Reverse Engineering of Parametric Behavioural Service Performance Models from Black-Box Components

Klaus Krogmann, Michael Kuperberg, and Ralf Reussner
Institute for Program Structures and Data Organisation
Universität Karlsruhe (TH)

{krogmann,mkuper,reussner}@ipd.uka.de

Abstract: Integrating heterogeneous software systems becomes increasingly important. It requires combining existing components to form new applications. Such new applications are required to satisfy non-functional properties, such as performance. Design-time performance prediction of new applications built from existing components helps to compare design decisions before actually implementing them to the full, avoiding costly prototype and glue code creation. But design-time performance prediction requires understanding and modeling of data flow and control flow accross component boundaries, which is not given for most black-box components. If, for example one component processes and forwards files to other components, this effect should be an explicit model parameter to correctly capture its performance impact. This impact should also be parameterised over data, but no reverse engineering approach exists to recover such dependencies. In this paper, we present an approach that allows reverse engineering of such behavioural models, which is applicable for black-box components. By runtime monitoring and application of genetic programming, we recover functional dependencies in code, which then are expressed as parameterisation in the output model. We successfully validated our approach in a case study on a file sharing application, showing that all dependencies could correctly be reverse engineered from black-box components.

1 Introduction

Nowadays, more and more applications integrate existing software systems. New software components are therefore combined with legacy systems or existing components from code repositories to form new applications. Predicting the performance of such applications is inherently complex as the performance of legacy applications or reused parts of existing components in a *new environment* (used by the new application) is not known beforehand.

Thus, it is desirable to have a performance prediction approach allowing to estimate the performance of the new application before implementing it to avoid high risks of inadequate performance. Predictions based on models are one solution to this problem. Accordingly, the performance impact of utilised components needs to be made explicit in the model to reflect the different usage contexts of existing components. If an existing component from a repository is going to be reused, it should be possible to estimate its

performance impacts in the new context without implementing the new application.

Some parametric performance models exist that explicitly parameterise over the usage context to allow model reuse (e.g. [**?**, **?**, **?**]) for performance predictions. However, currently no automated approach exists to reverse engineer such parametric behavioural performance models from existing code that explicitly state context dependencies. Instead, they need to be reverse engineered manually.

In this paper we present a reverse engineering approach for parametric behavioural service performance models making the context dependencies explicit. Therefore, the reverse engineered service models capture control and data flow parameterised over service argument data ("method arguments"). Our reverse engineering approach uses monitoring to capture runtime effects at the service interface level. Based on this monitoring, genetic programming is used to find functional dependencies (for example how often a required service is called and which arguments are used for the call, depending on the input data of this service).

We successfully validated our approach in a case study on a component-based file sharing application by comparing (partially) automated reverse engineered results from our approach with those manually recovered.

The contribution of this paper is a reverse engineering approach for behavioural models of component services. By monitoring call frequencies and data characteristics at the interface level, the black-box properties of components is preserved. Machine learning (genetic programming) enables automatically recovering functional dependencies from the monitoring data, leading to a behavioural model of component services whose control and data flow is explicitly parameterised to express dependencies between provided and required services.

The rest of this paper is structured as follows: In Section 2 we briefly talk about foundations of this work, before going into related work in Section 3. Section 4 then presents our approach, which is validated in Section 5. Section 6 lists assumptions and limitations before Section 7 concludes this paper.

2 Foundations

Terms. In the rest of this paper we will refer to the term "service" as something provided by software components. It has a signature like `void compressFile(byte[] file)`. The term "external service" will refer to component services implemented by another component, from the perspective of a considered service.

The "assembly context" expresses which components are connected to a component, for example which component is called if a required service is invoked. The "usage context" declares which other components use a provided component service and how users interact with a system (e. g. using large or small file for a upload service). The "deployment context" defines the execution environment (application server, virtual machine, operating system, hardware) a component is executed on. All contexts are not known a-priori at design time (or for a design time model) and thus need to be made explicit to allow model reuse.

Performance Model. In this paper, we explicitly deal with scenarios where these contexts change. The Palladio Component Model (PCM, [**?**]; a performance model) supports these requirements of third-party reuse (the model for each component is created once, independent from the context). Hence, we use it as the target model for reverse engineering in this paper. Within the behavioural service model of the PCM, a control and data flow abstraction parameterised over input data is supported for every provided service of a component. In line with the PCM, we focus on performance-relevant *abstractions* of data and control flow. Thus, we are not investigating performance itself but parametric dependencies that have influence on performance.

The behavioural model of the PCM supports scenarios where performance impacts are propagated through an architecture. For example, if the load of a load balancer increases, following components will have a higher load likewise; if a large amount of data is passed from one to another component following component need to handle increased amounts of data; or if parameters of one service decide on which following services are selected (due to a strategy or for delegating work) it depends on the selection which following component has an increased load – this is captured by the PCM's behavioural service model.

Genetic Programming. Genetic programming, which will be used in this paper as a machine learning approach (cf. [**?**]), is a kind of genetic algorithm, where chromosomes have a tree structure (like an abstract syntax tree) instead of a linear form. Genetic algorithms are used for optimisation problems as a meta-heuristic. Solution candidates are represented by a chromosome (an individual), which itself consists of multiple genes. Each gene is a building block (in the case of genetic programming for example an `if` statement). Iterating for multiple generations, genetic algorithms optimise a set of chromosomes using crossing and mutation of genes and a selection strategy of "fittest" individuals. The fitness of an individual is judged by a fitness function, which leads the optimisation. Koza [**?**] provides more background on genetic programming.

Genetic algorithms usually involve non-deterministic behaviour by using random initialisation, random mutation, and random crossover. This will be important in the validation section, as one run of the genetic algorithm is not indentical to another one. Thus, the results and convergence speed vary from run to run.

The way genetic programming is used in this paper, it can be considered as a data mining approach (see for example [**?**]). In this paper, genetic programming is used for finding parametric dependencies in monitoring data, though the origins of genetic programming lie in finding program code that performs a given task. Due to the development in both fields and the resulting variety of data mining algorithms and variations of genetic programming, a strict borderline between them is hard to draw.

3 Related Work

Work in this paper is related to reverse engineering of software performance models, component-based software engineering, and search-based software engineering.

Reverse engineering of performance models using traces is performed by Hrischuk et al. [**?**] in the scope of "Trace-Based Load Characterisation" (TLC). In practice, such traces are difficult to obtain and require costly graph transformation before use. The target model of TLC is not component-based.

Trace data is used by Israr et al. in [**?**] to determine the "effective" architecture of a software system. Using pattern matching, this approach can differentiate between asynchronous, blocking synchronous, and forwarding communication. Similar to our approach, Israr et al. support components but have no explicit control flow, yet they do not support inter-component data flow and do not support internal parallelism in component execution as opposed to the approach presented in this paper. As in TLC, Israr et al. use Layered Queueing Networks (LQNs) as the target performance model.

Search-based software engineering. Search-based approaches such as simulated annealing, genetic algorithms, and genetic programming have been widely used in software engineering [**?**]. However, these approaches have not been applied to reverse engineering, but to problems like finding concept boundaries, software modularisation, or testing.

"Classical" approaches like regression splines are used by Courtois et al. in [**?**] to recognise input parameter dependencies in code. Their iterative approach requires no source code analysis and handles multiple dimensions of input, as does the approach described by us. However, the output of the approach in [**?**] are polynomial functions that approximate the behaviour of code, but which are not helpful in finding non-continuity in component behaviour. The approach is fully automated, but assumes fixed external dependencies of software modules and fixed hardware.

Dynamic program analysis. Denker et al. [**?**] provide a short overview on methods for dynamic analysis of programs and discuss typical problems of dynamic analysis techniques. They point out the frequent re-implementation of analysis approaches in different flavours; specific to a certain need. Briand et al. focus on reverse engineering of behavioural models. They reverse engineer UML sequence diagrams [**?**] from running applications. For that, they apply a distributed monitoring approach that is able to deal with concurrency [**?**] which is comparable to our approach but does not capture data flow information. Systä [**?**] deals with reverse engineering of the behaviour of Java programs by combining static and dynamic analysis. In [**?**] reverse engineering is based on event trace information, without considering data flow and dependencies in code.

Component-based software engineering. The reverse engineering target of our approach are parameterised component behaviour models that allow capturing the effect of load and parameter propagation through a component-based architecture. Hamlet et al. [**?**] handle parameter propagation, but use a very restricted component model. They do not parameterise resource demands, while our approach focuses on load and parameter propagation, only. The Robocop component model [**?**] targets embedded low-level services. Bondarev et al. [**?**] extend Robocop by a specification language for parameterisation which allows parameter for required services and resource usage. Here, parameter values can be

only constants, which limits expressiveness. Parameterisation introduced by Eskenazi et al. [**?**] is simplified and has only limited support for reusability.

4 Reverse Engineering

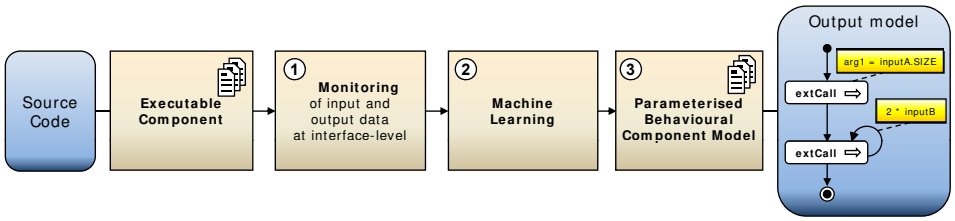

Figure 1: Reverse Engineering Process

The reverse engineering process (see Fig. 1) creates a parametric behavioural component service model ("service model" for short) from source code by monitoring executed services. The process starts with executing component services under investigation in a testbed (which can be a running application). Services are executed using representative workloads and their execution behaviour is monitored ① at the interfaces. For example, call frequencies of required services and characterisations of parameters (e.g. values, size) are monitored. In the second step ② , a machine learning approach (in our case Genetic Programming [**?**]) is used to find functional dependencies in the monitoring data. This might, for example be "call an external method x for each element of input parameter collection c".

The third step ③ creates the final parameterised model describing the behaviour of a component service. The control and data flow described by the model are parameterised over input data. A more detailed example of such a service behaviour model follows below.

4.1 Example

The example from Figure 2 shows the BusinessLogic component, which will also be used in the validation in Section 5. The component provides the service `uploadFiles` for uploading a number files to a web-based service. Files then might be shared with other users. In the simplified version shown, there are two required services: `storeLargeFile` for storing large and `storeSmallFile` for storing small files, responsible for persisting file data.

Users upload a number of files which then are stored either via the `storeLargeFile` service if a certain threshold (e.g. 1024 KB) is passed or via `storeSmallFile` otherwise.

Thus, for the simplified example an ideal behavioural service model would express the following functional dependencies:

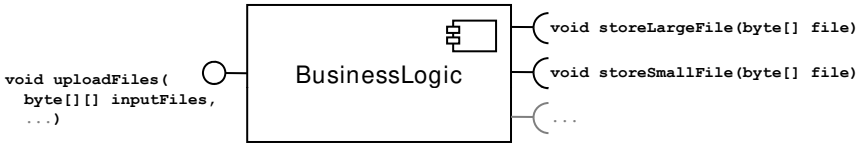

Figure 2: Simplified BusinessLogic Component; some required services are left out

- for each element of the method argument `uploadFiles[x][]` ("length") either `storeLargeFile` or `storeSmallFile` are called,
- the size of the files passed to `storeLargeFile`/`storeSmallFile` corresponds to the size of elements in `uploadFiles[][x]`,
- `storeLargeFile` is only called of files are larger than 1024 KB (in this example); otherwise `storeSmallFile` is called.

4.2 Step 1: Component service execution and monitoring

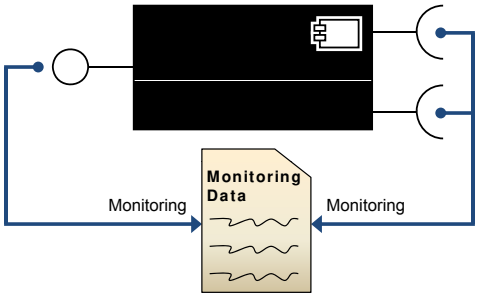

Figure 3: Gathering Monitoring Data from Black-Box Components

To capture the parametric dependencies between the application input and output, our approach monitors at the level of component interfaces (see Fig 3; the monitoring approach is to a certain extent comparable to [**?**]). First, a component service under investigation is executed in a testbed to gather runtime information about component behaviour. Such a testbed provides at least stubs for all required services of the component and has a load generator to call the provided service under investigation.

For this paper, we assume that a representative test load is given (for a recent overview on test data generation see [**?**]), having a broad selection of input data that causes the component service to be executed in different ways with every repetition. The datasets obtained from monitoring serve as the input for the machine learning (Fig. 1, Step ①) to learn the parametric dependencies between input and output.

For the service, it is monitored which input data (arguments of the provided service) causes which behaviour at the requires side (required services) of the component. Monitored data

properties are:

- for primitive types (i.e. `int`, `float`, `boolean` etc.): their actual values
- for all *one*-dimensional arrays (e.g. `int[]`, `String[]`), `Collection`, or `Map` types: the number of their elements
- for one-dimensional arrays of primitive type (e.g. `int[]`, `boolean[]`), additionally aggregated data, such as number of occurrences of values in an array (e.g. the number of '0's and '1's in an `int[]`)
- for a *multi*-dimensional array (e.g. `String[][]`): its size, plus results of individual recording of each included array (as described above)

For the service under investigation, we additionally monitor which required services are called by it how often and with which parameters. The described data monitoring and data recording can be applied to component interfaces without a-priori knowledge about their semantics, and without inspecting the internals of black-box components. Supporting and monitoring complex or self-defined types (e.g. objects, structs) requires domain expert knowledge to identify important properties of these data types. Still, generic data types (for example defined in a standard library) are used very often, and our approach can handle these cases automatically.

4.3 Step 2: Learning from Monitoring Data

Our approach uses genetic programming as machine learning approach for recovering functional dependencies in the monitored data. We use the Java Genetic Algorithm Package JGAP [**?**] to support machine learning (a general introduction for genetic programming, a special case of genetic algorithms, can be found in [**?**]).

For every gathered input data point (e.g. size of an input array or value of a primitive type) a gene representing that parameter in the resulting model is introduced. In the example from Fig. 2 the dimensions of the input `uploadedFiles` array, the size of individual elements, and the size of files passed to `storeLargeFile`/`storeSmallFile` are such genes.

For our approach, we combine genes representing mathematical functions to express more complex dependencies. Simple approaches like linear regression could be applied as well, but cannot handle non-continuous functions and produce little readable approximations by polynomials.

To express functional dependencies, in addition to default JGAP genes (e.g. mathematical operations for power, multiplication, addition, constants), we introduced new genes to support non-continuous behaviour (e.g. jumps caused by "if-then-else"). Such new genes are for example required to express when `storeLargeFile`/`storeSmallFile` from Fig. 2 are called.

4.4 Step 3: Created Model

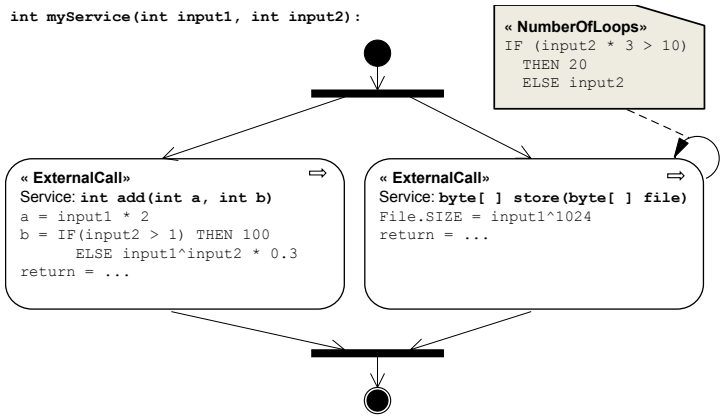

Figure 4: Example Instance of the Output Model

The target model is an instance of the Palladio Component Model [**?**][1], which is a software performance model. The model is comparable to UML2 activities. The model instance has a behavioural model for each provided service of a component (an example, independent from the one in Fig. 2, is shown in Fig. 4). A component service's behaviour model describes

- the control flow of the service:
  - which services of other components are used/required (`ExternalCalls`),
  - how often those services are called (`NumberOfLoops`) depending on input argument of the provided service, and
  - conditions under which required services are called.
- Additionally, the data flow captures:
  - which parameters are passed to used/required services, and
  - where return data from used/required services is passed to.

For the external calls (`add` and `store` in Fig. 4), the model includes dependencies between component service input arguments and external call parameters, with one formula per input parameter of an external call (e.g. `a = input1 * 2` in Fig. 4). Also, the number of calls to each required (external) service is annotated using parameterisation over input data (cf. `NumberOfLoops` grey box in 4).

As the target model is a performance model, we do only consider performance-relevant abstractions of control and data flow. For example, we are not interested in the content of a byte array. Instead, we focus on performance relevant characteristics like the size of

[1] see http://www.palladio-approach.net

the array. Imagine a service compressing a byte array; its processing time will dominantly depend on the size of the array.

5 Validation

We validated the reverse engineered model produced by our approach by comparing it with manually reverse engineered models. The manual reverse engineering was done by the authors by investigating the source code and manually extracting a parametric performance model. Manual extraction was done independently, without inspecting the results of the automated approach.

5.1 Example System

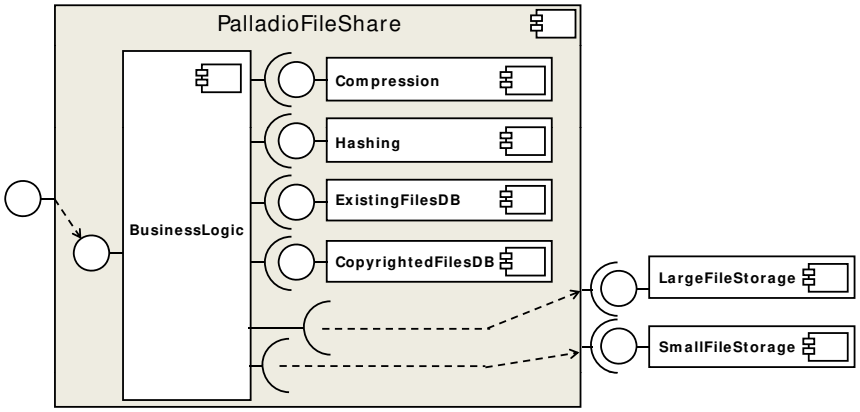

Figure 5: Evaluation Component Architecture – Considered Component: BusinessLogic

For evaluating our approach, we created a Java implementation of a file sharing application called "PalladioFileShare". Users upload files and share them with other users who can download these files. For the evaluation, we concentrated on the "upload" use case. Within the application architecture (Fig. 5), we focus on the `BusinessLogic` component.

The `BusinessLogic` offers the primary upload service. It itself relies on two system-external components `LargeFileStorage` and `SmallFileStorage`, responsible for persisting data amounts of different size. The `BusinessLogic` is controlling file uploads by triggering required services of other components. `Compression` allows to compress uploaded files, while `Hashing` allows to produce hashes for uploaded files. `ExistingFilesDB` is a database of all available files of the system; `Copyrighted-FilesDB` holds a list of copyrighted files that are excluded from file sharing.

The component `BusinessLogic` (c.f. Fig. 6) considered in this evaluation provides the `uploadFiles` service, which itself coordinates the processing of each uploaded file.

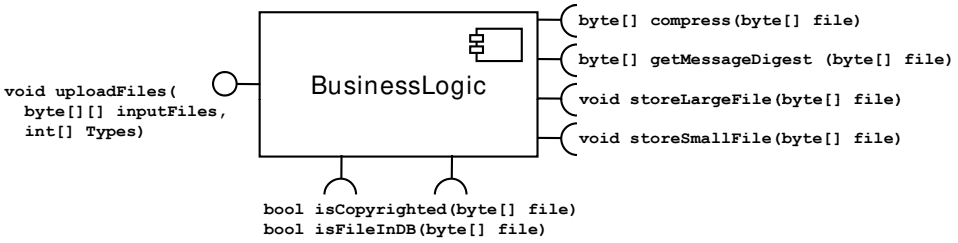

Figure 6: BusinessLogic Component

Therefore, it utilises its required services available from the architecture (Fig. 5).

The `uploadFiles` service has two arguments: first, a byte array of files `inputFiles`; and second, an int array `Types` characterising uploaded files. Uploaded files can have different filetypes (i.e. compressed files like JPEG, ZIP or non-compressed files like text files, xml documents).

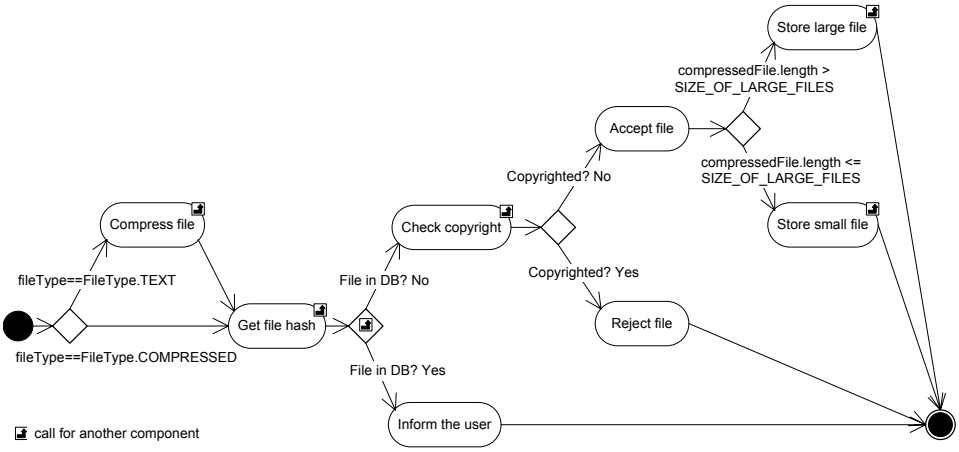

Figure 7: File Processing Steps – Control Flow Model of BusinessLogic Implementation (depicted here for reader's convenience; the reverse engineering approach does not have this model as an input but operates on BusinessLogic code directly)

The data dependent control flow of `BusinessLogic` is visualised in Fig. 7. The control flow is executed for every uploaded file. First, based on a flag derived from each uploaded file, it is checked whether the file is already compressed (e.g., a JPEG file). An uncompressed file is processed by the COMPRESSION component.

Afterwards, it is checked whether the file has been uploaded before (using `ExistingFilesDB` and the hash calculated for the compressed file), since only new files are to be stored in `PalladioFileShare`. Then, for files not uploaded before, it is checked whether they are copyrighted using `CopyrightedFilesDB`. Finally, non-copyrighted files are stored, either by `LargeFileStorage` for large files (if the file size is larger than a certain threshold) or `SmallFileStorage` otherwise.

5.2 Results

In the case study, we monitored the behaviour of `BusinessLogic` in 19 test runs, each with different input data (number of uploaded files, characterisation of files: text or compressed, and file sizes). The test runs were designed to cover the application input space as far as possible. The monitoring data was gathered once for the component. Then, for each required external service of the monitored component's service, genetic programming was started to answer:

- how often and under which conditions an required external service is called and
- for every parameter characterisation, how it depends on service input arguments and other services' return values.

In the rest of this section, we show some interesting excerpts from the complete results. We will go through the decisions in the control and data flow and point out which dependencies have been learned and which were expected to be learned by manual reverse engineering.

As the names of input parameters are used hereafter, we use the signature of the file sharing service,

```
void uploadFiles(byte[][] inputFiles, int[] fileTypes)
```

(c.f. Fig. 6). In the signature, `inputFiles` contains the byte arrays of multiple files for upload and `fileTypes` is an array indicating corresponding types of the files, e.g. `FileType.COMPRESSED` or `FileTypes.TEXT` (i.e., uncompressed).

Data dependent control flow: Use of Compression component for multiple files. In the BusinessLogic, the number of calls of the `Compress` component depends on the number of uncompressed files (`FileType.TEXT`) uploaded. Using genetic programming (JGAP), the correct solution for the number of calls was found to be:

$$inputFiles.length - fileTypes.SUM(FileType.COMPRESSED)$$

The number of all files reduced by those already compressed, where SUM is aggregated data from the monitoring step. The reverse engineering dependency exactly fits the expected one. The search time was less than one second, meaning that the fitness function indicated an optimal solution being found after that time, leading genetic programming to terminate.

When to use LargeFileStorage or SmallFileStorage. For answering this question, monitoring data from uploads with just one file was analysed. A set of eleven different input files (different file types, different size) was used as test data. JGAP (generally; see below) found an optimal solution: If the file size is larger than 200,000 (bytes), a file with the same size like the file passed to the `Hashing` component is passed to `LargeFileStorage`, else nothing is stored with `LargeFileStorage` (an opposite dependency was found for the usage of `SmallFileStorage`).

For `LargeFileStorage` the following expression resulted:

$$if(hashingInputFiles.length > 200,000)\{..\}$$

where `hashingInputFiles` is the size of an individual uploaded file passed to `Hashing`. In this case genetic programming selected the correct input variable (`hashingInputFiles` instead of for example `inputFiles`, or one of the other measuring data point) and guessed the constant '200,000'.

The search time was less than five seconds. The implementation-defined constant '200,000', which was identified by manual reverse engineering, was not always identified correctly by the approach due to the limited number of input files (there were no input file with a size of 200,000 and 200,001). Yet, the recovered function did not contradict the monitoring data. Adding further input files would have enabled the approach finding the correct constant in all cases.

We tested an additional run of JGAP where the monitoring data was disturbed by calls of `uploadFiles` that did not lead to a storage write because the file already existed in the database (one out of eleven calls did not lead to a write). Such effects depending on component state are visible at the interface level only as statistical noise that cannot be explained based on interface monitoring data. In this case the optimal solution could still be found, but within more time: less than 20 seconds (in average). For monitoring data with uncertainty, the confidence in the correctness ("fitness function" calculated by JGAP) of the result decreased: There was a deviation between monitored data and approximation of genetic programming due to the disturbed monitoring data.

The average behavioural impact of uploads where no storage takes place can be captured by computing the long-term probability of such uploads independently of the uploaded files.

SingleFileAnalysis – Size of files passed to the Compress component. The size of single files passed to the compression service was learned in another run of genetic programming. The input consisted of a series of eleven input files again. The dependency recovered was:

$$inputFiles[x].length * fileTypes[x]$$

where `inputFiles[x]` is the size of an individual uploaded file and `fileTypes[x]` the integer encoded type of a file, where 0 represented non-compressed files. Within a search time of less than one second the optimal solution could always be found.

Estimation of the compression ratio. In addition to applying the approach for the `BusinessLogic` component, we used it for estimating the compression ratio of the `Compression` component. `BusinessLogic` itself relies on the compression ratio of this component, as the size of non-compressed input files that are passed to the storage components is larger than the ones being compressed by `Compression` beforehand.

`Compression` is a Lempel-Ziv-Welch (LZW) implementation. The compression ratio of LZW strongly depends on the data characteristics (e.g. entropy, used encoding), no

optimal solution exists to describe the compression ratio. Therefore, JGAP produced a large variety of approximations of the compression ratio. A good approximation found after 30 seconds had the following form:

$$0.9 * 0.5 * (X3 - (0.9 * 0.5 * (X3 - (0.9 * (0.9 * 0.5 * X3) * 1.0))))$$

where $X3$ is the size of the file input for the `Compression` component, which was found (by the genetic programming) to be significant.

The complexity of these functions, which for approximations can get even more verbose, will be hidden from the user in a tool chain.

Findings In the investigated example, all dependencies (eleven in total) identified by manual reverse engineering by a human were also found by our automated approach. Hence, we consider our approach to be applicable for reverse engineering of parameterisations from business components comparable to `BusinessLogic`.

6 Limitations and Assumptions

For the monitoring step, we assume that a representative workload (including input parameter values) can be provided, for example by a test driver. For running systems, this data can be obtained using runtime monitoring. Otherwise, a domain expert needs to judge which scenarios are interesting or critical (for example [**?**] provides an overview on test data generation).

In the monitoring step of our approach, asynchronous communication (e.g. message-based communication) is not supported by the used logging framework. Hence, if there is asynchronous communication inside the component under investigation, monitored results will be misleading. This limitation will be addressed in next versions of our implementation.

To support the black-box component principle, monitoring should be performed in an automated way. In general, collecting dozens of metrics for input and output data is not justified by the requirements of our approach. At the moment, we provide heuristics for important characteristics only for primitive types or general collection types like `List`. Thus, currently a domain expert needs to specify important data characteristics manually for more complex data structures.

In the machine learning step, heavily disturbed results originating from sources not visible at the interface-level, decrease the convergence speed and lower the probability of finding a good solution.

The termination condition for genetic programming currently is very simple. Search terminated after a fixed number of generation (iterations) or if the fitness function shows that an optimal solution was found: Genetic programming perfectly matches the monitoring data.

7 Conclusion

In this paper we presented a reverse engineering approach capable of reconstructing behavioural service models of component-based applications. The approach recovers functional dependencies in control and data flow of component services, allowing to estimate the performance impact of a component on other components. As the approach only monitors data and control flow visible at the interface level, it supports black-box components, for which no internals need to be known.

Utilising genetic programming as machine learning technique, the validation showed that the approach is able to deal with the complex data and control flow of a typical information systems business logic component. Genetic programming supports data and control flow dependency recovery by: 1) appropriately selecting input variables from a large input space, 2) correctly finding multi-dimensional criteria, dealing with unknown constants, and 3) handling disturbed monitoring data, as the validation showed.

The approaches enables automating reverse engineering for parametric service performance models, which beforehand required costly manual reverse engineering. Through the unique combination of machine learning and monitoring, the field of control and data flow analysis is enriched with an approach to find parametric dependencies in software components.

For the future, we plan to improve automation of the approach for having an "integrated monitoring-learning-model-output" tool chain.

Though performance effect *propagation* is captured in this approach, the performance impact of component-*internal* behaviour (for example an internal sorting algorithm) is currently not captured. An improved version of the approach should reverse engineer this internal behaviour as well.