1. **Monitoring Probes Generation Process**

As mentioned before, we used the Vitruvius Framework and change-driven approach to create the probes in order to achieve the adaptive instrumentation. Moreover, we showed that the generation of the monitoring probes is equivalent to keeping the source code model and the instrumentation model consistent. the consistency between these models is realised within Vitruvius.

In this section, we will introduce our approach for the monitoring probes generation as well as the collection of the information that is needed for the instrumentation process. In section 1.1, we introduce the concepts that are used in our approach. In section 1.2, we present the Vitruvius VSUM of our approach. In section 1.3, we describe how we collect the information that we need in our approach and the transformation that keeps the source code and the instrumentation model consistent. In section 1.4, we provide an overview of the limitation of our approach.

* 1. **Terminology**

In this section, we present the terminologies and the concepts we use to achieve the goals of our approach.

* + 1. **Source Code Decorator Model**

The Source Code Decorator Model (SCDM) has been presented within the SoMoX approach (section 7). SoMoX reverse-engineers the source code and can extracts 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 5) 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, bevor, 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 corresponds 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 the Incremental SEFF Reconstruction Process (section 7.2) 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 1**.

Figure 1: Source Code Decorator Model (SCDM) maps between the SEFF Elements and the corresponding JaMoPP statements

**SCDM**

**SEFF**

**JaMoPP**

Branch Action

Internal Action

Loop Action

**∑** Statement

**∑** Statement

**∑** Statement

Service Call Action

**∑** Statement

* + 1. **Correspondence Metamodel**

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** shows the Vitruvius Correspondence Metamodel. It’s basically composed from two classes. The root class *Correspondences* contains the list of Correspondence. The class *Correspondence* is composed from two list 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 a generic and can be used for diverse metamodels. 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 identifies 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.

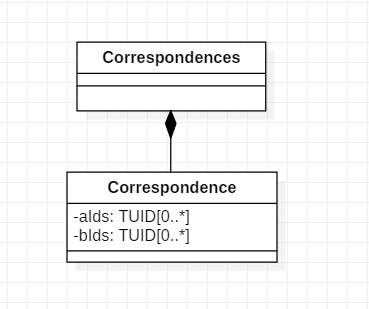


Figure 2: The Correspondence Metamodel of Vitruvius

* + 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 1**. 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 [1]. Projectional view types allows to show information from one metamodel solely. Combining view types can be used to show information from diverse metamodels.

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 models transformation that preserves the consistency.

MM1

MM2

MM3

CPR

CPR

VT2

VT3

VT1

View\_2a

View\_2b

View\_3a

View\_1a

Instance of

Instance of

Instance of

Instance of

**Legend:**

CPR

VT

View type

Consistency preservation rules

Consistency preservation process

Refers to

Figure 3: Overview of the Vitruvius VSUM and how model instances can be kept consistent

* 1. **The Vitruvius VSUM of our approach**

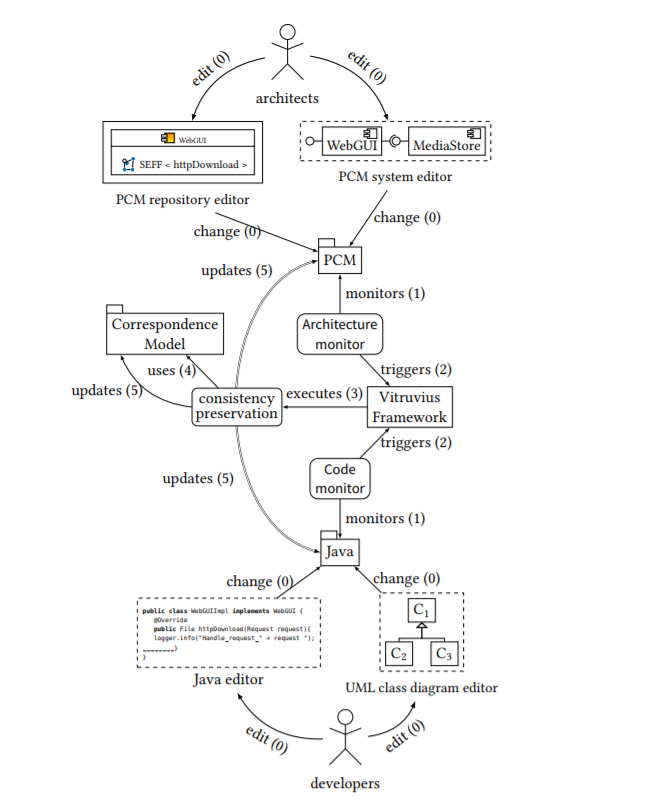
In this section, we will present the VSUM of our approach in order to generate the monitoring probes. We will precisely extend the VSUM **Figure 3** (in foundation) defined by the Coevolution approach.

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 from 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 5 in foundation).

In order to use Vitruvius for the adaptive monitoring purpose, we added an Instrumentation Metamodel (section 1.6 in chapter approach) that saves the monitoring probes and keeps them consistent with the source code. Figure 4 shows our VSUM which is extended from the VSUM of the Coevolution approach. The steps described in (section 5 in foundation) are kept the same. However, we added a transformation (see section 1.3.2) in the step (5) which keeps the Java source code and the instrumentation model consistent.



update (5)

InstrumentationModel

Figure 4: Overview of the Vitruvius VSUM of our approach, we extended the VSUM of the Coevolution approach

* 1. **Monitoring 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 collection 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.

* + 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 the 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 7.2 in foundation). This process reconstructs incrementally the SEFF of a 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 7.2 in foundation), the SoMoX is executed to reverse-engineer the changes service and creates its SEFF. SoMoX creates also the Source Code Decorator Model (SCDM) (section 1.1.1) which contains links 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 5**.

Are Changes relevant

Remove all abstract actions of the old method from the correspondence model

Run SoMoX to extract SEFF of the new method

Reconnect the newly extracted SEFF elements with the old elements

Create new correspondences between the new SEFF elements and the new method

Create correspondences between the new SEFF elements and JaMoPP statements

Figure 5: The extended Incremental Reconstruction Process of SEFF

* + 1. **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 1.6 in the chapter approach) 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 new added probes are by default activated (section 1.6 in the chapter approach).

* 1. **Limitation**

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

**Adaptive Instrumentation is limited to Service level**

This limitation is due to the implementation of the SEFF Reconstruction Process (section 1.6 in chapter foundation) 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 tough 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 was touched by the source code changes have to be updated. In such 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, new added or deleted.

**Probes with small response time must be ignored**

This limitation is related to service calls and internal actions. Basically, probes with 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 analysing 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 for 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 monitoring phase and is less efficient then 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 has a defined number and types of statements. In the second step, the second approach will remove the records with a defined response tine value, which could not be judged by the first approach.

**Reference**

[1]Erik Burger. “Flexible Views for View-based Model-driven Development”. PhD thesis. Karlsruhe, Germany: Karlsruhe Institute of Technology, July 2014. isbn: 978-3-7315-0276-0. doi: 10.5445/KSP/1000043437. url: http://digbib .ubka.uni-karlsruhe.de/volltexte/1000043437.