Optimizing Parametric Dependencies for Incremental Performance Model Extraction
Second Title Line

Code Review of the Bachelor's thesis of

Sonya Voneva

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

25. November 2019 – 13. Februar 2020

Contents

1 Introduction **1**

1.1 Requirements 2

1.2 Test case 2

2 Code Overview **4**

2.1 Building dataset 5

2.2 Feature selection 5

2.3 Initial dependencies 6

2.4 Optimizing with Genetic Programming 6

Bibliography **10**

1 Introduction

Often in software enginneering developers need to answer what-if scenarios. The most optimal way to do that (without investing resources in implementing the different alternatives) is with a so-called software performance model. In order for this model to be more accurate, it has to be parametrized. What this means, is that the parametric dependencies between service parameters of the software and some performance model parameters have to be taken into consideration. In our case the performance model parameters are:

- Loop iterations count
- Branch transition probability
- Resource demand
- External call arguments

The parametric dependencies for the first three were already studied in the work of Jägers [3]. The subject of my Bachelor's thesis is to identify parametric dependencies between the parameters of a service (serviceA in Listing 1.1) and the parameters of the external calls within this service (serviceB2 and serviceB3 in Listing 1.1). And additionally, to optimize the existing dependencies for the other types of performance model parameters.

```
  public class A {
  private B externalB;
  public int serviceA (int a, int b, boolean x){
    /* Some internal actions */
    int result1 = externalB.serviceB1(a * 5 + Math.pow(b,3));
    if(a>=0){
      ArrayList<Boolean> result2 = externalB.serviceB2(x);
    } else {
      ArrayList<Boolean> result2 = externalB.serviceB3(!x);
    }
    return result1 + result2.length;
  }
}
```

Listing 1.1: Example of a service (serviceA) calling external services (serviceB1 and serviceB2 or serviceB3)

We use the *Palladio Component Model(PCM)* [1] for modeling the software. The models are created by developers manually and each software component and its interfaces are stored in a PCM repository. The behaviour of a software component is refected in a so-called *Service*

Effect Specification (SEFF). A SEFF can contain internal actions, branch transitions, external call actions and many more code elements supported by the Palladio framework. Another specific of Palladio is that the performance model parameters (listed above) are defined using the specialized language of *Stochastic Expressions (StoEx)* [5].

1.1 Requirements

The setup used for development is Eclipse Modeling version 2019-09 R (4.13.0) with JavaSE 1.8 and the following plug-ins, which can be installed from the referenced update sites.

- Instrumentation Record Language [1]
- Palladio [2]
- JaMoPP [3]
- Vitruv [4]

Two of the projects are Maven projects, so Maven integration for Eclipse has to be installed as well (m2e and m2e connector for maven archiver pom properties).
The framework which is used for collecting monitoring data at runtime is Kieker [2]. There is no need to install it additionally, as it is built in as maven dependency.
The Java libraries, containing the estimation algorithms - Weka and Jenetics are also specified as maven dependencies.

1.2 Test case

The evaluation of the implemented optimization technique is yet to be done with a proper case study. For testing purposes following steps have to be conducted (for orientation see the simple example in `tests/tools.vitruv.applications.pcmjava.modelrefinement.parameters.tests/src/tools/vitruv/applications/pcmjava/modelrefinement/parameters/arguments`):

1. Create small java program, in which a service calls some external service
2. Create PCM for the program (`./test-data/arguments_test`)
3. Instrument the program with the help of the ThreadMonitoringController class
4. In the main method of the program: load, update and save the PCM
5. Run the program one time to collect monitoring records. During each run Kieker writes a new folder in `./test-data/arguments_test/kieker` containing the date and time in its name

[1] `https://build.se.informatik.uni-kiel.de/eus/mdm/snapshot/`
[2] `https://sdqweb.ipd.kit.edu/eclipse/palladiosimulator/releases/1.1.0`
[3] `http://update.jamopp.org/trunk/`
[4] `http://vitruv.tools/updatesite/nightly`

6. Copy the two kieker files from the newly created folder into `./test-data/arguments_test/kieker`

After the second run the parametric dependencies should be displayed in the console as output and the parameterized PCM should be in `./test-data/arguments_test/res`. The last two steps will be eliminated in the future. The monitoring and update of PCM should happen sequentially.

2 Code Overview

The workflow of our approach and the difference between related work and contribution can be seen in fig. 2.1.

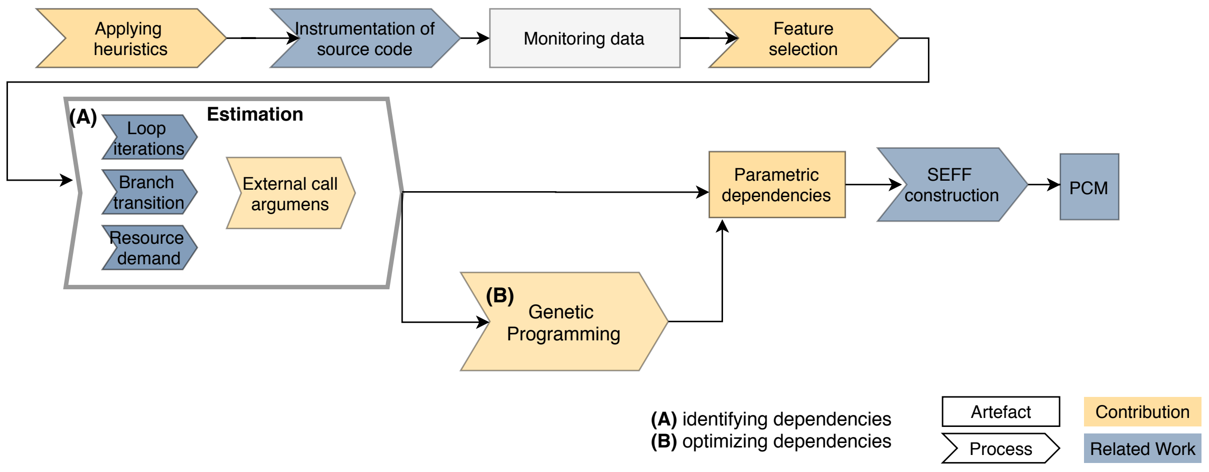

Figure 2.1: Workflow of the SEFF parameters estimation

The main project sources are located under the directory `/bundles` and the tests under the directory `/tests`. The code is separated in four projects as follows: fig. 2.2

- **tools.vitruv.applications.pcmjava.modelrefinement.parameters** contains the analysis part of the SEFF parameter estimation. Monitoring records are read and we estimate Stochastic Expressions based on them. The class SeffParameterEstimation is the main entry point.
- **tools.vitruv.applications.pcmjava.modelrefinement.parameters.monitoring** contains logic which is needed by instrumented code, in order to produce monitoring records. The main class is ThreadMonitoringController
- **tools.vitruv.applications.pcmjava.modelrefinement.parameters.optimizing** contains the logic for the genetic programming optimization of parametric dependencies. The class for starting a new optimization is Optimization.
- **tools.vitruv.applications.pcmjava.modelrefinement.parameters.tests** encapsulates different test cases and their data(PCM repositories, monitoring data...)

The following fig. 2.3 describes the architecture of the SEFF Parameter estimation. The functionality is divided into the six components External Call Argument Estimation, Loop

Estimation, Branch Estimation, Resource Demand Estimation, Utilization Estimation and SEFF Parameter Estimation, which delegates tasks to the other components. My code extends the External Call Argument Estimation (yellow marked). The rest of the code was available from previous work.

2.1 Building dataset

The first step of our approach is transforming the monitoring data into a dataset, which is the input of the estimation algorithms. The class KiekerMonitoringReader from the package `tools.vitruv.applications.pcmjava.modelrefinement.parameters.impl` reads a monitoring file from a specified path and returns monitoring dataset. For the case of external call arguments this is done with the getExternalServiceCalls() method, which returns a ServiceCallDataSet object. Then in the class WekaArgumentsModelEstimation from package `tools.vitruv.applications.pcmjava.modelrefinement.parameters.arguments.impl` the Weka dataset is created with the help of the previously implemented classes WekaDataSet and WekaDataSetBuilder. A new dataset (and estimation model) is created for *every* argument of an external call. The parameters are filtered by type - e.g. the dataset for String parameter would only contain String values. For example, the datasets generated from the code in listing 1.1 are presented in table 2.1,table 2.2 and table 2.3. The last attribute - class always holds the values for the parameter which we try to estimate.

a.VALUE	**b.VALUE**	**class**
a1	b1	a1*5 + Math.pow(b1,3)
a2	b2	a2*5 + Math.pow(b2,3)
...	...	...

Table 2.1: Dataset for the argument of serviceB1()

x.VALUE	**class**
x1	x1
x2	x2
...	...

Table 2.2: Dataset for the argument of serviceB2()

x.VALUE	**class**
x1	!x1
x2	!x2
...	...

Table 2.3: Dataset for the argument of serviceB3()

2.2 Feature selection

For reducing the dimensionality of the problem, or in other words, consider less parameters for a parametric dependency, we use some feature selection algorithms from the Weka library. The filtering of the dataset attributes is done directly in the constructors of the two classes WekaNumericArgumentModel and WekaNominalArgumentModel. For this purpose, we use the two Weka classes ClassifierSubsetEval and GreedyStepwise. The first one evaluates each attribute according to its 'merit' to the final result. The GreedyStepwise algorithm performs a greedy forward or backward search through the space of attribute subsets.

2.3 Initial dependencies

The classes WekaNumericArgumentModel and WekaNominalArgumentModel, implementing the interface ArgumentModel, are responsible for detecting initial dependencies and transforming them into valid stochastic expressions. The Weka algorithms which they adopt are LinearRegression classifier (for numeric parameters) and J48, which generates a pruned or unpruned C4.5 decision tree (for nominal parameters). Figure 2.4 shows a diagram of the implemented package.

2.4 Optimizing with Genetic Programming

After detecting initial parametric dependencies, the applyModel() method of class ArgumentEstimationImpl checks if the error of the initial estimation model is greater or equal to 10%. If this is the case, the optimization is initialized by creating a new instance of class Optimization (optimization is supported only for numeric parameters at this point of the work).
The genetic programming algorithm [4] behind the optimization is from the library Jenetics[1] and makes use of the Java 8 feature - streams. Generations of the evolution are streamed until a certain stop criteria is met. In our implementation we limit the generation by a number (10000) and by fitness threshold of 0.01. The fitness of each individual is judged by the mean square error function.
An individual for our problem domain is a mathematical expression, presented in a tree structure. This mathematical expression consists of genes, which are the "atoms" of the genetic programming approach. Jenetics supports genes, which are modeled like a program (mathematical expression), build upon an Abstract Syntax Tree of functions. The class is called ProgramGene. Figure 2.5 gives an example of such individual, formed by ProgramGene instances. After terminating the evolution process returns the best found individual. This individual is then transformed, so that it matches the stochastic expression syntax.

[1] `https://jenetics.io/`

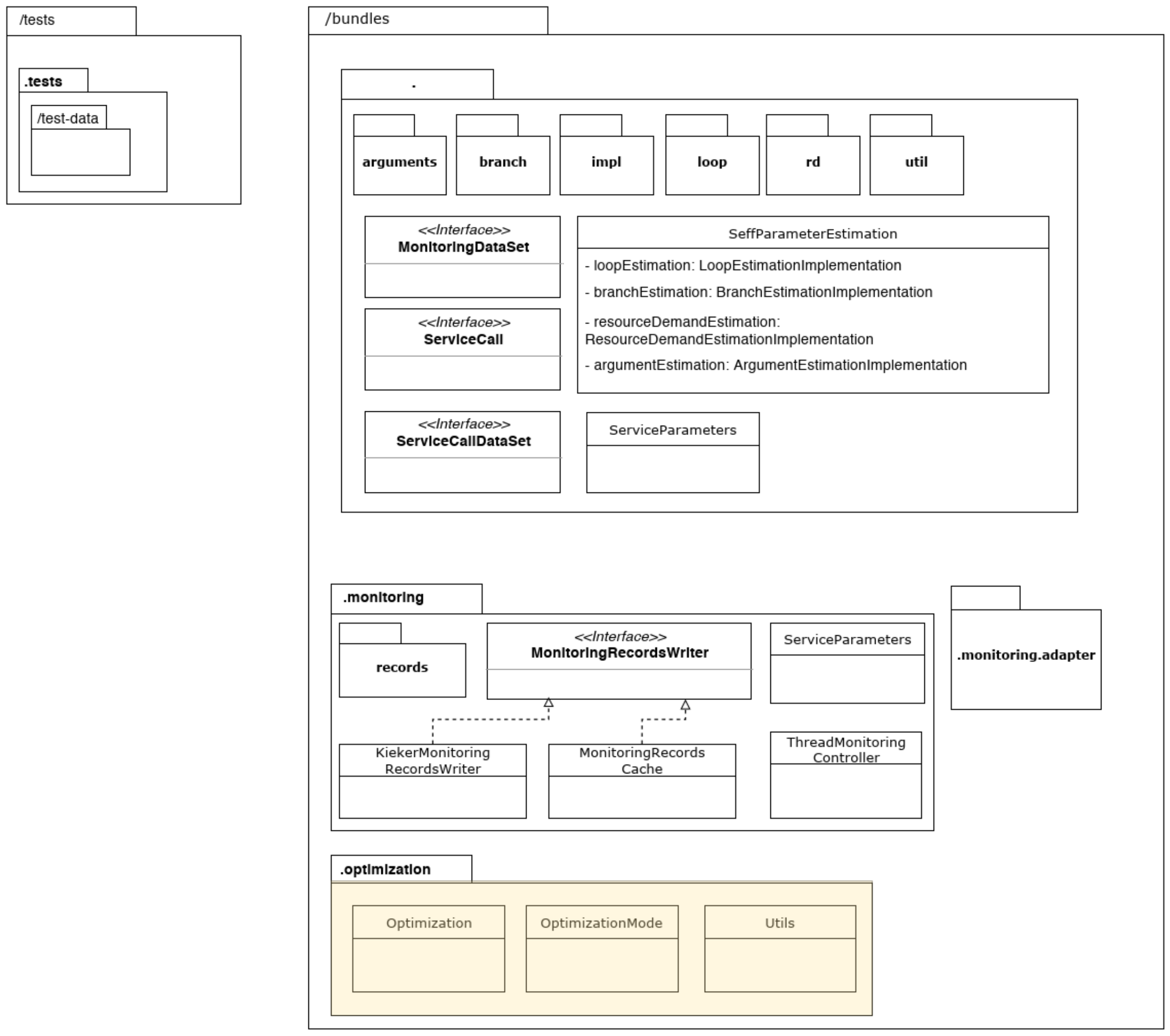

Figure 2.2: UML diagram of the source code (for better overview in the folder `/bundles` the current directory (.) represents `tools.vitruv.applications.pcmjava.modelrefinement.parameters`), yellow-marked packages are contribution

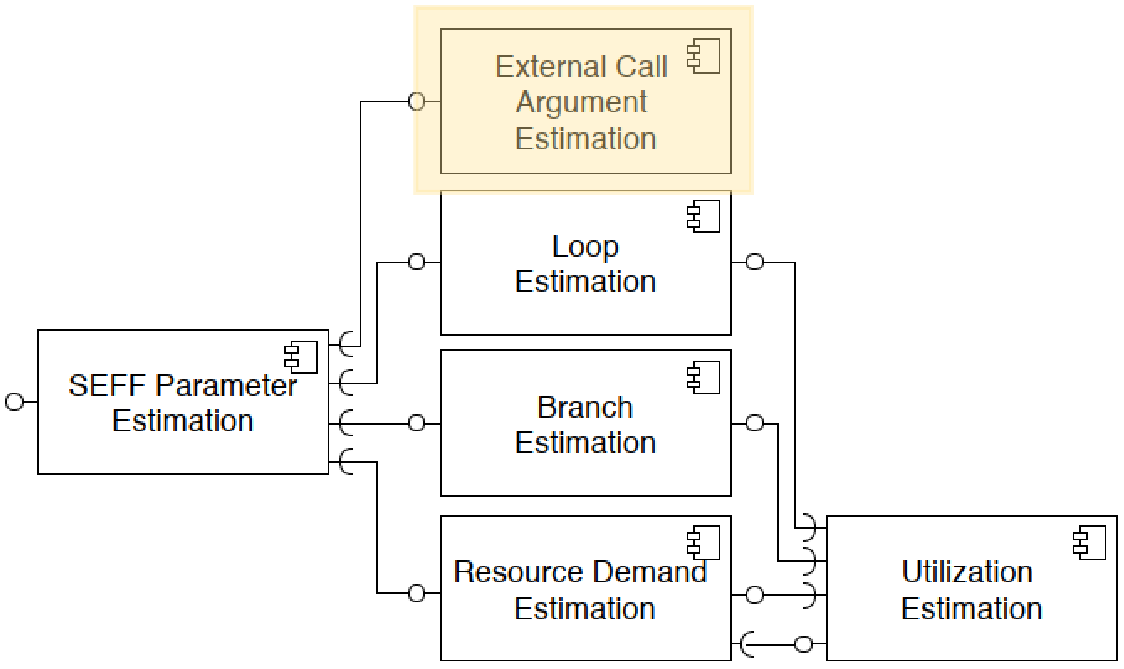
External Call Argument Estimation
Loop Estimation
SEFF Parameter Estimation
Branch Estimation
Resource Demand Estimation
Utilization Estimation

Figure 2.3: The component diagram shows the dependencies between the different components and their interfaces

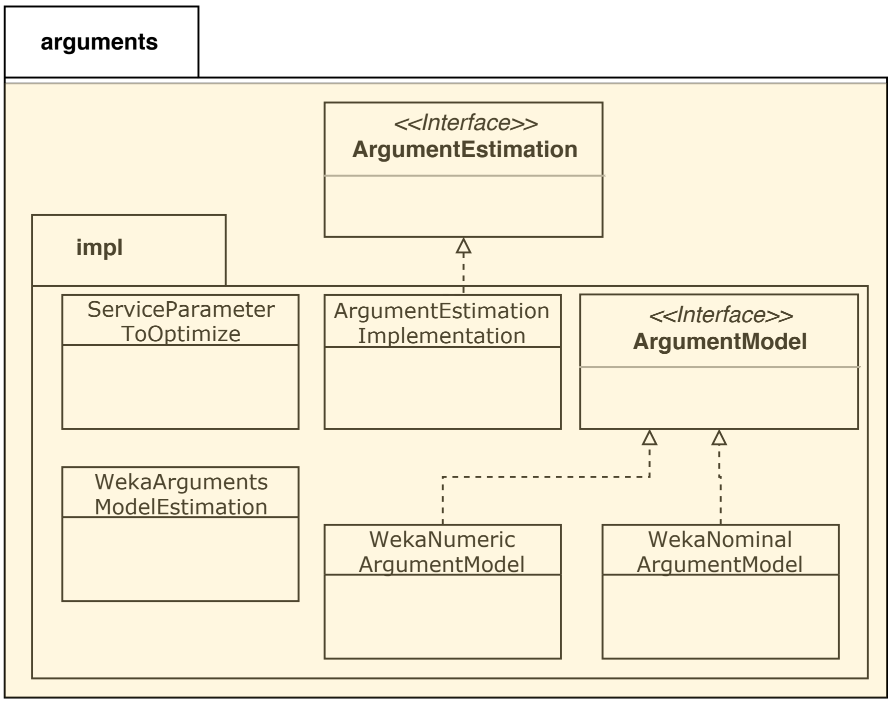
arguments
<<Interface>>
ArgumentEstimation
impl
ServiceParameter ToOptimize
ArgumentEstimation Implementation
<<Interface>>
ArgumentModel
WekaArguments ModelEstimation
WekaNumeric ArgumentModel
WekaNominal ArgumentModel

Figure 2.4: UML diagram of package `tools.vitruv.applications.pcmjava.modelrefinement.parameters.arguments`

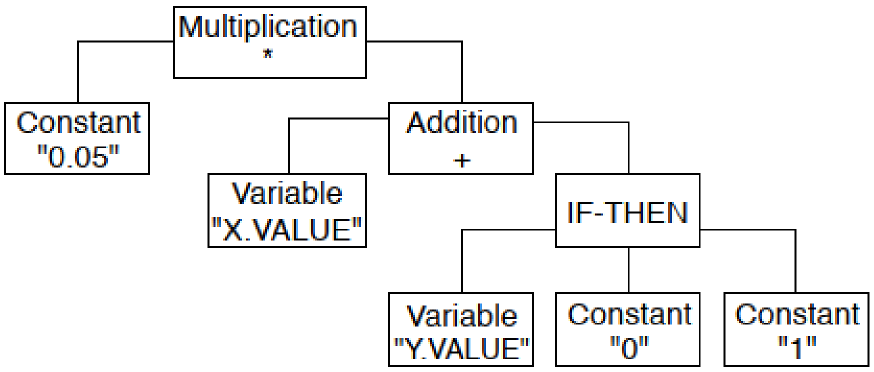

Figure 2.5: Genes representing mathematical expression structured in a tree

Bibliography

[1] Robert Heinrich et al. “The Palladio-bench for Modeling and Simulating Software Architectures”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE ’18. Gothenburg, Sweden: ACM, 2018, pp. 37–40. ISBN: 978-1-4503-5663-3. DOI: `10.1145/3183440.3183474`. URL: `http://doi.acm.org/10.1145/3183440.3183474`.

[2] 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*. ICPE ’12. Boston, Massachusetts, USA: ACM, 2012, pp. 247–248. ISBN: 978-1-4503-1202-8. DOI: `10.1145/2188286.2188326`. URL: `http://doi.acm.org/10.1145/2188286.2188326`.

[3] Jan-Philipp Jägers. “Iterative Performance Model Parameter Estimation Considering Parametric Dependencies”. In: 2018.

[4] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0-262-11170-5.

[5] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp. ISBN: 9780262034760. URL: `http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures`.