Java applications”. In: *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and ... 2009, p. 10.

[4] Andreas Brunnert et al. “Performance-oriented DevOps: A research agenda”. In: *arXiv preprint arXiv:1508.04752* (2015).

[5] Erik Burger. “Flexible Views for View-based Model-driven Development”. PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology, 2014.

[6] Thomas Goldschmidt, Steffen Becker, and Erik Burger. “View-based Modelling-A Tool Oriented Analysis”. In: *Proceedings of the Modellierung*. 2012.

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

[8] 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.

[9] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. “Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM, 2012. URL: http://doi.acm.org/10.1145/2188286.2188326.

[10] Jan-Philipp Jägers. “Iterative Performance Model Parameter Estimation Considering Parametric Dependencies”. MA thesis. KIT, 2018.

[11] Reiner Jung. *An instrumentation record language for kieker*. Tech. rep. Tech. rep. Kiel University, 2013.

[12] 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.

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

[14] Heiko Koziolek, 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.

[15] 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. URL: http://doi.acm.org/10.1145/2489861.2489864.

[16] Klaus Krogmann. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis*. Vol. 4. KIT Scientific Publishing, 2012.

[17] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. "Using genetic search for reverse engineering of parametric behavior models for performance prediction". In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 865–877.

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

[19] 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.

[20] Manar Mazkatli and Anne Koziolek. "Continuous Integration of Performance Model". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM. 2018, pp. 153–158.

[21] Ralf H Reussner et al. "Modeling and Simulating Software Architectures: The Palladio Approach (MIT Press)". In: (2016).

[22] Simon Spinner, Jürgen Walter, and Samuel Kounev. "A reference architecture for online performance model extraction in virtualized environments". In: *Companion Publication for ACM/SPEC on International Conference on Performance Engineering*. ACM. 2016, pp. 57–62.

[23] Dave Steinberg et al. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[24] Jürgen Walter et al. "An expandable extraction framework for architectural performance models". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM. 2017, pp. 165–170.