Optimizing Parametric Dependencies for Incremental Performance Model Extraction

Sonya Voneva[1], Manar Mazkatli[1], Johannes Grohmann[2], and Anne Koziolek[1]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
uzeci@student.kit.edu, manar.mazkatli@kit.edu, koziolek@kit.edu
[2] University of Würzburg, Würzburg, Germany
johannes.grohmann@uni-wuerzburg.de

Abstract. Model-based performance prediction in agile software development promises to evaluate design alternatives and to reduce the cost of performance tests. To minimize the differences between a real software and its performance model, parametric dependencies are introduced. They express how the so-called performance model parameters (such as loop iteration count, branch transition probabilities, resource demands, and external service call arguments) depend on impacting factors like the input data.

The approaches that perform model-based performance prediction in agile software development have two major shortcomings: they are either costly because they do not update the performance models automatically after each commit, or do not consider more complex parametric dependencies than linear.

This work extends an approach for continuous integration of performance model during agile development. Our extension aims to optimize the learning of parametric dependencies with a genetic programming algorithm to be able to detect non-linear dependencies.

The evaluation results show that using genetic programming enables detecting more complex dependencies and improves the accuracy of the updated performance model.

Keywords: Performance Model (PM) · parametric dependencies · Genetic Programming (GP) · agile development

1 Introduction

When software performance does not meet the predefined requirements, delays, higher costs, and failures on deployment may occur [26]. Thus, the approach of Software Performance Engineering (SPE) is crucial in nowadays software development process. Model-based Performance Prediction (MbPP), first introduced by Smith [22] under the name SPE, aims to avoid potential performance issues using a performance model of the considered system. This allows the reproduction of the time-critical behaviour of a system based on a simulation [19]. PMs allow the developers to judge the quality of their software components and the design alternatives without investing the effort of actually implementing and testing them.

To describe the specific implementation of the components better, the so-called *parametric dependencies* are introduced. They express the relation between input arguments of a service and the Performance Model Parameters (PMPs). The PMPs are represented by abstract source code characterisations like loop iterations count, branch transition probabilities, resource demands, and arguments of external service calls. The parameterization allows answering "what-if"- questions, like MbPP for unseen usage profiles or design alternatives. For example, if we detected that the resource demand of a specific service equals *its input argument * 5*, we can easily simulate the system under new conditions (new input).

One disadvantage of MbPP is that creating a PM and keeping it consistent with the source code during agile software development is a time-consuming task. Until recently, researchers have focused on automating the extraction of PMs, but two main flaws are found in existing works [25,4,15,13,3,23].

- in order to extract the PM after some update in the code, the whole system must be instrumented and run, which causes high monitoring overhead and discards the manual changes that may be applied to the extracted PMs.
- they don't examine how the PMPs depend on input data, i.e. the parametric dependencies, except [13,7].

In the approach, proposed by Mazkatli et al. [17], Continuous Integration of Performance Model (CIPM), both issues are addressed by incremental extraction and calibration of PMs with parametric dependencies.

The incremental calibration of CIPM [18] covers, however, only linear dependencies. This work extends CIPM by (1) advanced estimation of the external calls' arguments, considering the parametric dependencies and (2) by optimizing all the detected dependencies using a genetic algorithm. For goal (1), we filter the dependency candidates by applying feature selection. We furthermore search for a dependency not only to the input arguments of a service, but, considering the data flow, to the return values of the previous external calls.

This paper is structured as follows: section 2 gives an overview of the backgrounds of our work, section 3 presents a code example to clarify the definition of parametric dependencies. In section 4 we elaborate on the specific steps of our approach. Section 5 covers the evaluation part of the work. In section 6 the related work in the scientific field is discussed. Finally, section 7 concludes the paper and suggests some future work.

2 Foundations

This chapter contains the foundations of our approach. We discuss the different tools, libraries and algorithms, involved in the process.

2.1 Palladio

Palladio is an approach to model and simulate architecture-level PMs. Within Palladio, the so-called Palladio Component Model (PCM) defines a language for describing PMs: the static structure of the software (e.g. components and interfaces), the behavior, the required resource environment, the allocation of software components, and the usage profile.

The PCM Service Effect Specification (SEFF) [20] describes the behavior of a component service on an abstract level using different control flow elements: internal actions (a combination of internal computations that do not include calls to required services), external call actions (calls to required services), loops, and branch actions. SEFF loops and branch actions include at least one external call, otherwise they are merged into the internal actions to increase the level of abstraction. To predict the performance measures (response times, central processing unit (CPU) utilization, and throughput) the architects have to enrich the SEFFs with PMPs. Examples of PMPs are resource demands (processing amount that internal action requests from a certain active resource, such as a CPU or hard disk), the probability of selecting a branch, the number of loop iterations, and the arguments of external calls.

Palladio uses the stochastic expression (StoEx) language to define PMPs as expressions that contain random variables or empirical distributions. StoEx allows to refer to variable properties (e.g. `NUMBER_OF_ELEMENTS, VALUE, BYTESIZE`, and `TYPE`). StoEx also supports calculations (e.g. `5*file.BYTESIZE`) and comparisons (e.g. `(x.VALUE > 8) ? 1 : 2`) [20].

2.2 Kieker

Kieker [9] is an extensible open-source application performance management tool, which allows capturing, analyzing and visualizing execution traces of source code. Monitoring probes are inserted into the source code without modifying it. They can be predefined and customized or dynamic and adaptive. We use Kieker with manually instrumented code to store monitoring records. For defining the structure of the records we use the Instrumentation Record Language (IRL) [10].

2.3 Algorithms

For detecting initial parametric dependencies we integrated two Machine Learning (ML) algorithms from the Java library Weka [8]. Linear regression is used for estimating dependencies which consists only of numeric values. Decision tree is adopted for all dependencies which contain numeric and nominal values.

For refining the initial dependencies, Genetic Programming (GP) [12] was applied. It is a meta-heuristic machine learning technique which, inspired by the Darwinian principle of survival and evolution of the fittest, finds an optimal solution to a search problem. The definition of optimal is according to a predefined *fitness function*. Each potential solution is referred to as an *individual*. Furthermore, individuals consist of *genes*.

GP is a special kind of genetic algorithm with genes, forming a tree structure. The original idea of GP was to output automatically generated source code to a specified problem as an input. Nowadays, GP has become a widely spread optimization method.

In the following, the most important elements of GP will be described. A *gene repository* stores the genes, which itself is a base for creating a *chromosome repository*. The chromosome repository keeps all chromosomes. A chromosome is a potential solution of the problem, whereas the genes are the particles, of which that solution is composed. A set of chromosomes is called a *generation*.

A typical GP approach consists of multiple steps, which are repeated in many iterations. In the first iteration an initial generation is created from individuals in the chromosome repository. Next, the *crossover* and *mutation* take place. The process of crossover is analogous to biological crossover in human reproduction - parent chromosomes are recombined to form new children. Mutation is simply changing one or multiple genes of a chromosome to ensure genetic diversity.

The fitness function determines how "good" / "fit" an individual is. In order to define the fitness of an individual, domain expertise on properties of the expected optimal solution is required.

2.4 Continuous Integration of Performance Model

Continuous Integration of Performance Model (CIPM) is an approach to automatically keep the architectural PM consistent during the agile software development [17]. Its idea is to respond to the changes in source code by updating the PM using predefined consistence rules [14] that transform the changes using change-based model-based transformations. Then, CIPM instruments only the changed parts of source code to provide the required monitoring data to calibrate the new/updated parts of PMs. After executing the source code, CIPM analyses the generated monitoring data to calibrate PMs incrementally [11]. The incremental calibration [18] estimates the missing PMPs considering the parametric (linear) dependencies and updates the deployment and usage parts of PMs to respond to the potential changes in deployment or usage profile. Finally, CIPM validates the updated PMs by starting the simulation and calculating the variation between the monitoring data and the simulation results to show the estimation error.

3 Parametric Dependencies Example

To illustrate the meaning of a parametric dependency, listing 1.1 will be examined. In this code piece, we have two components - `A` and `B`. The presented method from component `A` - `serviceA()` calls three services from its external component - `B`. This means that component `A` is the *requiring* component and component `B` is the *providing* component. As one can notice, the branch transition and the arguments of the external service calls depend on the arguments of `serviceA()`.

The external calls in this scenario are the calls to `serviceB1(), serviceB2()` and `serviceB3()`. The dependencies, which we try to estimate are between each external call argument and the corresponding candidates for a dependency from the arguments of `serviceA()` or the data flow like the list `result`. The candidates for a dependency can be arguments from the same data type (as the external call argument) or arguments which have a characteristic from the same data type. These candidates are used to build a dataset which is the base of our ML algorithms. The dependencies are in the form of a *StoEx* (see section 2.1).

```java
public class A {

    private B componentB;

    public void serviceA(int x, int y, boolean b){
        /* Some internal action */
        if(b){
            /* Some internal action */
            List<Integer> result = componentB.serviceB1(4*y);
            componentB.serviceB2(Math.pow(x,2) + y);
            componentB.serviceB3(result.size());
        }
        ...
    }
}
```

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

4 Approach

The proposed approach is part of the vision described by Mazkatli and Koziolek [17], see section section 2.4. They describe a tool which automatically updates a PM, represented as PCM, from iterative source code changes. The incremental calibration [18] enriches the extracted PM with parametric dependencies of the form:

$$D_i(P) = (a * p_0 + b * p_1 + ... + z * p_n + C) \quad (1)$$

where $p_0, p_1..p_n$ are numeric service arguments or numeric attributes of the caller's arguments. $a..z$ are the weights of the input arguments and C is a constant. This work aims to additionally detect non-linear parametric dependencies for external call arguments and for all types of PMPs and to refine the linear dependencies.

In the following, we present an overview of our workflow (cf. fig. 1).

Preprocessing We begin by applying some heuristics, similarly to [13], before monitoring the source code. The point is to reduce the monitoring overhead

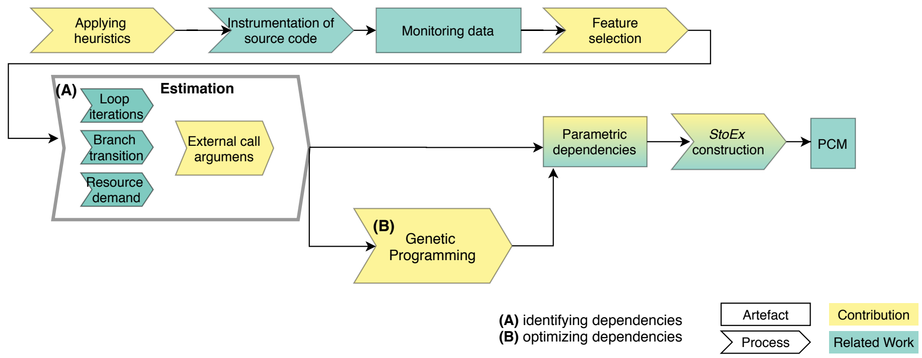

Fig. 1. Workflow of our approach

by recording only performance-relevant information. For example, if we have an input argument which has the type `List<T>`, we may not be interested in its specific elements. Therefore, we monitor only its size. In our approach, we defined which characteristics should be monitored for every data type which is handled.

Afterwards, the source code is instrumented using the framework Kieker, see section 2.2, similarly to [18].

The collected monitoring data is one of the inputs needed for our dependency estimation approach. The other input is the PCM of the system. We can easily differentiate between the records of the PMPs, because Kieker stores them as separate types. We have monitoring record types for loop, branch, internal action demanding a resource, and an external call action. For example, a monitoring record for the latter contains information like external action id, service execution id, caller id, caller execution id, input parameters, return value, entry and exit timestamps. More information on this can be found in this Bachelor's thesis [24].

Feature Selection and ML Models The monitoring records are then converted to datasets (for each PMP a separate one), which are valid as inputs for the algorithms of Weka [8]. We use this library for feature selection and then creating an estimation model for each PMP. We filter the dataset to remove all attributes which do not have an impact on the prediction quality. For judging this, the `ClassifierSubsetEval` class was chosen, which evaluates attribute subsets on training data. It uses a classifier to estimate the 'merit' of a set of attributes. In our case the classifiers are `LinearRegression` - for numeric values only, and `J48` - a decision tree, implementing the C4.5 algorithm ([21]), for both nominal and numeric values. The evaluator also needs a specified search technique. Our choice - `BestFirst` performs greedy hill climbing with backtracking; one can specify how many consecutive non-improving nodes must be encountered before the system backtracks. We defined the search to be bidirectional. After reducing the datasets, we can instantiate our classifiers. As the workflow shows, the construction of estimation models for the loops, branches and internal actions, demanding some resource, was already implemented. To generate the *StoEx*, we parse the classifier output (coefficients) and build a string from it.

Optimizing The major part of our approach is improving the linear dependencies detected with the incremental calibration [18]. The dependencies are refined only if the mean squared error, that they produce, is bigger than 0.1. The optimization is conducted according to the GP algorithm presented in section 2.3. In order to reduce the time needed by the algorithm to produce a solution, we set the output of the above-mentioned ML algorithms as an initial parametric dependency (starting point of the genetic evolution).

Similarly to the approach of Krogmann et al. [13], we model genes as mathematical functions to express more complex dependencies. Figure 2 depicts an example of a gene. This is very beneficial for our approach since both, the *Abstract Syntax Tree (AST)* of the *StoEx* language and the genes of GP have tree structures and we are able to easily transform the initial *StoEx* into a starting individual for the GP.

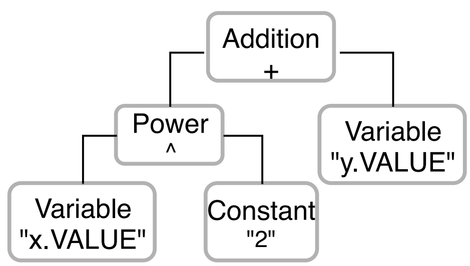

Fig. 2. Tree representation of the individual$Individual_A = x^2 + y$

Another worth-mentioning feature of the GP is the *fitness function*. In our implementation the fitness of an individual (mathematical expression) is judged according to its complexity (depth of AST) and prediction accuracy (mean squared error). Moreover, each algorithm run (evolution) is restricted by a maximum run time and a maximum number of generations - these limits are implemented as parameters of the algorithm.

In our work, we used the Jenetics library[3], written in Java, which provides a GP implementation. In contrast to other GP implementations, Jenetics uses the concept of an evolution stream for executing the evolution steps. Therefore, it is no longer necessary to perform the evolution steps in an imperative way.

The final step of the workflow is constructing the *StoEx* - this involves some string processing. Then, the *StoEx* is inserted in the PCM at the right place and as an output of our approach we deliver the PCM, enriched with the optimized parametric dependencies.

[3] `https://jenetics.io/manual/manual-5.1.0.pdf`

5 Evaluation

Our evaluation is twofold. First, we judge the importance of feature selection. Second, we evaluate the optimization using GP algorithm. In the first part, we compare the accuracy and the complexity of the estimated dependencies when feature selection is used and when not. The results do not show a significant impact of using the feature selection for numerical variables in contrast to using it for nominal ones [24]. Therefore, due to lack of space we focus in this paper on the second part of the evaluation to show the most representative results.

5.1 Goal and Scenario

Our main research question is: *which PM is more accurate - with GP optimization or without?* To answer this question, we calibrate three different PMs: one with our approach, one considering only linear dependencies and the last one - without any parametric dependencies. For the calibration, we use a monitoring data generated by a usage profile P1. Then, we use the three models to predict the performance for unforeseen usage profile P2. To compare the prediction power of these PMs, we compare the predicted response times by the simulations with the actual response times that we can measure for P2. Both response times are distributions, therefore we use the following metrics to compare the similarity: Kolmogorov-Smirnov-Test (KS-Test) [6] that tests whether two empirical distributions come from the same underlying distribution, the Wasserstein metric [16] that quantifies the effort needed to transfer one distribution into the other, and conventional statistical measures. For both KS-Test and Wasserstein, the lower the value is, the higher is the accuracy of PM.

5.2 Setup

To answer the research question above we implemented an artificial example - a small application with focus on the external service calls with complex dependencies. The most important components of the micro-system are:

`class A` contains our target method for incremental calibration of the PCM - `serviceA()`. Its first part is shown in listing 1.1. The rest of the method consists of a loop and some other external service calls. This method has three arguments - `int x, int y`, and `boolean b`. The component `class A` has the PCM role of a *requiring* component - this means it requires some external services.

`class B` encapsulates six methods which are called by `class A`. So `class B` has the PCM role of a *providing* component. The arguments of the called methods in `class B` each have a dependency to the service arguments of `class A`.

First, we apply a fine-grained monitoring using the following usage profile P1 to generate the required monitoring data for the calibration. For this, we ran `serviceA()` 500 times with ten simulated concurrent users. We choose the arguments `x, y` and `b` as follows: random integer from the set [0..9], random integer from the set [1, ..10], and random boolean. After this, we calibrate three different PMs - one only with distribution functions of the estimated PMPs, one after learning the linear dependencies as described in [18] and one with learning more complex (optimized) dependencies as described in this paper.

Then, we start the simulation using the three PMs to predict the response time of `serviceA()` for the unforeseen usage profile (P2): i.e., changing the `x` parameter to a random integer from the set [0..19], `y` parameter to random integer from the set $[1, ..20]$, and `b` to random boolean. We repeat the simulation 50 times for each PM to make the results more representative.

As a reference, we monitor `serviceA()` coarse-grained for the usage profile (P2), to create a validation set for the evaluation. Coarse-grained monitoring records the entry and exit times without the unnecessary monitoring overhead. Finally, we compare the simulation results with the actual monitoring data using the metrics defined in section 5.1.

5.3 Results

Table 1 presents a comparison between the response times of `serviceA()` over 50 iterations according to the monitoring data and to the three PMs simulations. From each distribution, the quartiles, as well as the minimum, maximum, and average values are calculated. As the table shows, the performance prediction of the PM that is calibrated with our approach - optimizing parametric dependencies, is the closest prediction to the actual monitoring response time in comparison to the prediction of other PMs: PM that is calibrated with linear dependencies and PM that is distribution functions, where parametric dependencies for external service calls are not handled at all.

Distribution	**Min**	**Q1**	**Q2**	**Q3**	**Max**	**Avg**
Monitoring	0.009	0.217	0.59	1.318	2.589	0.825
Distribution functions	**0.021**	1.576	2.857	4.369	7.151	3.045
Linear functions	0.025	1.643	2.904	4.546	7.082	3.078
Optimized	0.111	**0.676**	**1.222**	**1.676**	**2.232**	**1.199**

Table 1. Response times (in seconds) of the three PMs: first - parameterized only with distribution functions for the external call arguments, then - only with linear dependencies for all PMPs and finally - with more complex dependencies for all PMPs.

In table 2, again the simulations of the PMs are compared, but this time with different metrics - KS-Test [6] and Wasserstein [16]. As the numbers from table 2 indicate, the **Optimized** PM improves the KS-Test value by **0.302** and the Wasserstein value by **1.255** on average. This improvement is roughly **two** times for the KS-Test value and **five** times for the Wasserstein value. These results confirm that the **Optimized** PM has the highest similarity to the actual system.

Metric	Distribution functions	Linear functions	Optimized
KS Q1	0.585	0.581	**0.278**
KS Avg.	0.595	0.592	**0.293**
KS Q3	0.608	0.601	**0.306**
WS Q1	1.527	1.535	**0.292**
WS Avg.	1.568	1.56	**0.313**
WS Q3	1.609	1.591	**0.339**

Table 2. Comparison of the metrics KS-Test and Wasserstein of the three models: first - parameterized only with distribution functions for the external call arguments, then - only with linear dependencies for all PMPs and finally - with more complex dependencies for all PMPs.

6 Related Work

Various approaches for extracting an architectural model based on static (e.g. [2,14]), dynamic (e.g. [4,23,3]), or hybrid analysis (e.g. [13]) exist. In comparison to our approach, the aforementioned approaches require monitoring overhead to extract consistent PMs during agile software development and do not keep the previous potential manual changes to PMs. Similarly to the approach of Krogmann et al. [13], we use GP to detect the parametric dependencies. In contrast to their work, we use the GP during an incremental calibration of PMs. This reduces the required overhead by GP to learn the dependencies, because our approach uses GP only to optimize the PMPs that have been changed in the recent development iteration and have a high cross-validation error by the used initial ML algorithms.

The following works also consider parametric dependencies. Grohmann et al. [7] introduce an approach to identify and to characterize [1] parametric dependencies for PMs using monitoring data from a running system. This monitoring data is then analyzed and correlations between different parameters are identified with the use of different feature selection approaches from the area of the ML. This approach does not represent the parametric dependencies as *StoEx* or support the iterative updates to PM. Courtois et al. [5] use multivariate adaptive regression splines to extract parametric dependencies. They perform dedicated performance tests to obtain the data on which they fit the regression splines. This approach also lacks the incremental fashion of PM construction.

7 Conclusion and Future Work

The contribution of this work is twofold. First, we presented an approach for the incremental estimation of external calls' arguments for CIPM, considering parametric dependencies. For this, we apply some feature selection algorithms to reduce the number of candidates for the proposed ML algorithms that identify (initial) dependencies.

The second part of our work is the optimization of the parametric dependencies for all types of PMPs using a GP algorithm, which refines the outputs of the ML algorithms (*PMPs as StoEx*) and eventually finds more complex dependencies than linear. To sum up, the implemented mechanism needs two inputs - a PCM and monitoring data from instrumented source code. The output of our algorithms are the *optimized PMPs as StoEx*, which are inserted in the PCM, so that at the end we enriched a PCM with parametric dependencies.

To evaluate the implemented technique, we simulated three different PMs, parameterized in different ways, and compared between the response time of our target service according to the monitoring records and the simulated response times. The results show that the PM with optimized parametric dependencies has an accuracy of **two** times (Kolmogorov-Smirnov-Test value) and **five** times (Wasserstein metric) higher than a PM with linear or distribution functions only. This confirms that the optimization improves the accuracy of the PM.

Our approach promises to detect more complex dependencies during the incremental calibration to improve the accuracy of the iteratively updated PMs. Therefore, we plan to integrate our implementation with the implemented pipeline, proposed in [18], and to perform further evaluation using different case studies.

In future works we aim to develop an optimization mechanism which handles the dependencies of the nominal arguments as well, as our implementation lacks this feature. Additionally, we aim to extend our approach to detect the dependencies to the service arguments of composite data types. One idea in this direction is traversing all fields of the composite argument until reaching primitive ones.

References

1. Ackermann, V.: Blackbox learning of parametric dependencies for performance-models from monitoring data (2018)
2. Becker, S., Hauck, M., Trifu, M., Krogmann, K., Kofron, J.: Reverse engineering component models for quality predictions. pp. 194–197 (03 2010)
3. Brosig, F., Huber, N., Kounev, S.: Automated extraction of architecture-level performance models of distributed component-based systems. In: Proceedings of the 2011 26th IEEE/ACM Intl. Conference on Automated Software Engineering. p. 183–192. IEEE Computer Society (2011)
4. Brunnert, A., Vögele, C., Krcmar, H.: Automatic performance model generation for java enterprise edition (ee) applications. pp. 74–88 (09 2013). https://doi.org/10.1007/978-3-642-40725-3-7
5. Courtois, M., Woodside, M.: Using regression splines for software performance analysis. In: Proceedings of the 2nd Intl. Workshop on Software and Performance. pp. 105–114 (2000)
6. Dodge, Y.: The Concise Encyclopedia of Statistics, chap. Kolmogorov–Smirnov Test, pp. 283–287. Springer New York (2008)
7. Grohmann, J., Eismann, S., Elflein, S., von Kistowski, J., Kounev, S., Mazkatli, M.: Detecting parametric dependencies for performance models using feature selection techniques (2019)

8. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explor. Newsl. **11**(1) (2009)
9. van Hoorn, A., Waller, J., Hasselbring, W.: Kieker: A framework for application performance monitoring and dynamic software analysis. In: Proceedings of the 3rd ACM/SPEC Intl. Conference on Performance Engineering. ICPE '12 (2012)
10. Jung, R.: An instrumentation record language for kieker. Tech. rep., Tech. rep. Kiel University (2013). https://doi.org/10.13140/RG.2.1.3655.5689
11. Jägers, J.P.: Iterative performance model parameter estimation considering parametric dependencies (2018)
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
13. Krogmann, K., Kuperberg, M., Reussner, R.: Using genetic search for reverse engineering of parametric behavior models for performance prediction. IEEE Trans. Software Eng. **36**, 865–877 (2010). https://doi.org/10.1109/TSE.2010.69
14. Langhammer, M.: Automated Coevolution of Source Code and Software Architecture Models. Ph.D. thesis, Karlsruhe Institute of Technology, Karlsruhe, Germany (2017)
15. Langhammer, M., Shahbazian, A., Medvidovic, N., Reussner, R.H.: Automated extraction of rich software models from limited system information. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA). IEEE (2016)
16. Majewski, S., Ciach, M., Startek, M., Niemyska, W., Miasojedow, B., Gambin, A.: The wasserstein distance as a dissimilarity measure for mass spectra with application to spectral deconvolution (2018)
17. Mazkatli, M., Koziolek, A.: Continuous integration of performance model. pp. 153–158 (04 2018). https://doi.org/10.1145/3185768.3186285
18. Mazkatli, M., Monschein, D., Grohmann, J., Koziolek, A.: Incremental calibration of architectural performance models with parametric dependencies. In: IEEE Intl. Conference on Software Architecture (ICSA 2020) (2020)
19. Pooley, R.: Software Engineering and Performance: A Road-map. ICSE-Future of SE Track pp. 189–199 (2000)
20. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K.: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press (2016)
21. Ruggieri, S.: Efficient c4.5. IEEE Trans. on Knowl. and Data Eng. **14**(2), 438–444 (2002). https://doi.org/10.1109/69.991727
22. Smith, C.U.: Performance Engineering of Software Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edn. (1990)
23. Spinner, S., Walter, J., Kounev, S.: A reference architecture for online performance model extraction in virtualized environments. In: Companion Publication for ACM/SPEC on Intl. Conference on Performance Engineering. p. 57–62. Association for Computing Machinery, New York, NY, USA (2016)
24. Voneva, S.: Optimizing parametric dependencies for performance model extraction. bachelor's thesis (2020), `https://sdqweb.ipd.kit.edu/publications/pdfs/Voneva20a.pdf`
25. Walter, J., Stier, C., Koziolek, H., Kounev, S.: An expandable extraction framework for architectural performance models. In: Proceedings of the 8th ACM/SPEC on Intl. Conference on Performance Engineering Companion. pp. 165–170. ICPE '17 Companion, ACM, New York, NY, USA (2017)
26. Woodside, M., Franks, G., Petriu, D.: The future of software performance engineering. pp. 171–187 (06 2007). https://doi.org/10.1109/FOSE.2007.32