Evaluation of Trace Reduction Techniques for Online Trace Visualization Utilizing Kieker

Master's Thesis

B.Sc. Björn Weißenfels

May 12, 2014

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Florian Fittkau

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Software landscapes are becoming increasingly complex. Knowledge about the communication between the applications in the software landscape is often lost. The comprehension of such software systems can usually not be solved by static code analysis. The use of dynamic analysis in this process, e.g., with Kieker, leads rapidly to a large amount of trace data. For several purposes like storage or live trace visualization, e.g., with ExplorViz, these data must be reduced.

This thesis addresses this issue by presenting 12 different trace reduction techniques. These techniques are based on summarization, metric-based filtering, or removing utility methods. We evaluate all techniques based on our assessment criteria, and implement the most promising techniques as filters for ExplorViz. We also experiment with 6 execution traces of different object-oriented systems and study the gain attained by these reduction techniques.

Contents

Chapter 1

Introduction

Section 1.1 gives a motivation why trace reduction techniques are needed in online trace visualization. In Section 1.2 we describe the goals of this thesis in detail. Afterwards, the thesis' remaining structure is outlined in Section 1.3.

1.1 Motivation

Software landscapes in companies are becoming increasingly complex. Knowledge about the communication between the applications in the software landscape is often lost due to lack of documentation. Knowledge about the communication within the applications themselves can also be lost. This knowledge usually can not be recovered through static code analysis due to the complexity of such systems. Furthermore, the gap between a program's static specification and its dynamic behavior is particularly large in object-oriented programs.

Online trace visualization, i.e., displaying of program executions in the software landscape at run time, can recover the understanding of the software environment and applications.
The observed methods provide a very large amount of data, since each call ends in monitoring records. These entries are then combined to traces. These traces still form a very large amount of data that must be reduced for visualization. This must be conducted efficiently and with a minimum loss of information.

In the proposed thesis several trace reduction techniques are compared and rated for the use in ExplorViz, a tool for online trace visualization of large software landscapes.

1.2 Goals

This section describes our two aimed goals and their subgoals.

G1: Theoretical Evaluation of Trace Reduction Techniques

The first goal is a theoretical evaluation of trace reduction techniques. We divide this goal into the following three subgoals.

G1.1: Description of Trace Reduction Techniques

In Cornelissen et al. [2008], an assessment methodology for trace reduction techniques is described. The paper presents 14 techniques, which are assigned to 4 categories. In the proposed thesis, we will describe these categories and techniques in detail.

G1.2: Definition of Evaluation Criteria

Since the evaluation criteria by Cornelissen et al. [2008] do not fully comply our requirements, we will define the four evaluation criteria: reduction rate, performance, parallelizability, and information preservation. Furthermore, we will define the metrics by which they are measured.

G1.3: Theoretical Evaluation Regarding to the Criteria defined in G1.2

We will evaluate all of the described trace reduction techniques regarding to the criteria defined in G1.2. This theoretical evaluation will be partially based on estimated values.

G2: Practical Evaluation of Trace Reduction Techniques

The second goal is a practical evaluation of some of the previously described trace reduction techniques. We divide this goal into the following two subgoals.

G2.1: Implementation of Filters for Kieker

We will choose at least the four most promising techniques for the use in ExplorViz based on the theoretical evaluation and in collusion with the advisor. Afterwards, we will implement these selected techniques as filters in the high-throughput tuned version of Kieker.

G2.2: Practical Evaluation Regarding to the Criteria in G1.2

For the practical evaluation, we will use the sample applications described in Section 2.2 to generate different traces. In addition, we will use the sample traces of Cornelissen et al. [2008]. Thus, the filters will be tested with different traces and a comparison with the results of Cornelissen et al. [2008] will be possible.
We will evaluate the chosen trace reduction techniques regarding to the criteria defined in G1.2.

1.3 Document Structure

The remainder of this thesis is structured as follows:

- ▻ **Chapter 2** introduces the foundations of this thesis in Section 2.1 and the used technologies in Section 2.2.
- ▻ **Chapter 3** presents the trace eduction techniques investigated in this thesis. This chapter address the goal G1.1, as described in Section 1.2.
- ▻ **Chapter 4** defines the criteria for the assessment of trace reduction techniques. This chapter address the goal G1.2.
- ▻ **Chapter 5** starts with an overview of the evaluation of the investigated techniques. Subsequently, the ratings are explained in detail. The chapter concludes with an overall ranking in Section 5.6. This chapter address the goal G1.3.
- ▻ **Chapter 6** establishes a selection of techniques that are implemented in Section 6.1. Afterwards the design of the implementation is described in Section 6.2. This chapter address the goal G2.1.
- ▻ **Chapter 7** contains the results of the experimental evaluation. This chapter address the goal G2.2.
- ▻ **Chapter 8** presents related work of this thesis.
- ▻ **Chapter 9** concludes the thesis. It gives a summary of the presented approach and its evaluation in Section 9.1. In the end, Section 9.2 presents the chances for future work.

Chapter 2

Foundations and Technologies

This chapter describes the foundations and technologies that will be used in the following chapters. Section 2.1 describes what can be imagined under dynamic analysis and how a benchmarking of a Java application should be done. Section 2.2 describes the technologies used in this thesis.

2.1 Foundations

Dynamic analysis, and in this context the normal ExplorViz records, and Java benchmarking are described in this section.

2.1.1 Dynamic Analysis

Dynamic analysis is the testing and evaluation of an application during its runtime. For instance, Kieker and ExplorViz are tools for the dynamic analysis of software systems and are described in the following.

Salah and Mancoridis [2004] defines a trace as a sequence of runtime events, that describe the dynamic behavior of a running program. In this thesis two runtime events are considered, the *BeforeOperationEventRecord* and the *AfterOperationEventRecord* build by ExplorViz.

Figure 2.1 illustrates the structure of the used records, while Table 2.1 describes the meanings of the attributes occur in this class diagram.

2.1.2 Java Benchmarking

Java performance is difficult to benchmark because factors such as the Java application, the virtual machine, the garbage collector, and the heap size exert influence. Georges et al. [2007] compares Java performance evaluations and develops a statistically rigorous methodology for startup and steady-state performance.
For startup performance, multiple VM invocations executing a single benchmark iteration are run and subsequently compute confidence intervals.
For steady-state performance, multiple VM invocations are run, each executing multiple benchmark iterations. Then, a confidence interval is computed based on the benchmark

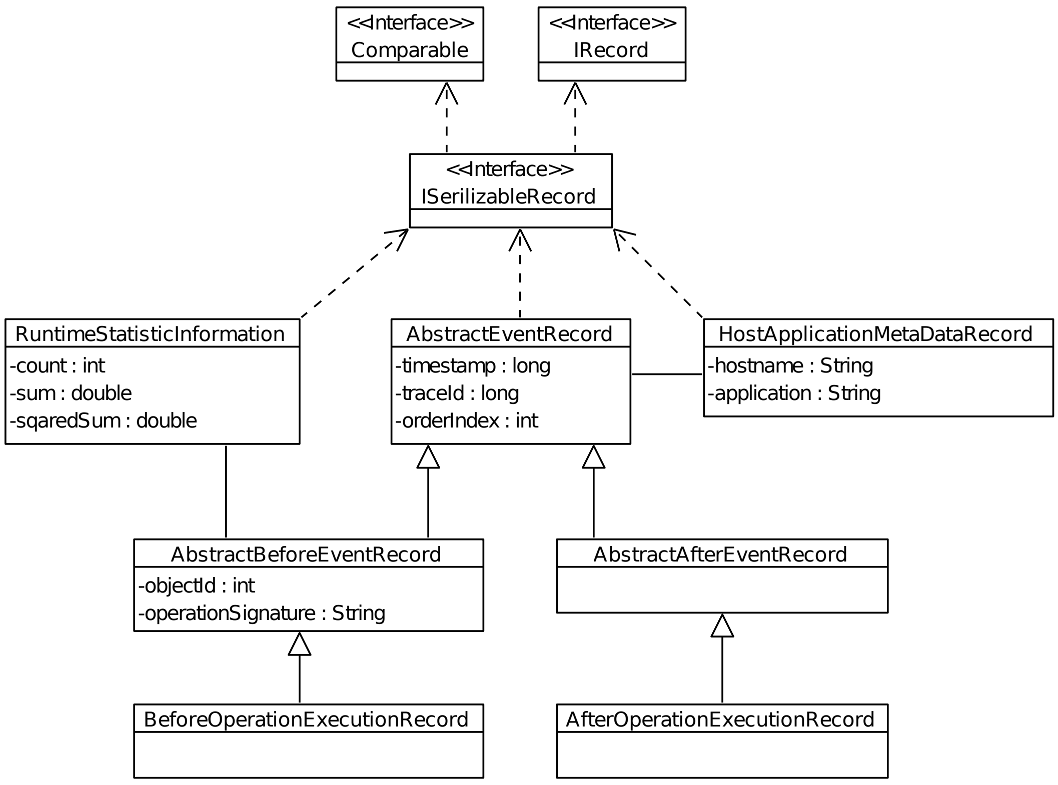


Figure 2.1. Structure of the used records with the regarded attributes

iterations across the various VM invocations once performance variability drops below a given threshold.

2.2 Used Technologies

This section gives a short description of Kieker and a more detailed of ExplorViz. Furthermore, the four sample applications are briefly introduced.

2.2.1 ExplorViz

ExplorViz is a tool for live trace visualization to support system and program comprehension in large software landscapes. It allows, for example, the discovery of the real communication between the components and the amount of usage for each component. Furthermore, it is an interactive approach for the live, explorable visualization of moni-

Table 2.1. Description of the attributes from Figure 2.1

Name	Description
timestamp	A timestamp in nanoseconds, when the operation was invoked.
traceId	A unique numerical value to identify the related records.
orderIndex	A numerical value, to bring the records into the correct order.
hostname	The name of the host the application is running on.
application	A name of the application under observation.
objectId	An identifier for the object which provides the operation.
operationSignature	A full qualified name of the operation.
count	Number of summarized records.
sum	Sum of the response times counted in this statistic information.
sqaredSum	Sum of squares of the response times counted in this statistic information.

toring traces. Therefore, perspectives for the software landscape and the system level are combined. The landscape level visualization is based on a mix of UML deployment and activity diagram elements, the system level view uses the city metaphor for each software system [Fittkau et al. 2013a]. Figure 2.2 shows an example of the landscape level view, which depicts the communication between applications in the PubFlow software landscape.

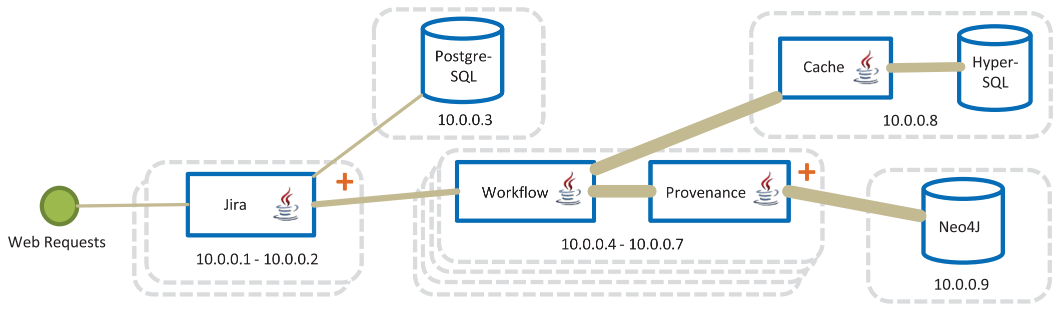
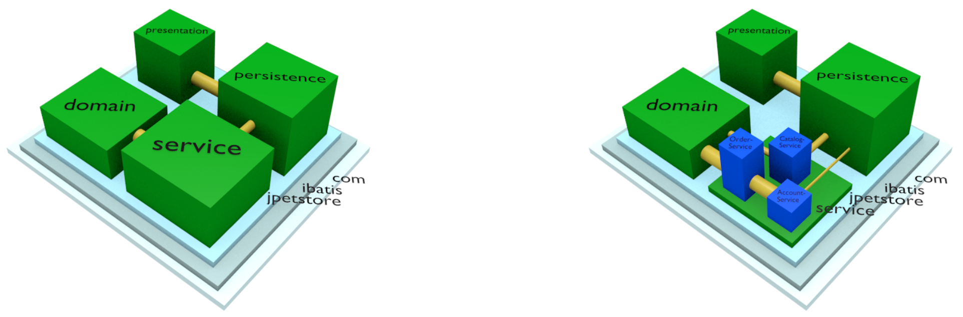


Figure 2.2. Macro view on landscape level showing the communication between applications in the PubFlow (http://pubflow.de) software landscape

Figure 2.3 shows the system level perspective of the JPetStore[*MyBatis JPetStore*] application for demonstrating the exploration concept. On the left side the package structure of JPetStore and the communication between these packages can be seen. On the right side the service component is opened, so one can see the deeper level components, in this case the classes of the service component.

Figure 2.4 provides an overview of the activities of ExplorViz. The first activity (A1) is to monitor the applications in the software landscape, e.g., with Kieker. The next activity (A2) is the preprocessing of the collected monitoring data. The incoming monitoring records

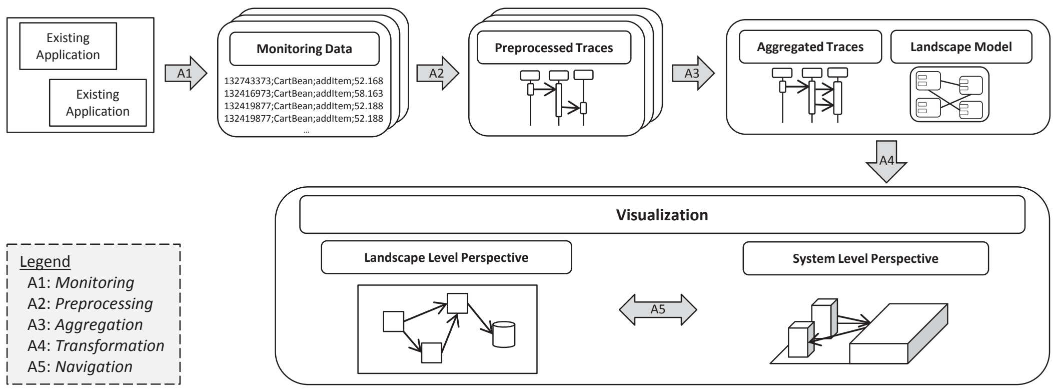


(a) Macro view visualizing four components of jPetStore

(b) Relationship view with opened service component

Figure 2.3. Mockup of system level perspective on the example of jPetStore[*MyBatis JPetStore*] for demonstrating the exploration concept

are consolidated into traces, which then can be reduced through the reduction techniques described in this thesis. To handle a large amount of monitoring data, this activity can be parallelized. The third activity (A3) creates and updates the landscape model. The fourth activity (A4) transforms the landscape model into a visualization model, while the last activity (A5) is the navigation between the described perspectives.



```
132743373;CartBean;addItem;52.168
132416973;CartBean;addItem;58.163
132419877;CartBean;addItem;52.188
132419877;CartBean;addItem;52.188
...
```

Figure 2.4. Activities in our ExplorViz approach for live trace visualization of large software landscapes

2.2.2 Kieker

Kieker is an extensible framework for continuous monitoring and analyzing the runtime behavior of software systems [van Hoorn et al. 2012]. It monitors the response times and control-flow for selected operations [Rohr et al. 2008]. The framework provides an extensible architecture and supports the construction of additional monitoring record types, probes, analyses, and visualizations [van Hoorn et al. 2009]. Configurable readers and writers allow Kieker to be used in different scenarios, like writing on the file system, or via TCP for online analysis.

High-throughput tuned Kieker

In order to meet the real time requirements of ExplorViz, a variant of Kieker was designed with special emphasis on providing high-throughput for the monitoring data. Both the monitoring and the analysis components were revised for this project. [Fittkau et al. 2013b]

2.2.3 PubFlow

PubFlow is a workflow oriented data publication framework. It is built to provide the tools and the infrastructure, which is needed by scientists and data managers to get the research data out of the institutional data repositories into the publicly available data centers [Brauer and Hasselbring 2013].

2.2.4 Checkstyle

Checkstyle is a medium-size Java source code validation tool. It can be used from command line, but it can also be integrated in the build process or a development environment. Thus it automates the process of checking Java code against a coding standard.

2.2.5 JPacman

JPacman-Framework is a small application used for teaching testing purposes at Delft University of Technology. The program is an implementation of the well-known Pacman game in which the player can move around on a graphical map while eating food and evading monsters.

2.2.6 JHotDraw

JHotDraw is a medium-size open source Java graphics framework for structured drawing editors. It was developed as a showcase for design pattern usage and is acknowledged to be well-designed. It provides a GUI that offers various graphical features such as the insertion of figures and drawings.

Chapter 3

Trace Reduction Techniques

Since monitoring of large software landscapes produces a large amount of execution traces, techniques for reducing these data are needed. Cornelissen et al. [2008] grouped trace reduction techniques into the four categories summarization, metrics-based filtering, language-based filtering, and ad hoc, and named techniques for each of these categories. Based on the references of Cornelissen, these techniques are described below.

3.1 Summarization

Summarization techniques try to shorten a trace by replacing part of its contents by more concise notations. Typical summarization targets include recurrent patterns. The four investigated techniques in this category are execution pattern notation, pattern summarization, object and event clustering, and monotone subsequence summarization.

3.1.1 Execution Pattern Notation

The execution pattern notation described by De Pauw et al. [1998], is a visualization of execution traces at varied levels of abstraction. It is inspired by Jacobson's interaction diagrams [Jacobson 1992]. An example of a simple interaction diagram is shown in Figure 3.1. A vertical line represents a class with the class name on top. An instance of this class is displayed as a rectangle on that line. Arrows represent method calls. Time is shown on the vertical axis, the first method call is on top and the latest on the bottom.

Furthermore, it shows a call trace where object A calls a method m1 from object B, then object B calls a method m2 on object C. After that object B calls a method m3 on object A and finally object A calls a method m4 on object D.

However, there are ambiguities. The order of classes along the horizontal axis is undefined. Furthermore, it's unclear whether the method from A to D is called within the method m3.

The execution pattern notation described in De Pauw et al. [1998] counteracts these ambiguities by utilizing a tree structure. The diagram is read from top left to bottom right. Objects are represented as colored rectangles. Method calls are displayed as labeled arrows between these rectangles. The label shows the name of the invoked method. The legend

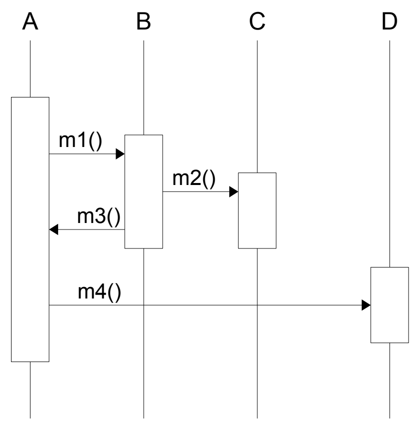


Figure 3.1. Example of a simple interaction diagram based on [De Pauw et al. 1998]

on the right shows which color represents which class. Each object contains a numerical identifier (ID).

For example, Figure 3.2 shows the same call trace as Figure 3.1 in the execution pattern notation. First an instance of class A with the ID 1 calls the method m1 on an instance of class B with the ID 2. Then, this object calls the method m2 on an instance of class C with the ID 3, and then the method m3 on the object with ID 1. After that, method m4 is called. In contrast to the interaction diagram, m4 is unambiguously not called within the method m3.

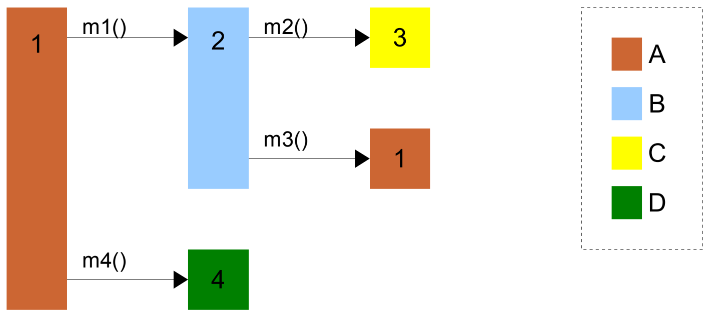


Figure 3.2. Example of the execution pattern notation based on [De Pauw et al. 1998]

Figure 3.3 shows the possibility to hide some information and simplify the pattern for easier understanding. The black border around the blue rectangle indicates that there is at least one hidden method call from this object.

Figure 3.4 shows a more complex execution pattern. There is one object of class A, B

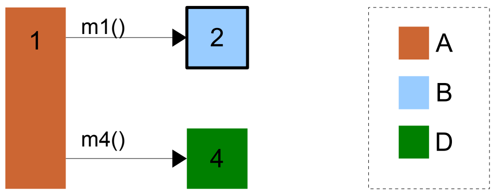


Figure 3.3. Example of a collapsed execution pattern based on [De Pauw et al. 1998]

and C, and three objects of class B. The objects with the ID 5 and 6 have a black border, meaning they are collapsed. One can recognize a repeating pattern. Alternately, method m1 of class B and method m4 of class D are called.

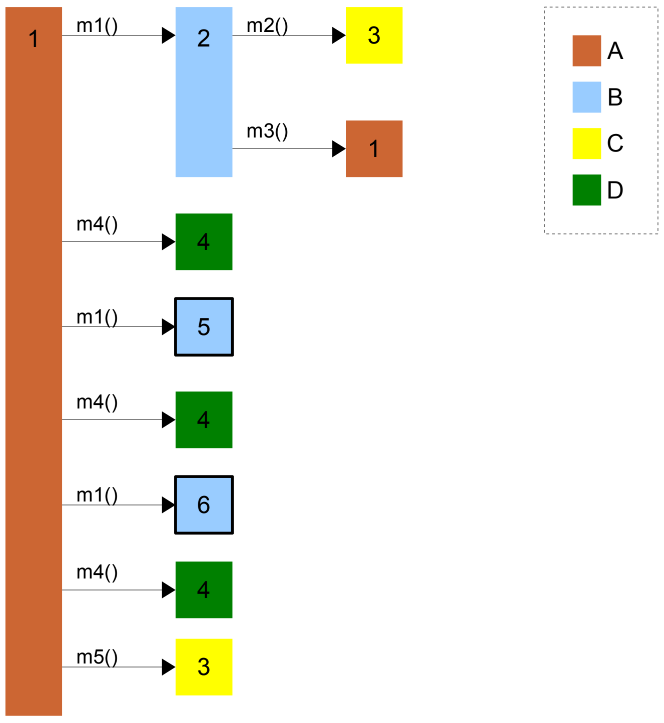


Figure 3.4. Example of a repetition pattern based on [De Pauw et al. 1998]

As shown in Figure 3.5, the above repetition can also be collapsed. A black border around the repeated methods and their receiving objects represents the repeated unit. The number of repetitions is shown in the lower left corner. To sum up different objects of the same class, their IDs are omitted in this view.

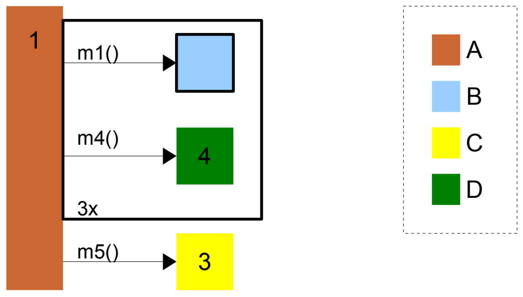


Figure 3.5. Example of a collapsed repetition pattern based on [De Pauw et al. 1998]

De Pauw et al. [1998] designates eight criteria when patterns can be regarded as equivalent. These are:

Identity - Patterns match with respect to identity, if they have the same graph structure, and every object and every message in the patterns are identical.

Class Identity - In contrast to *Identity*, objects no longer need to be identical but only descended from the same class.

Message Structure - Two patterns are equivalent, if their message (method) structure is identical and there is a nontrivial color substitution from one's coloring to the other.

Depth-Limiting - Limiting the depth of the trace means ignoring methods beyond a given depth. This will shorten the trace and can also lead to more pattern matches.

Repetition - Repetition can be divided in iteration and recursion. In the execution pattern view, an iteration, such as a loop, would appear as a repeated structure along the vertical dimension. A repeated structure in the horizontal direction would be a recursion. Two patterns match with respect to repetition, if they have the same graph structure after collapsing the repetitions.

Polymorphism - Two patterns match with respect to polymorphism, if they only differ in one or more classes, each having the same base class. In the collapsed view these classes are substituted through their base classes.

Associativity - Associativity of objects means ignoring self invocation on class level. Two patterns match with respect to associativity, if their structure is the same after removing all self invocations. As an example, Figure 3.6 shows on the left side a pattern with self invocations and on the right side the same pattern after removing the self invocations.

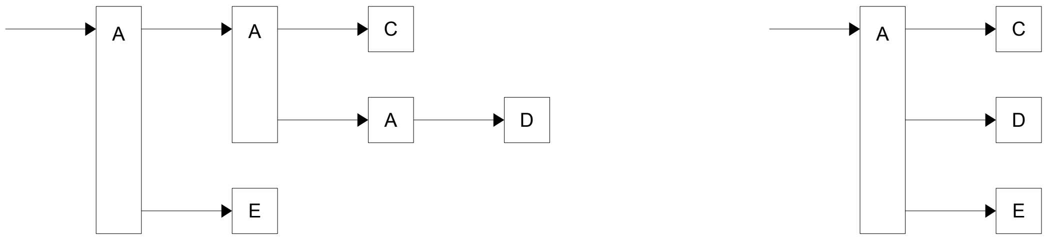


Figure 3.6. Example of equivalent pattern regarding associativity based on [De Pauw et al. 1998]

Commutativity - Commutativity with respect to method calls, means ignoring the order of the calls. The only difference between the left and the right pattern in Figure 3.7 is the order of the method calls m1 to m3. This is an example of two patterns which will match with respect to commutativity.

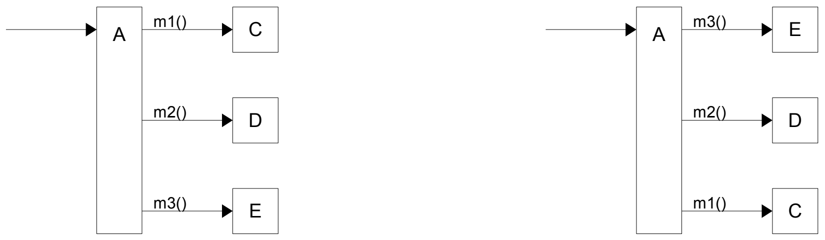


Figure 3.7. Example of equivalent pattern regarding commutativity based on [De Pauw et al. 1998]

3.1.2 Pattern Summarization

Safyallah and Sartipi [2006] defines an execution pattern as a contiguous part of an execution trace, that is contained by a minimum number of execution traces, where the minimum is defined by the user. Figure 3.8 shows the dynamic analysis framework to identify feature functionality in the source code proposed by Safyallah and Sartipi [2006]. For every considered feature a scenario-set is built in the pattern repository. The execution patterns are divided in three categories. *Intra-scenario-set common patterns* are patterns that are typical for a specific feature. Hence they occur in the majority of the scenario-set traces of a specific scenario-set. *Inter-scenario-set common patterns* are patterns that occur in the majority of all traces. The third category is *noise patterns*, which are patterns that do not contribute to a major system functionality.

Hamou-Lhadj and Lethbridge [2002] describe two lossless trace compression techniques. The first one is to detect and replace sequences. A sequence is a contiguous part of the

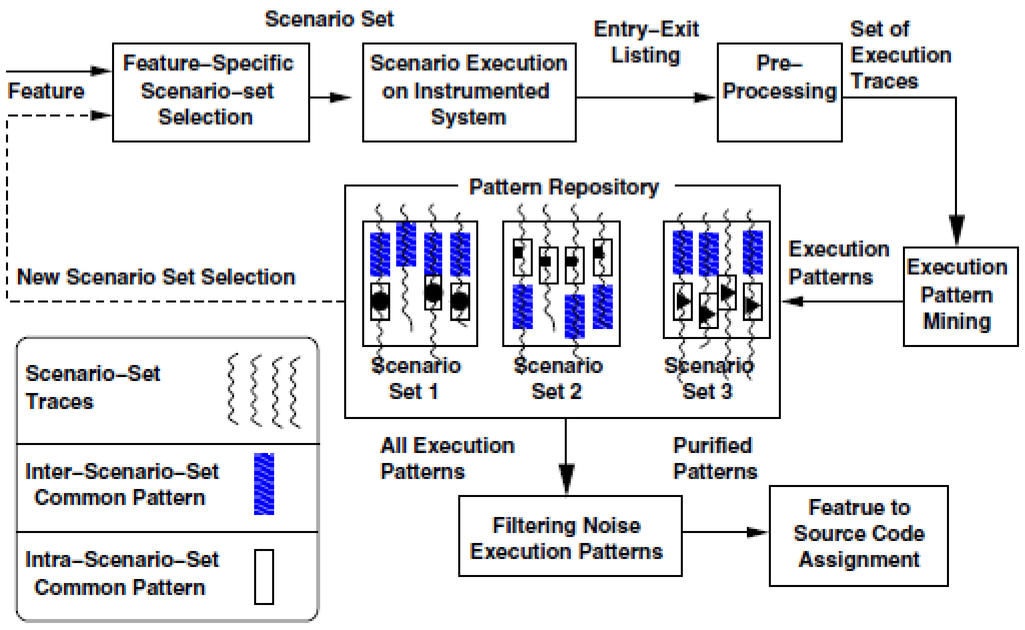

Scenario Set
Feature
Feature–Specific Scenario–set Selection
Scenario Execution on Instrumented System
Entry–Exit Listing
Pre–Processing
Set of Execution Traces
Pattern Repository
New Scenario Set Selection
Execution Patterns
Execution Pattern Mining
Scenario Set 1
Scenario Set 2
Scenario Set 3
Scenario–Set Traces
Inter–Scenario–Set Common Pattern
Intra–Scenario–Set Common Pattern
All Execution Patterns
Purified Patterns
Filtering Noise Execution Patterns
Featrue to Source Code Assignment

Figure 3.8. Dynamic analysis framework to identify feature functionality in the source code. Taken from [Safyallah and Sartipi 2006]

trace, which occurs several times in succession. These redundant calls caused by loops and recursion can be summarized as shown in Figure 3.9. In more detail, this figure shows on the left side an execution trace, where the events *B* and *C* occur alternately three times. On the right side of Figure 3.9, the same trace is shown in a compressed way. The repeated occurrences of *B* and *C* are summarized.

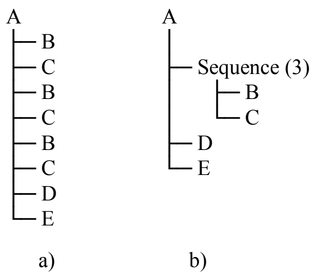


Figure 3.9. A contiguous sequence replaced with the number of its occurrences. Taken from [Hamou-Lhadj and Lethbridge 2002]

Hamou-Lhadj and Lethbridge [2002] presents also an algorithm to summarize nonoverlapping sequences in traces as shown in Figure 3.9. The algorithm uses a user specified threshold d for the maximum size of a repeated sequence. If n is the size of the trace, the algorithm takes $O(dn)$ time. Assuming that d is substantially smaller than n, the algorithm runs in linear time. To sum up also nested sequences, the algorithm only needs to be executed multiple times, once for each nesting level. Since the nesting levels of sequences should also be substantially smaller than n, the algorithm further runs in linear time. Figure 3.10 shows an example trace and the results after passing one and two times the algorithm.

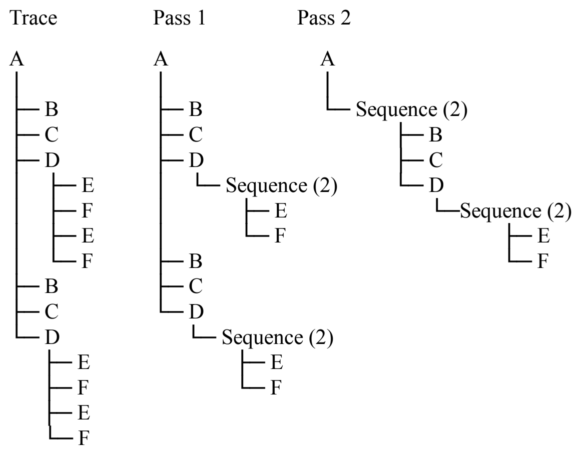


Figure 3.10. An example of detection and replacement of nested sequences. Taken from [Hamou-Lhadj and Lethbridge 2002]

The algorithm does not recognize recursions and also not all loops. Hamou-Lhadj and Lethbridge [2002] uses this technique only as preprocessing for their second compression technique, the common subexpression algorithm. This algorithm transforms a rooted tree to an acyclic graph, so that all the isomorphic subtrees are represented only once. This will remove all redundancies. Reiss and Renieris [2001] present an algorithm to transform a dynamic call tree to a directed acyclic graph. Figure 3.11 shows an example of a tree representation of a trace and its corresponding directed acyclic graph. In this example the number of nodes is reduced from 9 to 5, and the number of edges from 8 to 6.

3.1.3 Object and Event Clustering

Gargiulo and Mancoridis [2001] describe their tool Gadget for extracting the structure of Java programs. Since a static analysis of the source code is often not sufficient to determine the structure of object-oriented programs, Gadget collects runtime data of the examined program. Filtering and abstraction techniques help to select classes of interest. The resulting traces are shown in a dynamic dependency graph. To make the large dynamic structures

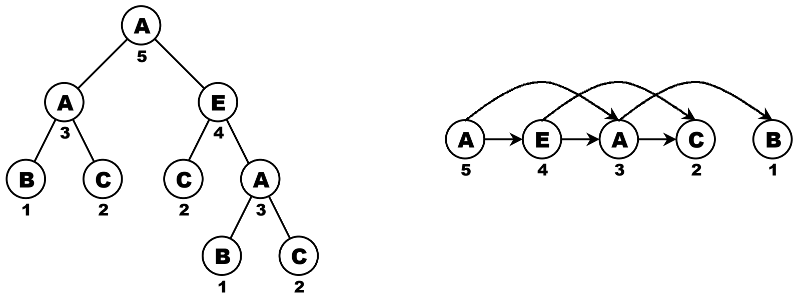


Figure 3.11. An example of a tree representation of a trace and the corresponding directed acyclic graph based on [Hamou-Lhadj and Lethbridge 2002]

more modular and easier to understand, a clustering tool, called Bunch, is used to partition the graph.

Figure 3.12 shows a dependency graph of a file system as an example of an input for Bunch. Figure 3.13 shows the corresponding output, an automatically produced high-level system organization of the same file system.

Bunch is a clustering tool, that decomposes the structure of software systems into meaningful subsystems [Mancoridis et al. 1999]. A software system is seen as a graph of modules and dependencies. This graph is partitioned so that modules within a partition have many dependencies, and modules in different partitions as few dependencies as possible. To find a good partition, clustering algorithms based on hill-climbing optimization techniques and genetic algorithms are used [Mancoridis et al. 1998].

This technique doesn't reduce the trace. However, based on this partitioning, other techniques like Package Filtering can be used.

3.1.4 Monotone Subsequence Summarization

Kuhn and Greevy [2006] describe a technique to present traces as signals in time and reduce the size of the visualization through the monotone subsequence summarization.

Figure 3.14 illustrates the procedure. In the upper part an example trace is shown. In the middle part this trace is divided into its monotone subsequences. That is, the trace is divided between each two consecutive events, where the nesting level decreases. In the lower part, each subsequence is compressed into one method-call-chain. As can be seen, this representation is space saving.

For a larger reduction the trace can be divided between each two consecutive events, where the nesting level decreases by a given threshold.

Cornelissen et al. [2008] extended this approach to a reduction technique by representing each subsequence only by its first event. Figure 3.15 shows the result of this extension on the previous example.

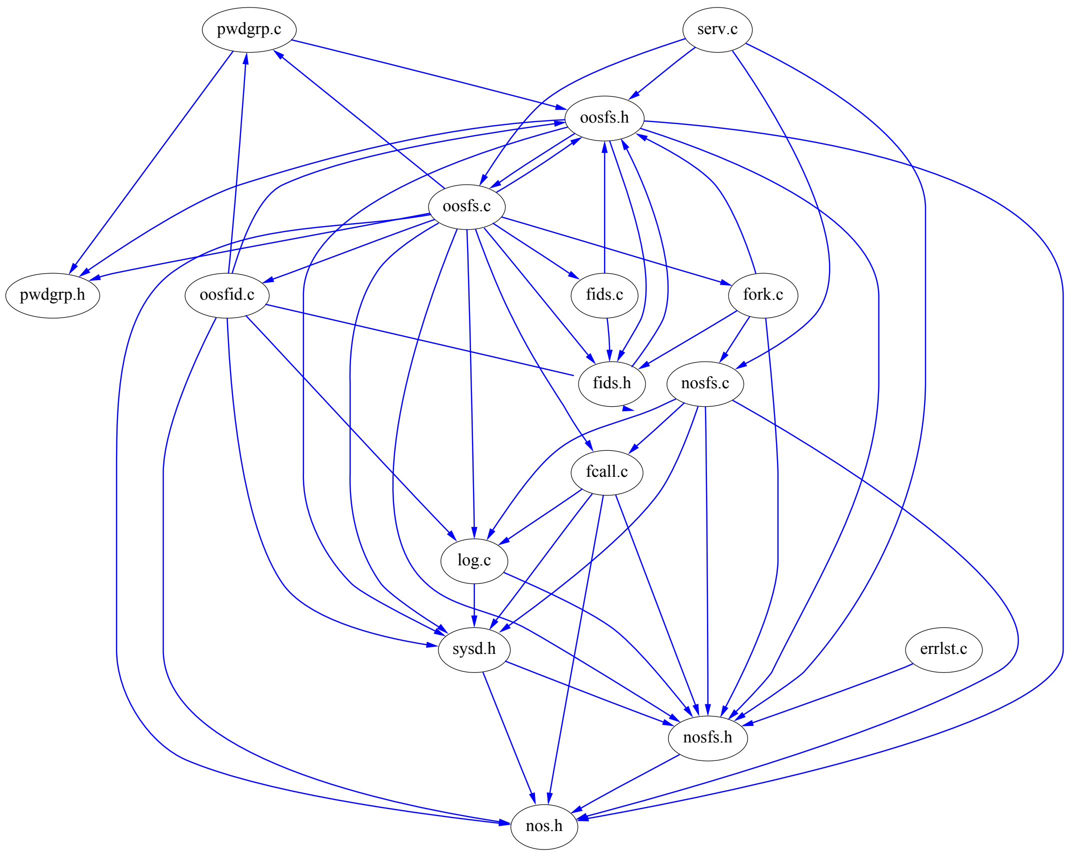


Figure 3.12. Example of module dependency graph of a file system taken from [Mancoridis et al. 1998]

3.2 Metrics-Based Filtering

Metrics-based filtering is centered around the use of certain metrics. The four investigated techniques in this category are Frequency Spectrum Analysis, Utilityhood Measure, Webmining and Stack Depth Limitation.

3.2.1 Frequency Spectrum Analysis

A program profile is a table with program entities and their total frequencies in a program execution. The frequency spectrum analysis is to analyze this profile with respect to the absolute frequencies and equal frequencies. Ball [1999] calls three tasks in which frequency spectrum analysis is useful.

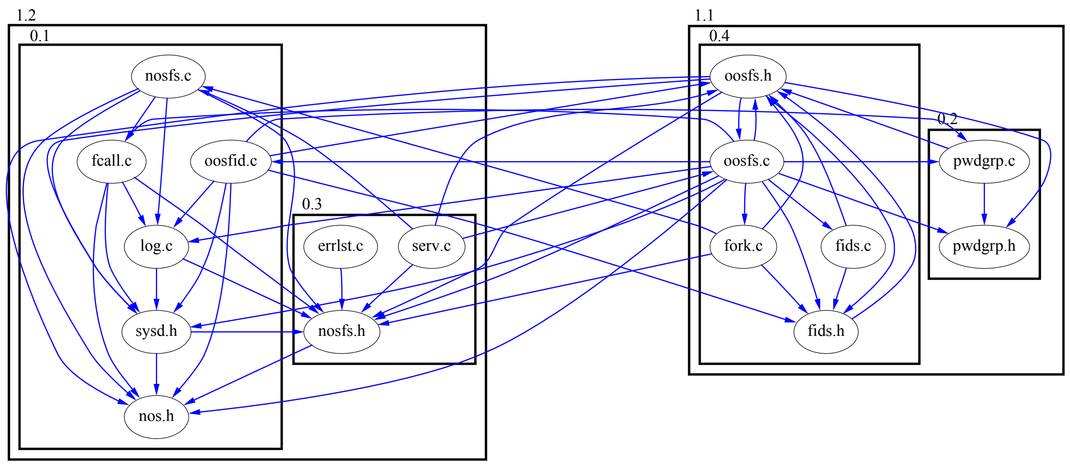


Figure 3.13. Example of an automatically produced high-level system organization of the same file system taken from [Mancoridis et al. 1998]

The first task is to partition the program by levels of abstraction. Methods implementing a high-level architectural pattern will be called fewer times, than methods implementing the guts of an algorithm. A high frequency indicates, that a method is involved in a repetitive computation.

The second task is to find related computations. If two methods are called with the same frequency, it is likely that they are related. Reasons for the same frequency could be, that one method invokes the other, or that they are called together. Methods with the same frequency form so-called *frequency clusters*.

The third task is to find computations related to specific attributes of a program's input or output. If you know something about the input or the output, like the size, you can compare the frequencies of the methods with such additional information. With this technique one can identify the parts of a program responsible for input or output.

Zaidman and Demeyer [2004] describe a heuristical clustering process based on the execution frequency analysis of Ball [1999]. They start with removing irrelevant events from the trace, like low-level method calls. In the next step the events are counted and every event gets annotated with its absolute frequency. With these frequencies a dissimilarity measure is computed for every n consecutive events. Zaidman and Demeyer [2004] uses the Euclidian distance

$$d = \sqrt{\sum_{j=1}^{w-1} (f_{j-1} - f_j)^2}$$

with window size w and events f_j. The last step is to identify near-zero regions and

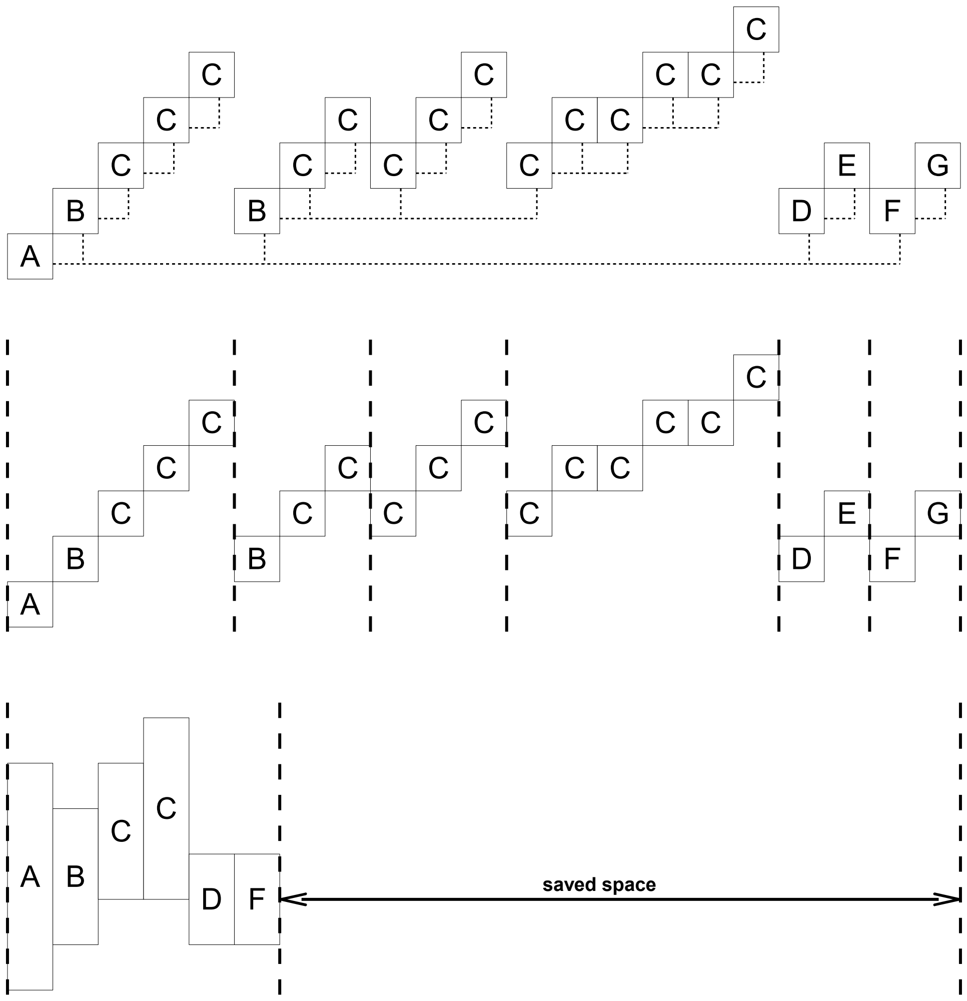


Figure 3.14. Example of monotone subsequence summarization based on [Kuhn and Greevy 2006]

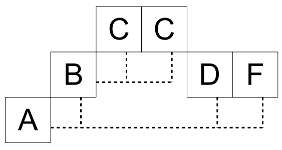


Figure 3.15. Example of Cornelissen's monotone subsequence summarization

frequency patterns.

3.2.2 Utilityhood Measure

To recover high-level behavioral models, Hamou-Lhadj et al. [2005] describe how to identify utility components, to separate them from the high-level components. Hamou-Lhadj et al. [2005] define a utility as: *"Any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program."*

To detect the utilities, a dependency graph is built from static analysis. Figure 3.16 shows an example with 7 components and a few dependencies between each other. Let **S** be the set of components considered in the analysis, and **IN** be the subset of **S**, that consists of the components that depend on a given component **C**. The utilityhood metric, **U**, of the component **C** is then defined as:

$$\mathbf{U} = \frac{|\mathbf{IN}|}{|\mathbf{S}| - 1}$$

U is between 0 and 1, where $\mathbf{U} = 0$ means that no other component calls **C**, and $\mathbf{U} = 1$ means that every other component calls **C**. The last case indicates, that **C** is an utility component.

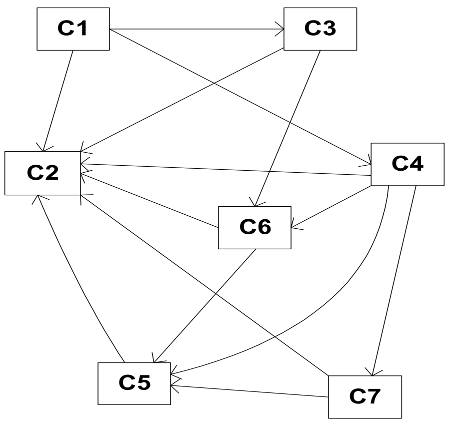


Figure 3.16. Example of a dependency graph taken from [Hamou-Lhadj et al. 2005]

$\mathbf{U}_i$ is computed for each $\mathbf{C}_i$, with i from 1 to 7. To make the differences between the components significantly, the mean μ and standard deviation σ is calculated from **U** and thus determines the so-called z-score:

$$\mathbf{Z} = \frac{\mathbf{U} - \mu}{\sigma} \quad , \quad \mu = \frac{1}{n}\sum_{i=1}^{n} \mathbf{U}_i \quad , \quad \sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\mathbf{U}_i - \mu)}$$

Table 3.1 shows the values for $|\mathbf{IN}|$, **U** and **Z** for the above example, and the mean and standard deviation for **U** and **Z**. After these values are determined, a threshold **D** is set, so that every component $\mathbf{C}_i$ with $\mathbf{U}_i > \mathbf{D}$ is identified as utility.

Table 3.1. Example of applying fan-in analysis taken from [Hamou-Lhadj et al. 2005]

	|IN|	U	Z
C1	0	0.00	-1.00
C2	**6**	**1.00**	**2.00**
C3	1	0.17	-0.50
C4	1	0.17	-0.50
C5	3	0.50	0.50
C6	2	0.33	0.00
C7	1	0.17	-0.50
	MEAN	**0.33**	**0**
	STDEV	**0.33**	**1**

After identifying the utility components, the methods within these components can be removed from the traces to reduce the size and make them easier to understand.

3.2.3 Webmining

Zaidman et al. [2005] describes a technique for an easier system comprehension, based on a heuristic to find key classes. The goal is to provide a list of key classes or a visualization of these key classes and their immediately collaborating classes. The software engineer can then use these key classes as starting point for system comprehension.

The technique is split in four steps. The first step is to define an execution scenario, which at least contains the part of the system which is of interest. The next step is a non-selective profiling, which means that all method calls during the execution of the scenario are logged. The resulting traces can become very large, even for small software systems and precisely defined execution scenarios. The third step is to create a compacted call graph from these traces and compute the *hubiness* and *authority* for each node of the graph. This process is based on a webmining-algorithm called HITS, which is explained below. The last step is to interpret the results and identify the key classes.

Figure 3.17 shows an example of a directed graph with five nodes and seven directed edges. Such graphs are the input of the webmining-algorithm described below.

Kleinberg [1999] introduces the concepts of *hub* and *authority*. An *authority* is a page that contains important information. A *hub* is a page that points to good *authorities*. Therefore, a good *authority* is pointed to by good *hubs*. Based on this relation, the HITS algorithm has been developed.

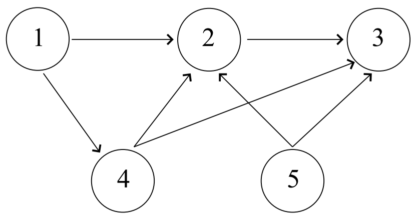


Figure 3.17. Example web-graph taken from [Zaidman et al. 2005]

For every node N_i two values are computed, the *hubiness* h_i and the *authority* a_i:

$$h_i = \sum_{i \to j} w[i,j] \cdot a_j \qquad a_j = \sum_{i \to j} w[i,j] \cdot h_i$$

where $i \to j$ denotes that there is an edge from i to j and $w[i,j]$ is the weight of the edge $i \to j$. In the example graph illustrated in Figure 3.17, the weights are all 1. For this graph the *hubiness* and *authority* values converge to the following:

$h1 = 64$	$h2 = 48$	$h3 = 0$	$h4 = 100$	$h5 = 100$
$a1 = 0$	$a2 = 100$	$a3 = 94$	$a4 = 24$	$a5 = 0$

From these values it can be concluded, that the nodes N_4 and N_5 are good *hubs* and the nodes N_2 and N_3 are good *authorities*. While this webmining-algorithm uses only the links between webpages and not their content, it can be used on any directed graph to identify *hubs* and *authorities*. In our domain *hubs* correspond to classes with coordinating tasks and *authorities* correspond to utilities, as they are used by many other classes. Hence, the *hubs* are the key classes.

A reduction with this technique is possible through slicing, i.e., removing everything from the trace except the method calls from key classes and their immediately collaborating classes.

3.2.4 Stack Depth Limitation

Cornelissen et al. [2007] describes two variants of stack depth limitation. The first variant is a **maximum stack depth limitation**, where all events that occur above a given stack depth threshold d_{max} are omitted. This will remove the detailed utility methods. The second variant is a **minimum stack depth limitation**, where all events below a given threshold d_{min} are omitted. This will remove high-level events. This variant of the technique can split the trace into smaller ones. Figure 3.18 shows an example trace with six objects and seven nested method calls, hence the depth of this trace is seven. Figure 3.19 shows on the left side the result for filtering this trace with a maximum stack depth limitation $d_{max} = 4$, and on the right side the filtered trace with a minimum stack depth limitation $d_{min} = 3$.

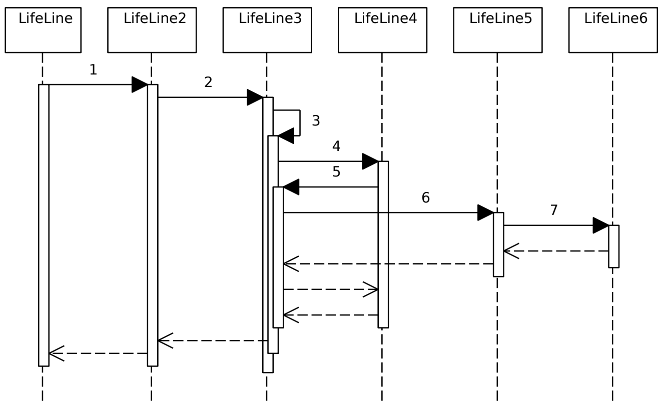


Figure 3.18. Example trace with depth seven

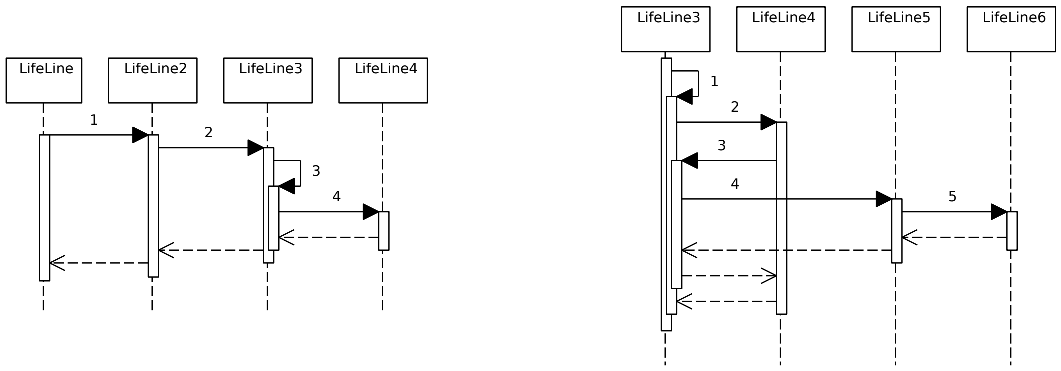


Figure 3.19. Filtered trace with threshold $d_{max} = 4$ on the left and $d_{min} = 3$ on the right

3.3 Language-Based Filtering

Language-based filtering techniques are targeted at the omission of constructs such as getters and setters, private methods, and so forth. The four investigated techniques in this category are Package Filtering, Visibility Specifiers, Getters and Setters and Constructor Hiding.

Figure 3.20 shows a class diagram with five classes grouped in two packages. The package *application.service* includes the class *ServiceClazzA*, which provides a public method *start*, and the class *ServiceClazzB*, which provides the public method *computeXY*. The package *application.common* includes the class *CommonClazz* with an attribute *rate* with public getter and setter methods, a public constructor, and another public and two private methods. Furthermore, the package includes two entity classes each with a public constructor, an attribute and public getter and setter for this attribute.

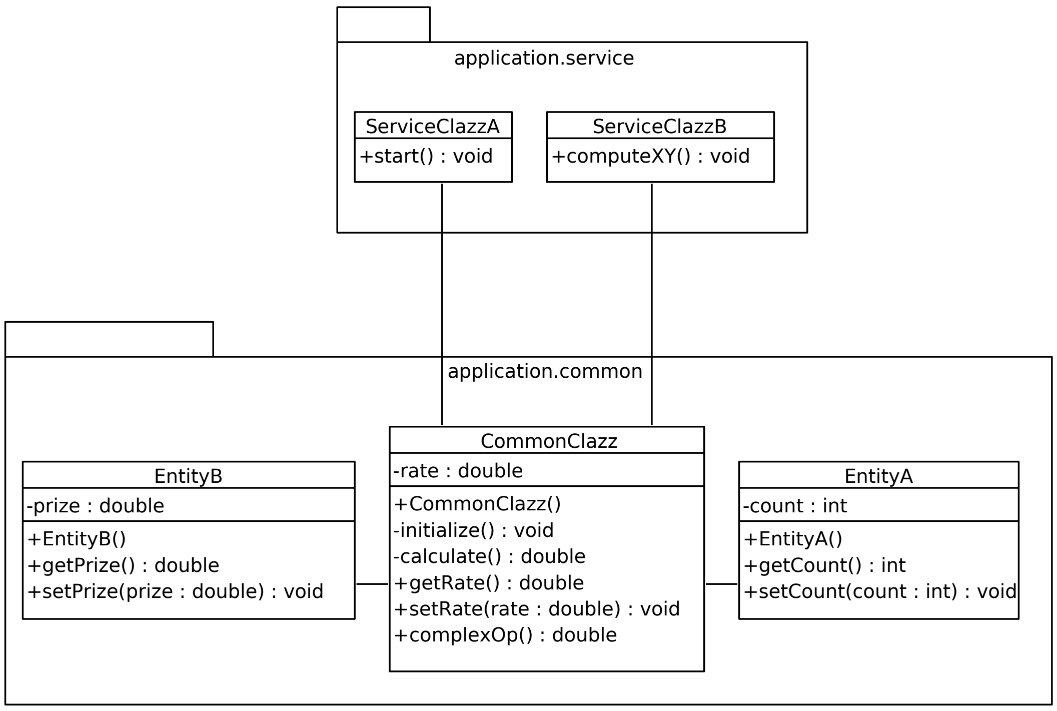


Figure 3.20. Class diagram of an example application

Figure 3.21 shows a possible execution trace as sequence diagram, build on the classes of diagram 3.20. The trace can be divided in two regions. The first region starts with the *start* method of the class *ServiceClazzA* and describes an initialization phase with constructor calls. The second region starts with the *computeXY* method of the class *ServiceClassB* and describes an ordinary control flow with public and private methods and getters and setters. Based on this trace, the language based filtering techniques are described in the following.

3.3.1 Package Filtering

Package filtering means selecting the packages to be included in the analysis [Gargiulo and Mancoridis 2001]. With this technique one can restrict the runtime data of a program on a desired feature. Figure 3.22 shows the trace from Figure 3.21 restricted to the package *application.service*. The most methods are omitted here, only the methods *start* and *computeXY* are left. The trace was reduced from 12 to 2 events.

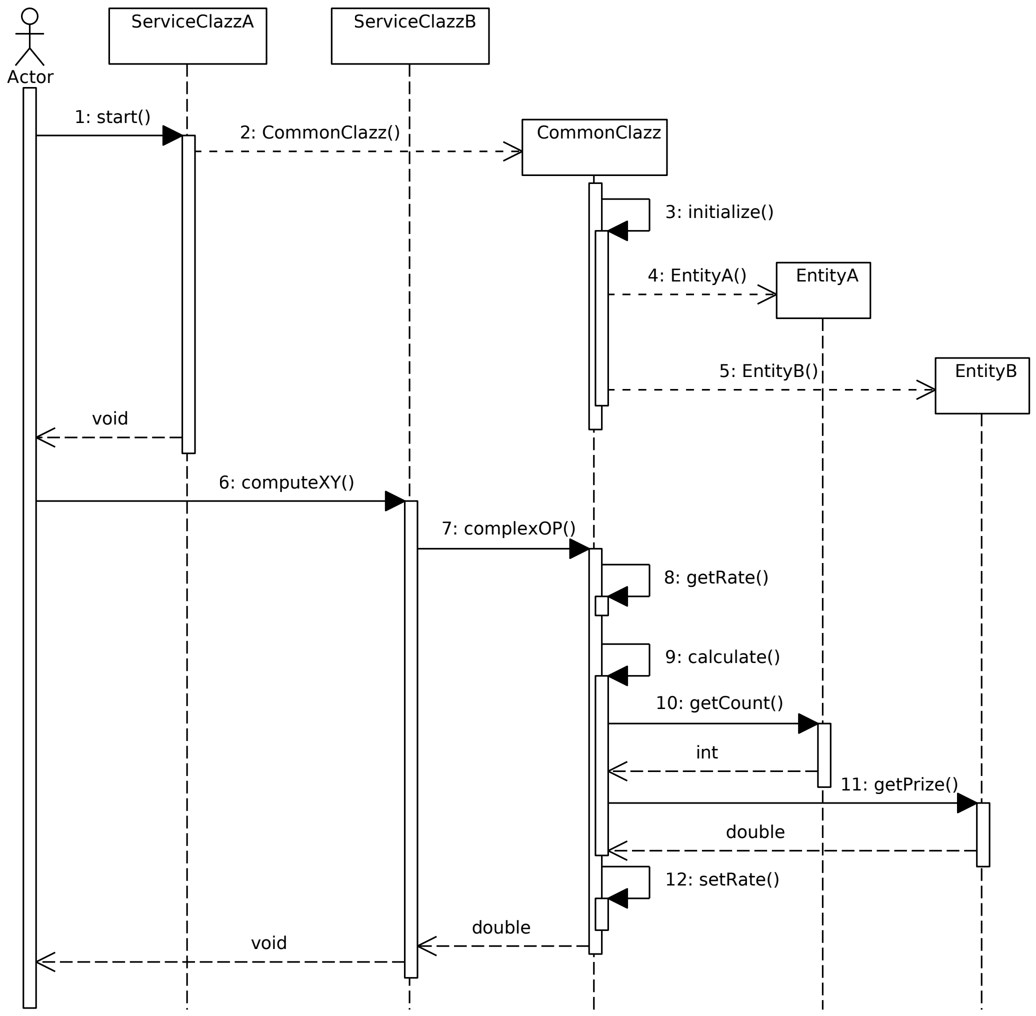


Figure 3.21. Example of an execution trace

3.3.2 Visibility Specifiers

This technique removes method calls with a specific visibility, like private or protected methods [Cornelissen et al. 2008]. Figure 3.23 shows the example trace from Figure 3.21, where the private methods *initialize* and *calculate* are omitted. The trace was reduced from 12 to 10 events.

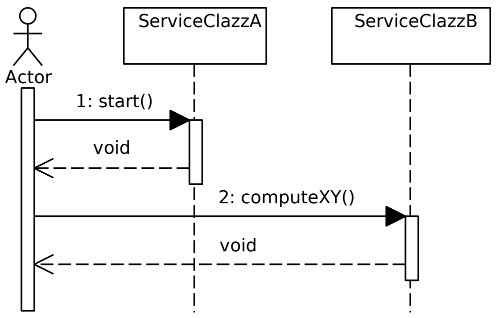


Figure 3.22. Example of package filtering

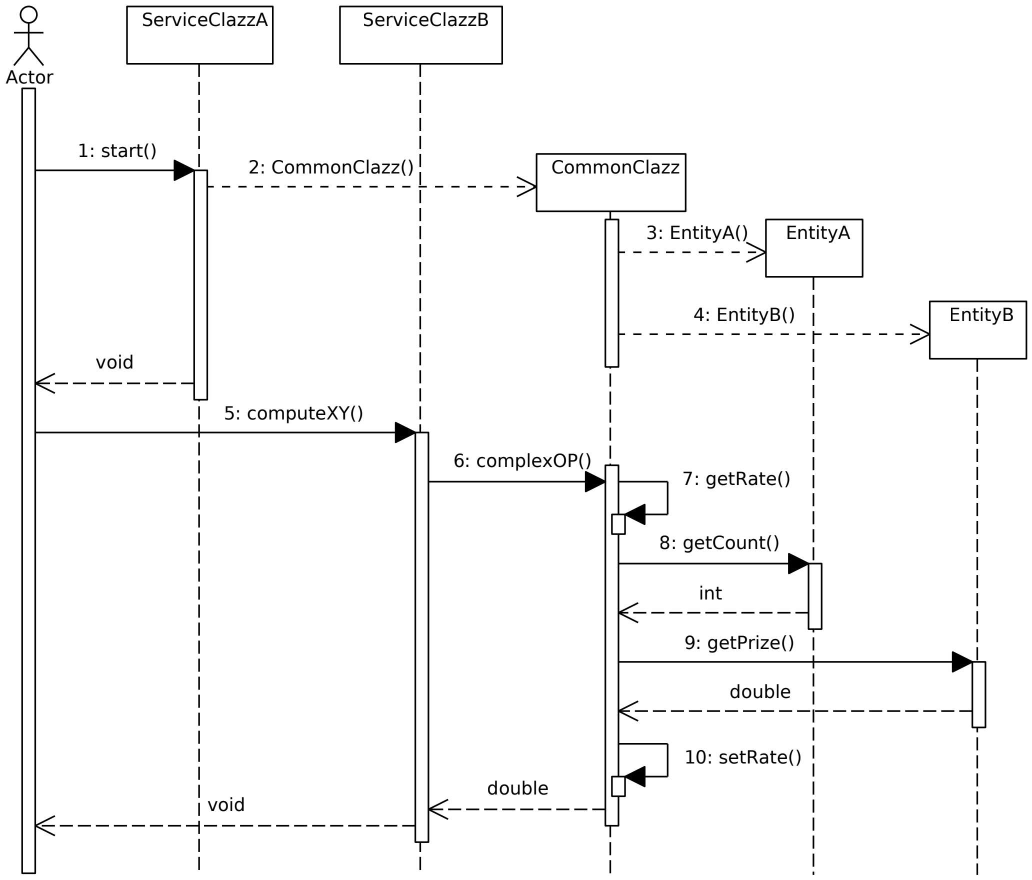


Figure 3.23. Example of visibility specifiers filtering

3.3.3 Getters and Setters

This technique removes the getter and setter methods of the trace. According to Cornelissen et al. [2007] the omission of getters and setters can be applied without loss of essential information. Figure 3.24 shows the example trace without getters and setters. The trace was reduced from 12 to 8 events.

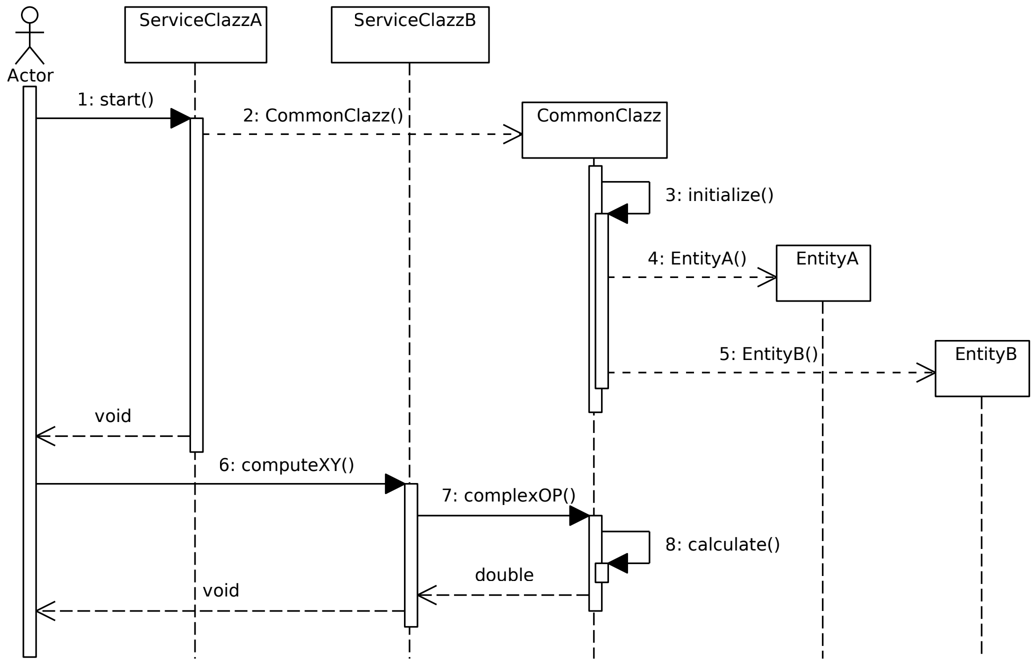


Figure 3.24. Example of getters and setters filtering

3.3.4 Constructor Hiding

This technique removes the constructors and their control flows [Cornelissen et al. 2008]. Such implementation details are not essential for program comprehension [Hamou-Lhadj and Lethbridge 2006]. A refinement of this technique is a selective constructor hiding, where only irrelevant constructors are omitted, like those pertaining to objects that are never used. Figure 3.25 shows this technique applied to the example trace from Figure 3.21. The constructors and their control flows are omitted and the trace was reduced from 12 to 9 events.

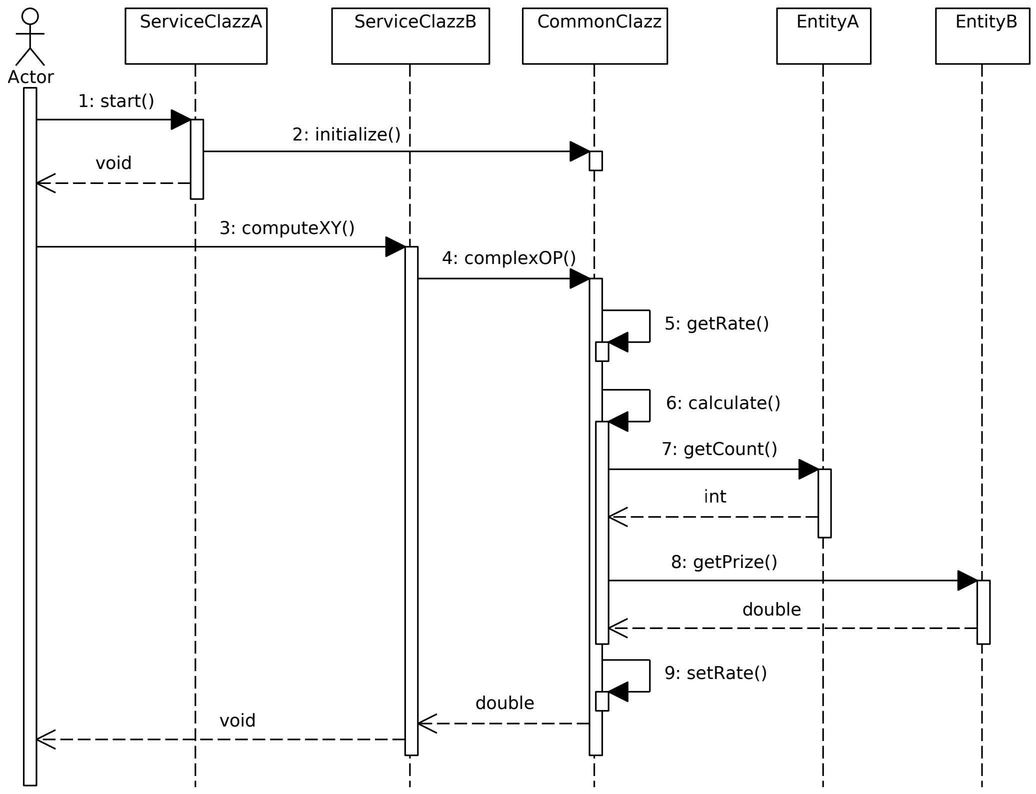


Figure 3.25. Example of constructor hiding

3.4 Ad Hoc

Ad hoc approaches concern the use of 'black-box' techniques that do not consider the trace contents. The two investigated techniques in this category are Sampling and Fragment Selection.

3.4.1 Sampling

Chan et al. [2003] describe six sampling variants, three memory and three control flow variants. The first one for memory events, like object allocation or deallocation, is to take every n-th memory event. The second one is to take the first memory event that occurs during or after every n-th timestamp. The third one is to do the first or the second one, and snapshot the call stack before each sampled memory event. Consecutive snapshots are then compared to determine which methods have been entered or exited, thereby providing control-flow context for the memory events. For control-flow events, the first variant is to take every n-th control-flow event, like method entry or method exit. The second variant is to take a snapshot of the call stack every n-th event, and compare consecutive snapshots to determine which methods have been entered or exited. The third variant is the same as before, except that the snapshots are taken every n-th timestamp. Cornelissen et al. [2008] uses the first of the control flow variants for their experiments.

3.4.2 Fragment Selection

The goal of fragment selection is to locate the code relevant to a particular feature [Wong et al. 1999]. Fragments of a trace can be selected in various ways. Vasconcelos et al. [2005] use use-case scenarios and message depth level to slice a trace. Hamou-Lhadj and Lethbridge [2004] mention a slicing technique where all activation paths of an object or a method are kept, while the rest of the trace is removed.

3.5 Own Approach

Since the landscape model of ExplorViz is not using the complete information of a trace, we developed a reduction technique for ExplorViz, where only the necessary informations are preserved. This technique, which we call *Tree Summarization*, is described in this section.

3.5.1 Tree Summarization

On the lowest, most detailed level, ExplorViz shows the involved classes and the method calls as connections between them. To extract these connections, we consider a trace as a tree, i.e., as a special graph. Now we extract each occurring edge, which is a pair of a caller and a callee. Edges that occur more than once are combined, through merging the callee statistics. From this collection of edges a new trace is built. Since inner class method calls are not shown, edges with caller equal to callee are removed. Figure 3.26 shows an example trace on the left, the resulting extracted edges in the middle, and the reduced trace on the right. This technique covers the criteria *Identity*, *Object Identity*, *Message Structure*, *Repetition*, *Associativity* and *Commutativity* from Execution Pattern Notation.

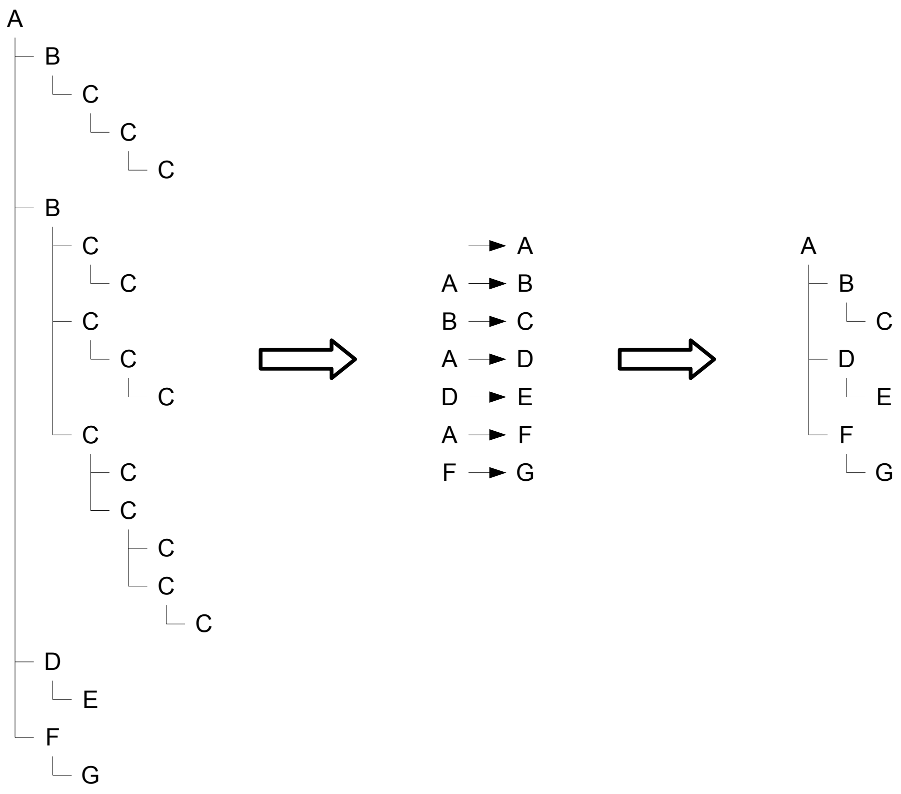


Figure 3.26. Example of an execution trace and its reduction through tree summarization.

Chapter 4

Assessment Criteria for Trace Reduction Techniques in Respect to Online Trace Visualization

In this chapter, we describe four assessment criteria for trace reduction techniques with respect to online trace visualization. The four criteria are reduction rate, information preservation, parallelizability and performance.

4.1 Reduction Rate

The reduction rate is the percentage by which the initial trace has been reduced. The size of a trace is measured by the number of records it includes. Let N be the size of the trace before filtering and M the size of the trace after filtering. The reduction rate R is then defined as

$$R = 1 - \frac{M}{N} \tag{4.1}$$

Since M should be smaller than N, R is between 0 and 1, where 0 is no reduction and 1 is complete reduction, which means nothing is left.

4.2 Information Preservation

Information preservation is the percentage to which information from the original trace are preserved after reduction. To take account to the context of online trace visualization with ExplorViz, we introduce weightings for different information.

A trace T consists of a list of records $R_1, R_2, \ldots, R_n$. Since the records have different importance for the online trace visualization, we define β_i as a weighting of the importance of R_i. Each β is between 0 and 1, where 1 means very important and 0 means unimportant. A record R_i consists of a list of attributes $A_{i1}, A_{i2} \ldots A_{i10}$ explained in Section 2.1. The attributes of a record also differ in their importance, so we define α_{ij} as a weighting of the importance of A_{ij}. As shown in Table 4.1a, we distinguish between important and less important attributes. An important attribute has a 10 times higher weighting than an unimportant. The *traceId* and the *orderIndex* are only necessary for the reconstruction of the trace. The *timestamp* is necessary for calculating the response times. This calculation is done during the reconstruction. Since the reduction takes place after the reconstruction, these three attributes are less important. The *objectIds* are collected during the reduction in the *objectSet*. Hence, the *objectId* also is less important. The other attributes are important because they are necessary for the visualization. To simplify the computation of the information value, we set $\sum_{j=1}^{11} \alpha_{ij} = 1$, if all attributes are present.

For our monitoring records(see Figure 2.1) the weightings are listed in Table 4.1b. For getters and setters we set $\beta = 0.2$ because they do not significantly contribute to program comprehension and furthermore generally do not include complex calculations. For non public methods we set $\beta = 0.3$ because in most cases they only cover implementation details. For methods where the caller is equal to the callee, i.e., a method call within a class, we set $\beta = 0.1$ because such calls are not visualized in ExplorViz. For constructors we set $\beta = 0.5$ because object instantiation is usually not that important for program comprehension. For all other methods we set $\beta = 1.0$. The information I of a trace is then measured as follows:

$$I = \sum_{i=1}^{n} \beta_i \cdot \text{count} \cdot \sum_{j=1}^{11} \alpha_{ij} \tag{4.2}$$

Let I_{before} be the information of the original trace and I_{after} the information of the reduced trace. The information preservation IP is then computed by

$$IP = \frac{I_{after}}{I_{before}} \tag{4.3}$$

Table 4.1. Overview of the weightings of the attributes and the event types

Index	Name	Weighting
1	timestamp	less important
2	traceId	less important
3	orderIndex	less important
4	hostname	important
5	application	important
6	objectId	less important
7	operationSignature	important
8	count	important
9	sumResponsetime	important
10	sqaredSum	important
11	objectSet	important

(a) Indexes and weightings of the attributes

Type	Weighting
caller = callee	0.1
getter & setter	0.2
non-public	0.3
constructor	0.5
rest	1.0

(b) Weightings of the event types

4.3 Parallelizability

Parallelizability is a measure of the ability to parallelize a technique. It is not only dependent on the reduction technique but also on the data structure of the trace. Therefore, we divide the trace into subtraces. In our context, a subtrace is a part of a trace forming also a valid trace, i.e., for each *BeforeRecord*, the corresponding *AfterRecord* is included. In addition the caller method of the first event should be included. Figure 4.1 shows an example trace and a possible division into seven subtraces.

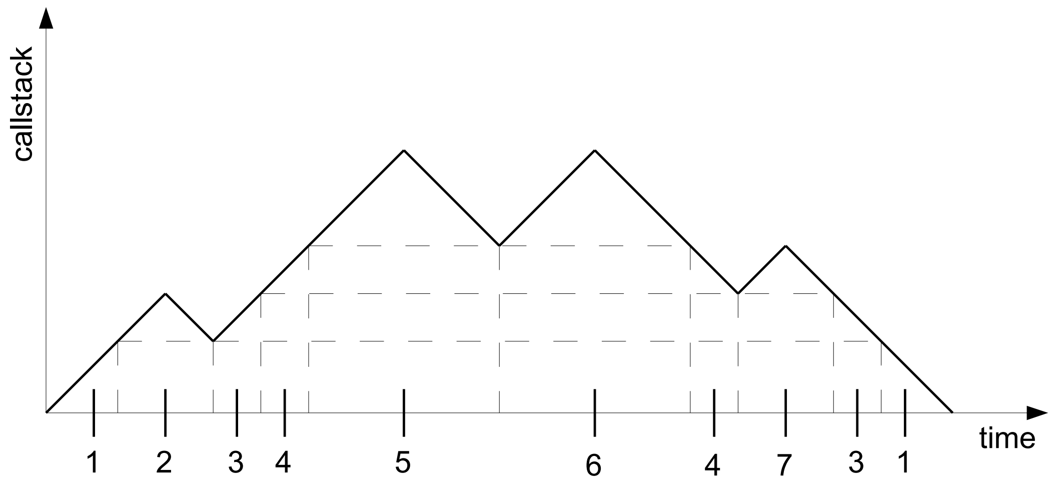


Figure 4.1. Example trace divided in subtraces

Since the division of the trace can influence the reduction rate, we assign an estimated value between 0 and 1 as a measure for parallelizability. 0 means a parallelization is impossible. 1 means a parallelization is possible and won't have an influence on the reduction rate. A value between 0 and 1 means a parallelization is possible but the stronger the effect on the reduction rate, the lower the value.

4.4 Performance

To measure the performance of a reduction technique in the context of ExplorViz, we define performance as the ration of the processing time without reduction to the processing time using a reduction. Therefore , a value greater than 1 represents an acceleration, while a value less than 1 describes a slowdown.

Chapter 5

Theoretical Evaluation of Trace Reduction Techniques

In this chapter the trace reduction techniques from Chapter 3 are evaluated based on the criteria from Chapter 4 and an example trace from PubFlow. Table 5.1 gives an overview of our theoretical evaluation results. For each trace reduction technique the reduction rate, the information preservation, and the parallelizability is indicated. The order is not judgmental.

Table 5.1. Overview of our theoretical evaluation results

Techniques	**Reduction Rate**	**Inf. Preservation**	**Parallelizability**
Iteration	0.05	1.00	0.7
Associativity	0.40	0.87	1.0
Pattern Summarization	0.37	0.94	0.5
Object and Event Clustering	0	1	0
Monotone Subsequence Sum.	0.78	0.28	1.0
Frequency Spectrum Analysis	0	1	0
Utilityhood Measure	0	1	0
Webmining	0	1	0
Stack Depth Limitation	0.36	0.65	0.0
Package Filtering	0.59	0.58	1.0
Visibility Specifiers	0.51	0.69	1.0
Getters and Setters	0.42	0.84	1.0
Constructor Hiding	0.11	0.90	1.0
Sampling	0.80	0.25	1.0
Fragment Selection	0.68	0.12	1.0
Tree Summarization	0.93	0.99	0.9

Before reducing the trace, all attributes are present and the count attribute is always 1.

The computation of the information value simplifies to

$$I = \sum_{i=1}^{n} \beta_i \cdot count \cdot \sum_{j=1}^{11} \alpha_{ij} = \sum_{i=1}^{n} \beta_i$$

The example trace consists of $\mathbf{N} = \mathbf{3124}$ events. For 1249 of these events the caller and callee classes are the same. 631 of the remaining 1875 events are getters and setters. 667 of the remaining 1244 events are not public. The remaining 577 events include 112 constructors. This leads to the following information value for this trace:

$$I_{before} = 1249 \cdot 0.1 + 631 \cdot 0.2 + 667 \cdot 0.3 + 112 \cdot 0.5 + 465 \approx 972.2$$

This value is used for the information preservation calculation in each technique.

5.1 Summarization

The eight criteria for equivalent patterns from Execution Pattern Notation lead to a few reduction techniques. The criteria *Identity*, *Class Identity*, *Message Structure*, *Recursion*, and *Commutativity* are covered by Pattern Summarization and Tree Summarization. *Depth-Limiting* is treated in Stack Depth Limitation. *Polymorphism* is not covered, since the necessary information for this purpose are not included in the trace data. Instead of Execution Pattern Notation, *Iteration*, the second part of *Repetition*, and *Associativity* are evaluated in this section. Furthermore, the techniques pattern summarization, object and event clustering, and monotone subsequence summarization are evaluated in this section.

5.1.1 Iteration

For this technique we only consider simple iterations, i.e., directly consecutive methods with the same signature.

Reduction Rate

Applying this technique to the example trace, 193 of the 3124 events are summed up to 52 events. Hence, the reduced trace consists of $\mathbf{M} = \mathbf{3124} - \mathbf{193} + \mathbf{52} = \mathbf{2983}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{2983}{3124} \approx 0.05$$

Information Preservation

145 events, where the caller and the callee classes are the same, are summed up to 35 events. 28 getter and setter methods are summed up to 10 events. 12 non-public methods

are summed up to 3 events. 2 constructors are summed up to 1. 6 uncategorized methods are summed up to 3 events.

When summarizing methods with the same signature within a trace, the three attributes *timestamp*, *orderIndex*, and *objectId* are lost. Since these attributes are less important, it is

$$\sum_{j=1}^{11} \alpha_{ij} = \frac{71}{74}$$

The information value of the reduced trace is then calculated as follows:

$$I_{after} = (1104 + 145 \cdot \frac{71}{74}) \cdot 0.1 + (603 + 28 \cdot \frac{71}{74}) \cdot 0.2 + (655 + 12 \cdot \frac{71}{74}) \cdot 0.3 \tag{5.1}$$

$$+ (110 + 2 \cdot \frac{71}{74}) \cdot 0.5 + (459 + 6 \cdot \frac{71}{74}) \approx 968.2 \tag{5.2}$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{968.2}{972.2} \approx 1.00$$

Parallelizability

We rate the parallelizability with 0.7 because iterations inside a subtrace can be summarized, so a parallelization is possible. However, there is a possibility that iterations are separated, when dividing the trace. Hence, the smaller the subtraces, the lower the reduction rate of this technique.

5.1.2 Associativity

This technique removes all method calls, which are called within a class, i.e., where the caller class is equal to the callee class.

Reduction Rate

Applying this technique to the example trace, 1249 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M = 3124 - 1249 = 1875}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{1875}{3124} \approx 0.40$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$$I_{after} = 0 \cdot 0.1 + 631 \cdot 0.2 + 667 \cdot 0.3 + 112 \cdot 0.5 + 465 \approx 847.3$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{847.3}{972.2} \approx 0.87$$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation.

5.1.3 Pattern Summarization

This technique has two parameters, distance and nesting level, which can affect the degree of reduction. The larger the both values, the greater the reduction can be. However, both parameters are multipliers for the running time, i.e. the distance is doubled, the running time is doubled.

Table 5.2 shows rates of reduction and information preservation for different distances and nesting levels applied to the example trace. Distance 310 and nesting level 3 leads in our example trace to the highest reduction. This distance corresponds to about 5% of the trace length. As seen in the table, shorter distances already provide good results. Distance and nesting level can thus be kept constant and small, which is important for the running time of the algorithm.

Reduction Rate

Applying this technique to the example trace, 1558 of the 3124 events are summed up to 395 events. Hence, the reduced trace consists of $\mathbf{M = 3124 - 1558 + 395 = 1961}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{1961}{3124} \approx 0.37$$

Information Preservation

713 events, where the caller and the callee classes are the same, are summed up to 150 events. 297 getter and setter methods are summed up to 96 events. 353 non-public methods are summed up to 89 events. 42 constructors are summed up to 12. 153 uncategorized methods are summed up to 48 events.

Table 5.2. Information preservation and reduction rate for the example trace using pattern summarization with different values for distance and nesting level

Distance	**Nesting Level**	**Reduction Rate**	**Inf. Preservation**
310	**3**	**0.37**	**0.94**
310	2	0.37	0.94
310	1	0.34	0.94
110	3	0.23	0.95
110	2	0.22	0.95
110	1	0.19	0.95
40	2	0.14	0.96
40	1	0.13	0.97
20	2	0.11	0.97
20	1	0.10	0.97
10	2	0.09	0.98
10	1	0.09	0.98

Such as in the technique *Iteration* methods with the same signature within a trace are summarized. Hence, here also holds

$$\sum_{j=1}^{11} \alpha_{ij} = \frac{71}{74}$$

The information value of the reduced trace is then calculated as follows:

$$\begin{aligned} I_{after} =& (536 + 713 \cdot \frac{71}{74}) \cdot 0.1 + (334 + 297 \cdot \frac{71}{74}) \cdot 0.2 + (314 + 353 \cdot \frac{71}{74}) \cdot 0.3 \\ &+ (70 + 42 \cdot \frac{71}{74}) \cdot 0.5 + (312 + 153 \cdot \frac{71}{74}) \approx 913.8 \end{aligned}$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{913.8}{972.2} \approx 0.94$$

Parallelizability

We rate the parallelizability with 0.5 because patterns inside a subtrace can be summarized, so a parallelization is possible. However, there is a possibility that patterns or their repetitions are separated, when dividing the trace. Hence, the smaller the subtraces, the lower the reduction rate of this technique. Especially, because the points at which the trace can be divided into subtraces are also possible points for repetitions. For example the subtraces 5 and 6 of the example in Figure 4.1 could match the same pattern.

5.1.4 Object and Event Clustering

Object and event clustering is no reduction technique, so we rate the reduction rate with $R = 0$ and the information preservation with $IP = 1$. Furthermore, we rate the parallelizability with 0 because there is nothing to parallelize.

5.1.5 Monotone Subsequence Summarization

Reduction Rate

Applying this technique to the example trace, 2429 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M} = \mathbf{3124} - \mathbf{2429} = \mathbf{695}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{695}{3124} \approx 0.78$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$$I_{after} = 105 \cdot 0.1 + 264 \cdot 0.2 + 138 \cdot 0.3 + 33 \cdot 0.5 + 155 \approx 276.2$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{276.2}{972.2} \approx 0.28$$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation. Whether the first record must be removed, can be determined via its index and the index of the caller.

5.2 Metrics-Based Filtering

In this section the four techniques frequency spectrum analysis, utilityhood measure, webmining, and stack depth limitation are evaluated.

5.2.1 Frequency Spectrum Analysis

Frequency spectrum analysis is no reduction technique, so we rate the reduction rate with $R = 0$ and the information preservation with $IP = 1$. Furthermore, we rate the parallelizability with 0 because there is nothing to parallelize.

5.2.2 Utilityhood Measure

Utilityhood Measure is no reduction technique, so we rate the reduction rate with $R = 0$ and the information preservation with $IP = 1$. Furthermore, we rate the parallelizability with 0 because there is nothing to parallelize.

5.2.3 Webmining

Webmining is no reduction technique, so we rate the reduction rate with $R = 0$ and the information preservation with $IP = 1$. Furthermore, we rate the parallelizability with 0 because there is nothing to parallelize.

5.2.4 Stack Depth Limitation

We choose the maximum stack depth limitation variant for this evaluation. The example trace has a maximum stack depth of 22. Table 5.3 shows the reduction rate and the information preservation for different thresholds. We choose a stack depth of 11 for the evaluation because the information preservation of the reduced trace is relatively high, while the size is significantly reduced.

Table 5.3. Information preservation and reduction rate for the example trace using stack depth limitation with different values for the stack depth

Stack Depth	Reduction Rate	Inf. Preservation
5	0.99	0.02
8	0.85	0.15
10	0.61	0.45
11	**0.36**	**0.65**
12	0.23	0.75
14	0.09	0.92
17	0.04	0.98

Reduction Rate

Applying this technique to the example trace, 1139 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M = 3124 - 1139 = 1985}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{1985}{3124} \approx 0.36$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$I_{after} = 673 \cdot 0.1 + 473 \cdot 0.2 + 468 \cdot 0.3 + 87 \cdot 0.5 + 284 \approx 629.8$

This leads to the following information preservation value:

$IP = \frac{I_{after}}{I_{before}} \approx \frac{629.8}{972.2} \approx 0.65$

Parallelizability

We rate the parallelizability with 0.0 because this technique needs the stack depth of the original trace, which is not detectable in a subtrace. It is imaginable to enrich the subtraces with this information, this would lead to a parallelizability value of 1.0.

5.3 Language-Based Filtering

In this section the four techniques package filtering, visibility specifiers, getters and setters, and constructor hiding are evaluated.

5.3.1 Package Filtering

Table 5.4 shows the reduction rate and the information preservation for almost each package in the example trace. It is difficult to decide automatically which package is important and which is not. For the evaluation we choose the package *org.apache.xerces.dom* because it has the largest sum of reduction rate and information preservation in this example.

Reduction Rate

Applying this technique to the example trace, 1832 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M = 3124 - 1832 = 1292}$ events. This leads to the following reduction rate:

$R = 1 - \frac{M}{N} = 1 - \frac{1292}{3124} \approx 0.59$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated

Table 5.4. Information preservation and reduction rate for each package in the example trace

Packagenames	Reduction Rate	Inf. Preservation
de.pubflow	0.00	0.99
de.pubflow.common.persistence	0.00	1.00
de.pubflow.common.properties	0.00	1.00
org	1.00	0.00
org.apache	0.82	0.18
org.apache.log4j	0.18	0.71
org.apache.log4j.config	0.01	1.00
org.apache.log4j.helpers	0.08	0.85
org.apache.log4j.or	0.00	1.00
org.apache.log4j.spi	0.06	0.90
org.apache.xerces	0.65	0.47
org.apache.xerces.dom	**0.59**	**0.58**
org.apache.xerces.impl.dtd	0.04	0.95
org.apache.xerces.impl.dtd.models	0.04	0.95
org.apache.xerces.jaxp	0.00	1.00
org.apache.xerces.parsers	0.02	0.94
org.hsqldb	0.16	0.84
org.hsqldb.lib	0.01	0.97
org.hsqldb.lib.java	0.00	1.00
org.hsqldb.map	0.09	0.90
org.hsqldb.persist	0.01	0.99
org.hsqldb.resources	0.00	1.00
org.hsqldb.server	0.04	0.98
org.postgresql	0.00	1.00
org.slf4j	0.02	0.98
org.slf4j.helpers	0.00	1.00
org.slf4j.impl	0.01	0.99

as follows:

$I_{after} = 379 \cdot 0.1 + 316 \cdot 0.2 + 130 \cdot 0.3 + 92 \cdot 0.5 + 375 \approx 561.1$

This leads to the following information preservation value:

$IP = \frac{I_{after}}{I_{before}} \approx \frac{561.1}{972.2} \approx 0.58$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation.

5.3.2 Visibility Specifiers

Table 5.5 shows the reduction rate and the information preservation of the reduced trace, after removing one of the four visibility modifiers *private*, *protected*, *package*, or *public*, or a combination of the first three, named *nonpublic*. For the evaluation we choose the last one because we regard public method calls as a more important part of the trace than the non public method calls.

Table 5.5. Information preservation and reduction rate for each visibility modifier of the example trace

Removed Visibility	Reduction Rate	Inf. Preservation
private	0.17	0.94
protected	0.14	0.91
package	0.19	0.84
public	0.49	0.38
nonpublic	**0.51**	**0.69**

Reduction Rate

Applying this technique to the example trace, 1602 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M} = \mathbf{3124} - \mathbf{1602} = \mathbf{1522}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{1522}{3124} \approx 0.51$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$$I_{after} = 366 \cdot 0.1 + 578 \cdot 0.2 + 0 \cdot 0.3 + 112 \cdot 0.5 + 466 \approx 674.2$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{674.2}{972.2} \approx 0.69$$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation.

5.3.3 Getters and Setters

Reduction Rate

Applying this technique to the example trace, the 1299 getter and setter method calls are removed. Hence, the reduced trace consists of $\mathbf{M} = \mathbf{3124} - \mathbf{1299} = \mathbf{1825}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{1825}{3124} \approx 0.42$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$$I_{after} = 424 \cdot 0.1 + 0 \cdot 0.2 + 811 \cdot 0.3 + 114 \cdot 0.5 + 476 \approx 818.7$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{818.7}{972.2} \approx 0.84$$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation.

5.3.4 Constructor Hiding

Reduction Rate

Applying this technique to the example trace, the 340 constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M} = \mathbf{3124} - \mathbf{340} = \mathbf{2784}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{2784}{3124} \approx 0.11$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$$I_{after} = 1190 \cdot 0.1 + 631 \cdot 0.2 + 448 \cdot 0.3 + 0 \cdot 0.5 + 495 \approx 880.6$$

This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx \frac{880.6}{972.2} \approx 0.91$$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation.

5.4 Ad Hoc

In this section the two techniques sampling and fragment selection are evaluated.

5.4.1 Sampling

Like Cornelissen et al. [2008], we choose the simplest of the described variants, and take every *n-th* control flow event. Table 5.6 shows the reduction rate and the information preservation for the example trace for a few sampling distances.

Table 5.6. Information preservation and reduction rate for a few sampling distances

Dinstance	Reduction Rate	Inf. Preservation
5	**0.80**	**0.25**
10	0.90	0.12
20	0.95	0.06
100	0.99	0.01

Reduction Rate

Applying this technique to the example trace, 2499 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M = 3124 - 2499 = 625}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{625}{3124} \approx 0.80$$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$I_{after} = 44 \cdot 0.1 + 211 \cdot 0.2 + 237 \cdot 0.3 + 18 \cdot 0.5 + 115 \approx 241.7$

This leads to the following information preservation value:

$IP = \frac{I_{after}}{I_{before}} \approx \frac{241.7}{972.2} \approx 0.25$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation. The records can be chosen based on the index.

5.4.2 Fragment Selection

For this technique we choose the most frequented object of the example trace and remove the rest of the trace.

Reduction Rate

Applying this technique to the example trace, 2117 method and constructor calls are removed. Hence, the reduced trace consists of $\mathbf{M = 3124 - 2117 = 1007}$ events. This leads to the following reduction rate:

$R = 1 - \frac{M}{N} = 1 - \frac{1007}{3124} \approx 0.68$

Information Preservation

The events in the reduced trace are not changed by this technique, so only the information of the removed events are lost. The information value of the reduced trace is then calculated as follows:

$I_{after} = 890 \cdot 0.1 + 89 \cdot 0.2 + 22 \cdot 0.3 + 3 \cdot 0.5 + 3 \approx 117.9$

This leads to the following information preservation value:

$IP = \frac{I_{after}}{I_{before}} \approx \frac{117.9}{972.2} \approx 0.12$

Parallelizability

We rate the parallelizability with 1.0 because this technique can be applied to subtraces without any limitation.

5.5 Own Approach

In this section we evaluate the tree summarization.

5.5.1 Tree Summarization

Reduction Rate

Applying this technique to the example trace, the 1249 method and constructor calls, where the caller and callee classes are equal, are removed. Furthermore, 1813 events are summed up to 153 events. Hence, the reduced trace consists of $\mathbf{M = 3124 - 1249 - 1813 + 153 = 215}$ events. This leads to the following reduction rate:

$$R = 1 - \frac{M}{N} = 1 - \frac{215}{3124} \approx 0.93$$

Information Preservation

The information preservation for this technique is ambiguous. Depending on how the methods are combined, the information preservation value for the example trace differs from 0.60 to 1.37. For the evaluation we choose the means of these values. This leads to the following information preservation value:

$$IP = \frac{I_{after}}{I_{before}} \approx 0.99$$

Parallelizability

We rate the parallelizability with 0.9. There are two possibilities to parallelize this technique. The fist is to apply the technique unmodified to the subtraces. The resulting reduction is considerably reduced. The second way is to divide the technique into two parts. The first part forms in parallel collections of edges from the subtraces. The second part merges these collection and builds the resulting trace.

5.6 Overall Result

To compare the techniques we form an overall measure OM of the reduction rate R, the information preservation IP and the parallelizability P. We have weighted these values to express their difference importance. We then determined their Euclidean norm.

$$OM = \sqrt{(0.8 \cdot R)^2 + IP^2 + (0.4 \cdot P)^2} \quad (5.3)$$

Table 5.7 shows an ordered list of the evaluated reduction techniques with $R > 0$.

Table 5.7. Overall Result

Place	Technique	Overall Measure
1.	Tree Summarization	1.29
2.	Iteration	1.04
3.	Associativity	1.01
4.	Pattern Summarization	1.01
5.	Getters and Setters	0.99
6.	Constructor Hiding	0.99
7.	Visibility Specifiers	0.90
8.	Package Filtering	0.85
9.	Sampling	0.80
10.	Monotone Subsequence Summarization	0.79
11.	Fragment Selection	0.69
12.	Stack Depth Limitation	0.62

Chapter 6

Implementation

In this Chapter we give an overview of the implemented software artifacts.

6.1 Selection

For our practical evaluation we choose *Tree Summarization*, *Iteration*, *Associativity*, *Pattern Summarization*, *Getters and Setters*, and *Constructor Hiding*. Thus, we choose the top six reduction techniques from our theoretical evaluation.

6.2 Design

Figure 6.1 gives an overview of the new implemented classes and the package structure in the worker component. The gray-shaded classes illustrate the dependencies to existing classes.

We also implement the other six reduction techniques for testing purposes, they are in the package *filter.reduction.evaluation* which is not illustrated here.

Figure 6.2 shows the new classes of the external-monitoring-logs-adapter component. The class *CornelissenImportMain* includes a main method to start reading a trace, and the filenames of the available traces, which are stored in the examples directory in this project.

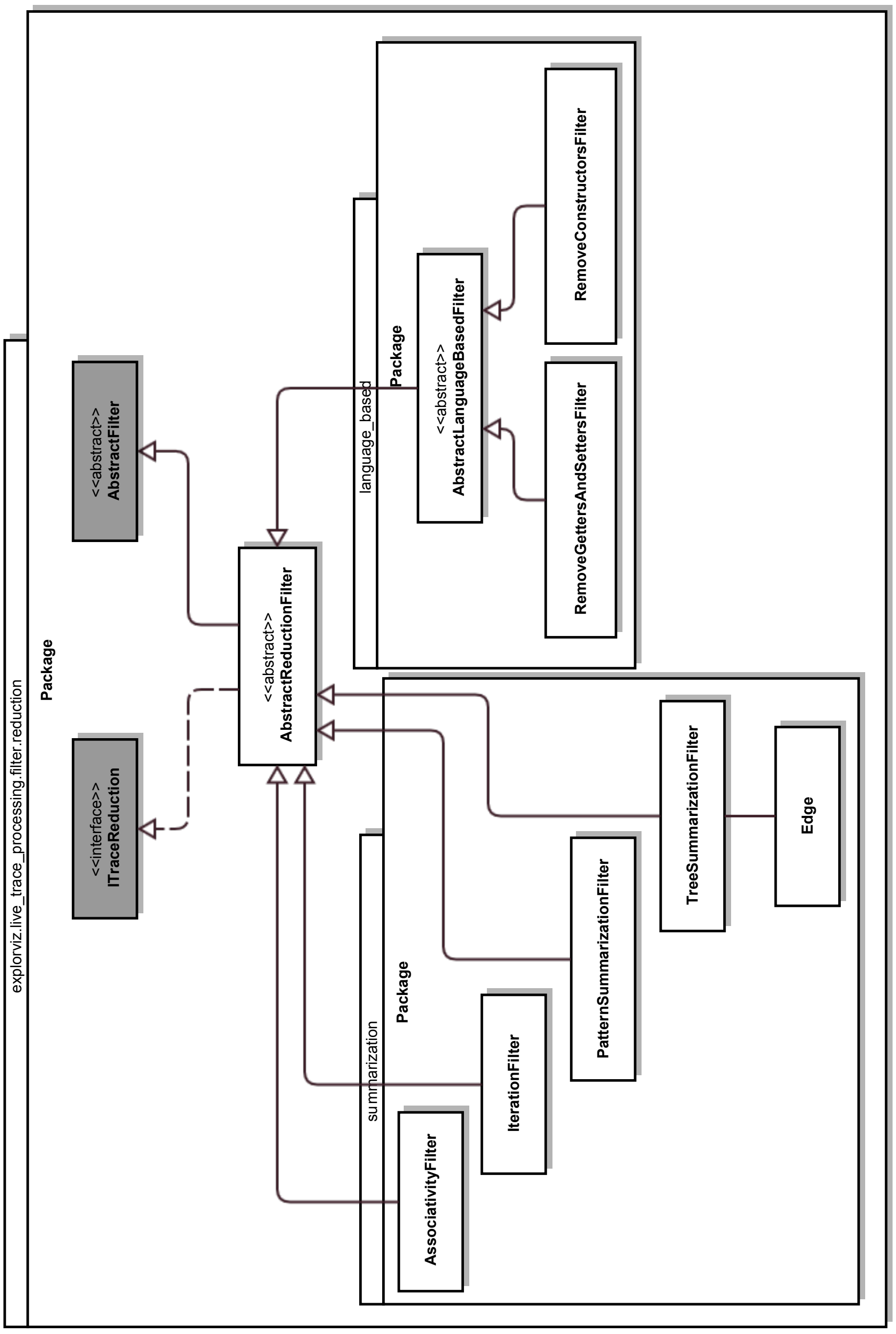


Figure 6.1. Class diagram of the new classes in the worker component

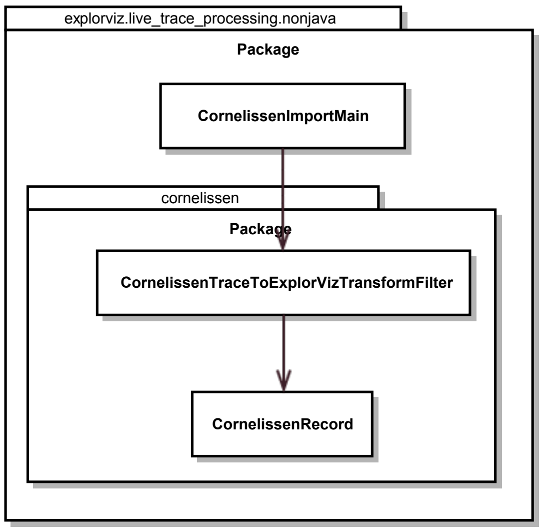

explorviz.live_trace_processing.nonjava
Package
CornelissenImportMain
cornelissen
Package
CornelissenTraceToExplorVizTransformFilter
CornelissenRecord

Figure 6.2. Class diagram of the new classes in the external-monitoring-logs-adapter component

Chapter 7

Practical Evaluation of Trace Reduction Techniques

In this chapter we evaluate the chosen trace reduction techniques with practical examples. In Section 7.1 we describe the example traces which we use for our evaluation. In Section7.2 we present our experimental setup which leads to the results in Section 7.3. Section 7.4 discusses the results, while Section 7.5 shows the limits of the experiment.

7.1 Scenarios

For our practical evaluation we build a test set that consists of six different execution traces from four different Java systems. We take three example traces from the test set of Cornelissen et al. [2008], the example trace from our theoretical evaluation, and two more traces from PubFlow.

- ▻ **pubflow-1** is the smallest trace with 1341 method calls.
- ▻ **pubflow-2** is the example of our theoretical evaluation and consists of 3124 method calls.
- ▻ **pubflow-3** is the largest example trace from PubFlow with 8912 method calls.
- ▻ The processing of a small input file of 50 lines of commented Java code with Checkstyle lead to the **checkstyle** trace with 31237 method calls.
- ▻ The **pacman** trace with 139582 method calls reflects the start of a game, several player and monster movements, player death, start of a new game, and quit.
- ▻ The **jhotdraw** trace consists of 161088 method calls and was build by the creation of a new drawing in which five different figures were inserted, after which the drawing was closed. This process was repeated two times.

7.2 Experimental Setup

Figure 7.1 shows a part of the data flow in ExplorViz. A trace is reconstructed from monitoring data, then this trace is reduced by one of the reduction techniques described in this thesis. The reduced trace updates the landscape model. To evaluate the effectiveness of our chosen trace reduction techniques, we set four measuring points. With these measuring points we define three phases. The first phase, between measuring point 1 and measuring point 2, measures the processing time of reducing the trace, and is called **reduction**. The next phase, between measuring point 2 and measuring point 3, we call **transfer**. The last phase, between measuring point 3 and measuring point 4, we call **model update**. The difference between measuring point 4 and measuring point 1 is the **total duration**.

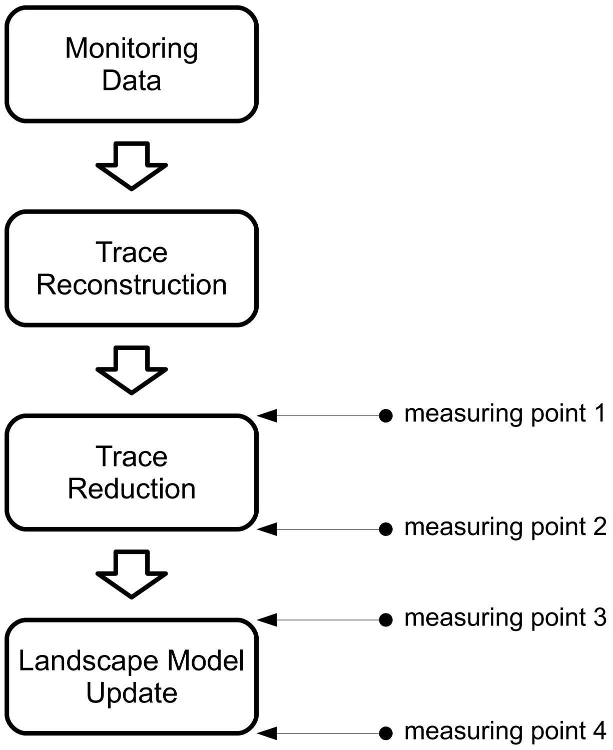


Figure 7.1. Measuring points for performance evaluation

The experiment: For each trace, ExplorViz was started in a Tomcat locally, and the trace was sent to ExplorViz 50 times in a row with a few seconds apart.

The following performance measurements are executed on a contemporary commercial off-the-shelf desktop computer (AMD FX(tm)-4100 Quad-Core, 4 cores at 3.60 GHz, 8 GB RAM, Windows 7 Professional, Java HotSpot(TM) 64-Bit Server VM 1.7).

7.3 Results

In this section the measurement results are presented for the six chosen techniques *Tree Summarization*, *Iteration*, *Associativity*, *Pattern Summarization*, *Getters and Setters*, and *Constructor Hiding*. From the 50 measured values we have always only taken the last 30 into account in order to exclude external influences as much as possible.

Table 7.1 shows the average processing time in milliseconds of each trace in each phase without any reduction. For measuring the reduction phase in this case we implement a filter that passes the trace unchanged to the next receiver. These values are used as initial values for the calculation of the performance. In this and the following tables, the traces are ordered by their size from small to large.

Table 7.1. The processing time (in ms) of the traces in the different phases without reduction

Traces	Reduction	Transfer	Model Update	Total	#calls
pubflow-1	1.29 ms	45.06 ms	4.99 ms	51.34 ms	1341
pubflow-2	1.03 ms	0.89 ms	13.75 ms	15.67 ms	3124
pubflow-3	1.53 ms	4.00 ms	42.72 ms	48.25 ms	8912
checkstyle	0.73 ms	1.16 ms	123.70 ms	125.60 ms	31237
pacman	0.79 ms	1.51 ms	392.00 ms	394.30 ms	139582
jhotdraw	0.75 ms	1.71 ms	722.40 ms	724.80 ms	161088

7.3.1 Tree Summarization

Table 7.2 shows the processing time in milliseconds of each trace in each phase with usage of *Tree Summarization*, and the number of method calls left after reduction. *Tree Summarization* is the technique with the highest reduction rate, thus the reduced traces are very small. The number of method calls after the reduction ranges between 108 and 706, wherein the reduced size does not depend on the starting size. The total processing time is highly dependent on the processing time of the reduction phase, which ranges between 34.7 milliseconds and 14.37 seconds.

Table 7.3 shows the performance of *Tree Summarization* for the six example traces in the different phases. The performance values for the reduction phase are close to zero, on the other hand the performance values for the model update phase are very high. For the two largest traces the model update phase is a hundred times faster than without reduction. Unfortunately, the total performance values show that this technique in general leads to a slowdown of the processing time of a trace.

Table 7.2. The processing time (in ms) of the traces in the different phases with *Tree Summarization*

Traces	Reduction	Transfer	Model Update	Total	Reduced Size
pubflow-1	34.70 ms	4.06 ms	1.12 ms	39.88 ms	204
pubflow-2	138.20 ms	1.52 ms	5.56 ms	145.30 ms	430
pubflow-3	665.90 ms	36.90 ms	2.13 ms	704.90 ms	600
checkstyle	1195.00 ms	1.42 ms	5.26 ms	1202.00 ms	488
pacman	875.10 ms	2.51 ms	3.96 ms	881.60 ms	108
jhotdraw	14370.00 ms	1.99 ms	7.14 ms	14370.00 ms	706

Table 7.3. The performance of *Tree Summarization* in the different phases

Traces	Reduction	Transfer	Model Update	Total	Reduction Rate
pubflow-1	0.04	11.10	4.45	1.29	92.4%
pubflow-2	0.01	0.58	2.47	0.11	93.1%
pubflow-3	0.00	0.11	20.09	0.07	96.6%
checkstyle	0.00	0.82	23.51	0.10	99.2%
pacman	0.00	0.60	99.09	0.45	99.9%
jhotdraw	0.00	0.86	101.11	0.05	99.8%

7.3.2 Iteration

Table 7.4 shows the processing time in milliseconds of each trace in each phase with usage of *Iteration*, and the number of method calls left after reduction. *Iteration* is the technique with the lowest reduction rate. The number of method calls after the reduction ranges between 1319 and 139119, wherein the reduced size does depend not only on the structure of the trace but also on the starting size. Therefore, the traces are still sorted according to size. The processing time in the reduction phase and in the model update phase increases with the size of the trace. The total processing time is highly dependent on the model update phase. An exception is the *pubflow-1* trace, where the transfer phase takes unusually long time.

Table 7.5 shows the performance of *Iteration* for the six example traces in the different phases. The reduction rates of the traces with this technique are between 0.6% and 24.5%. Thus, it depends strongly on the structure of the trace. The higher the reduction rate, the better the total performance. As of a reduction rate of 4.5%, this technique speeds up the processing time of the traces of our test set.

Table 7.4. The processing time (in ms) of the traces in the different phases with *Iteration*

Traces	Reduction	Transfer	Model Update	Total	Reduced Size
pubflow-1	1.57 ms	46.14 ms	5.13 ms	52.84 ms	1319
pubflow-2	1.16 ms	0.85 ms	10.73 ms	12.75 ms	2983
pubflow-3	2.40 ms	4.40 ms	44.76 ms	51.55 ms	8857
checkstyle	2.56 ms	1.67 ms	93.97 ms	98.20 ms	23596
pacman	7.45 ms	1.53 ms	404.40 ms	413.30 ms	138306
jhotdraw	10.81 ms	1.47 ms	544.60 ms	556.80 ms	139119

Table 7.5. The performance of *Iteration* in the different phases

Traces	Reduction	Transfer	Model Update	Total	Reduction Rate
pubflow-1	0.82	0.98	0.97	0.97	1.6%
pubflow-2	0.89	1.04	1.28	1.23	4.5%
pubflow-3	0.64	0.91	0.95	0.94	0.6%
checkstyle	0.29	0.70	1.32	1.28	24.5%
pacman	0.11	0.99	0.97	0.95	0.9%
jhotdraw	0.07	1.16	1.33	1.30	13.6%

7.3.3 Associativity

Table 7.6 shows the processing time in milliseconds of each trace in each phase with usage of *Associativity*, and the number of method calls left after reduction. The reduction phase takes 1.89 to 52.56 milliseconds, the model update phase 4.46 to 486.8. Hence, the total processing time depends strongly on the update model phase.

Table 7.6. The processing time (in ms) of the traces in the different phases with *Associativity*

Traces	Reduction	Transfer	Model Update	Total	Reduced Size
pubflow-1	2.13 ms	37.65 ms	4.46 ms	44.24 ms	1112
pubflow-2	1.89 ms	0.85 ms	14.16 ms	16.89 ms	1875
pubflow-3	10.72 ms	4.85 ms	35.40 ms	50.97 ms	6915
checkstyle	13.43 ms	1.53 ms	74.54 ms	89.49 ms	18926
pacman	37.95 ms	1.37 ms	224.50 ms	263.90 ms	83553
jhotdraw	52.56 ms	1.17 ms	486.80 ms	540.60 ms	123597

Table 7.7 shows the performance of *Associativity* for the six example traces in the different phases. The reduction rates of the traces with this technique are between 17.1% and 40.1%. Four of the six traces are accelerated by this technique.

Table 7.7. The performance of *Associativity* in the different phases

Traces	Reduction	Transfer	Model Update	Total	Reduction Rate
pubflow-1	0.61	1.20	1.12	1.16	17.1%
pubflow-2	0.55	1.05	0.97	0.93	40.0%
pubflow-3	0.14	0.82	1.21	0.95	22.4%
checkstyle	0.05	0.76	1.66	1.40	39.4%
pacman	0.02	1.10	1.75	1.49	40.1%
jhotdraw	0.01	1.47	1.48	1.34	23.3%

7.3.4 Pattern Summarization

Table 7.8 shows the processing time in milliseconds of each trace in each phase with usage of *Pattern Summarization*, and the number of method calls left after reduction. This technique has two parameters which affect the reduction, and speed. For this experiment we choose a distance of 300 and a nesting level of 1. This leads to 3 to 110 milliseconds processing time in the reduction phase, and 3.5 to 339 milliseconds in the model update phase. The total processing time is between 20 and 450 milliseconds, thus it depends more on the update model phase, but not that much like in the techniques mentioned before.

Table 7.8. The processing time (in ms) of the traces in the different phases with *Pattern Summarization*

Traces	Reduction	Transfer	Model Update	Total	Reduced Size
pubflow-1	3.09 ms	13.26 ms	3.58 ms	19.93 ms	538
pubflow-2	7.06 ms	0.95 ms	12.19 ms	20.21 ms	2517
pubflow-3	9.82 ms	6.17 ms	10.61 ms	26.60 ms	1544
checkstyle	18.00 ms	1.26 ms	50.53 ms	69.79 ms	12001
pacman	29.27 ms	1.25 ms	154.70 ms	185.20 ms	53353
jhotdraw	110.20 ms	1.23 ms	339.20 ms	450.60 ms	83479

Table 7.9 shows the performance of *Pattern Summarization* for the six example traces in the different phases. The reduction rates of the traces with this technique are between 19.4% and 82.7%. Five of the six traces are accelerated by this technique.

7.3.5 Getters and Setters

Table 7.10 shows the processing time in milliseconds of each trace in each phase with usage of *Getters and Setters*, and the number of method calls left after reduction. The reduction phase takes 3.56 to 350.40 milliseconds, the model update phase 3.81 to 348.40. Hence, the total processing time depends approximately equal on both phases.

Table 7.9. The performance of *Pattern Summarization* in the different phases

Traces	Reduction	Transfer	Model Update	Total	Reduction Rate
pubflow-1	0.42	3.40	1.39	2.58	59.9%
pubflow-2	0.15	0.94	1.13	0.78	19.4%
pubflow-3	0.16	0.65	4.03	1.81	82.7%
checkstyle	0.04	0.92	2.45	1.80	61.6%
pacman	0.03	1.20	2.53	2.13	61.8%
jhotdraw	0.01	1.39	2.13	1.61	48.2%

Table 7.10. The processing time (in ms) of the traces in the different phases with *Getters and Setters*

Traces	Reduction	Transfer	Model Update	Total	Reduced Size
pubflow-1	3.56 ms	23.73 ms	3.81 ms	31.10 ms	965
pubflow-2	9.86 ms	1.44 ms	6.89 ms	18.18 ms	1825
pubflow-3	25.90 ms	5.53 ms	23.12 ms	54.55 ms	6515
checkstyle	59.98 ms	1.18 ms	67.12 ms	128.30 ms	19974
pacman	223.50 ms	1.19 ms	124.10 ms	348.70 ms	51192
jhotdraw	350.40 ms	1.19 ms	348.40 ms	699.90 ms	90184

Table 7.11 shows the performance of *Getters and Setters* for the six example traces in the different phases. The reduction rates of the traces with this technique are between 26.9% and 63.3%. Three of the six traces are accelerated by this technique. The most total performance values are close to 1, which means the processing time with and without reduction is the same.

Table 7.11. The performance of *Getters and Setters* in the different phases

Traces	Reduction	Transfer	Model Update	Total	Reduction Rate
pubflow-1	0.36	1.90	1.31	1.65	28.0%
pubflow-2	0.10	0.62	2.00	0.86	41.6%
pubflow-3	0.06	0.72	1.85	0.88	26.9%
checkstyle	0.01	0.98	1.84	0.98	36.1%
pacman	0.00	1.27	3.16	1.13	63.3%
jhotdraw	0.00	1.43	2.07	1.04	44.0%

7.3.6 Constructor Hiding

Table 7.12 shows the processing time in milliseconds of each trace in each phase with usage of *Constructor Hiding*, and the number of method calls left after reduction. The reduction phase takes 1.42 to 7.13 milliseconds, the model update phase 4.72 to 575.30. Hence, the total processing time depends strongly on the update model phase.

Table 7.12. The processing time (in ms) of the traces in the different phases with *Constructor Hiding*

Traces	Reduction	Transfer	Model Update	Total	Reduced Size
pubflow-1	1.42 ms	42.19 ms	4.72 ms	48.33 ms	1228
pubflow-2	4.66 ms	0.99 ms	10.95 ms	16.60 ms	2784
pubflow-3	2.12 ms	4.39 ms	39.60 ms	46.11 ms	8691
checkstyle	2.10 ms	1.54 ms	99.55 ms	103.20 ms	28844
pacman	7.03 ms	1.58 ms	379.80 ms	388.40 ms	138487
jhotdraw	7.13 ms	1.47 ms	575.30 ms	583.90 ms	155403

Table 7.13 shows the performance of *Constructor Hiding* for the six example traces in the different phases. The reduction rates of the traces with this technique are between 0.8% and 10.9%. The most total performance values are close to 1, which means the processing time with and without reduction is the same.

Table 7.13. The performance of *Constructor Hiding* in the different phases

Traces	Reduction	Transfer	Model Update	Total	Reduction Rate
pubflow-1	0.91	1.07	1.06	1.06	8.4%
pubflow-2	0.22	0.90	1.26	0.94	10.9%
pubflow-3	0.72	0.91	1.08	1.05	2.5%
checkstyle	0.35	0.76	1.24	1.22	7.7%
pacman	0.11	0.95	1.03	1.02	0.8%
jhotdraw	0.10	1.16	1.26	1.24	3.5%

7.4 Discussion of the Results

Figure 7.2 shows the average performance values of the reduction phase for all evaluated techniques and the complete test set. This bar chart shows that *Iteration* and *Constructor Hiding* require the least processing time and *Tree Summarization* by far the most.

Figure 7.3 shows the average performance values of the model update phase for all evaluated techniques and the complete test set. While the performance values of *Tree*

Summarization and for example, *Iteration*, are significantly different, the values of the y-axis scale logarithmically. *Tree Summarization* is by far the best technique in this phase, but *Pattern Summarization* and *Getters and Setters* also have values significantly greater than 1.

Figure 7.4 shows the average performance values of the combined phases for all evaluated techniques and the complete test set. *Tree Summarization* leads in the most cases to a significant slowdown. Hence, with this experimental setup we can not advise this technique. *Pattern Summarization* on the other hand leads in the most cases to a significant acceleration, and thus is the best of our evaluated techniques. *Iteration*, *Associativity*, *Getters and Setters*, and *Constructor Hiding* are close to 1. Hence, in this setup they are not useful.

Since the reduction phase can be parallelized, the performance values of the model update phase demonstrate the potential of the reduction techniques for other experimental setups.

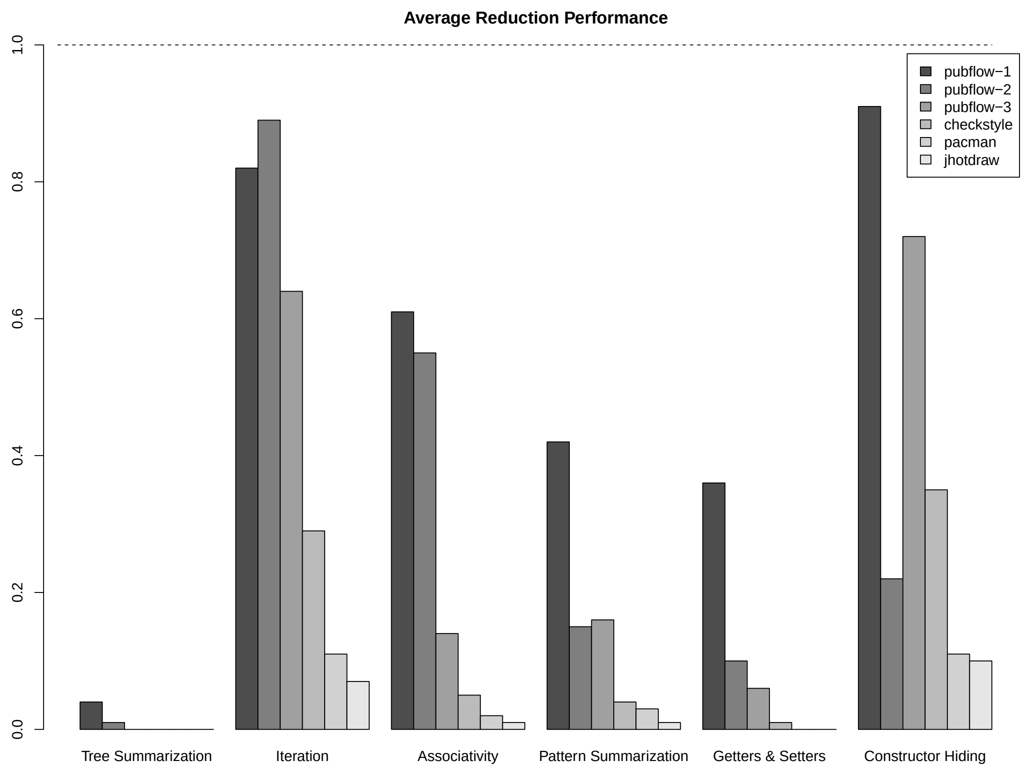


Figure 7.2. Average performance rate for the reduction phase for all techniques and traces

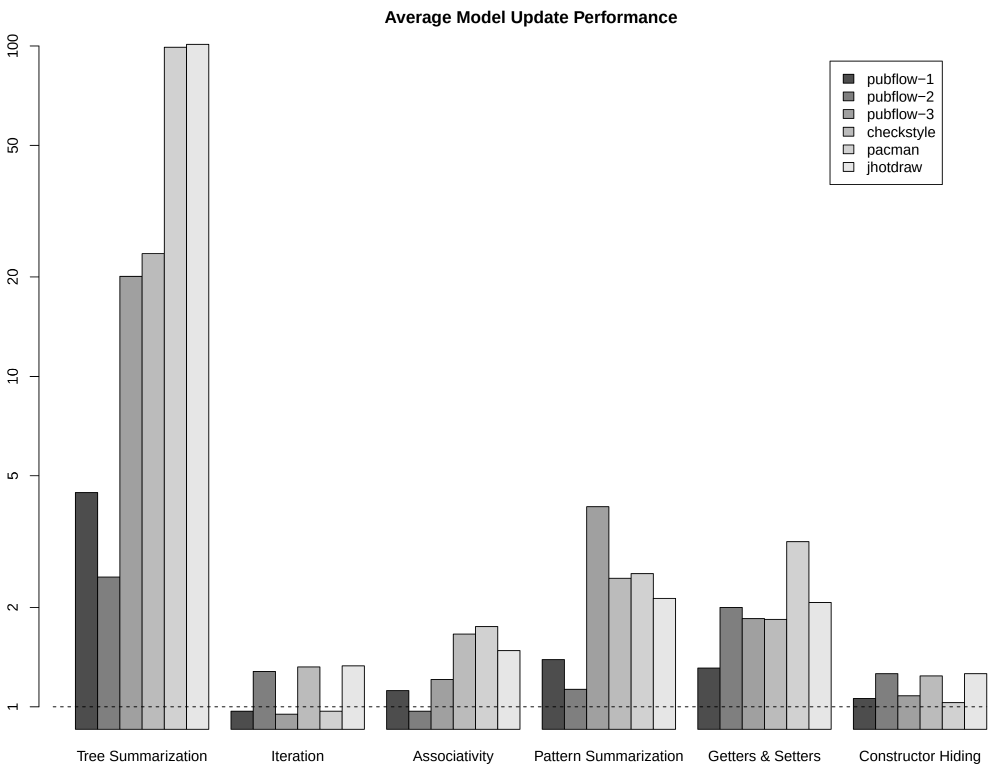


Figure 7.3. Average performance rate for the model update phase for all techniques and traces

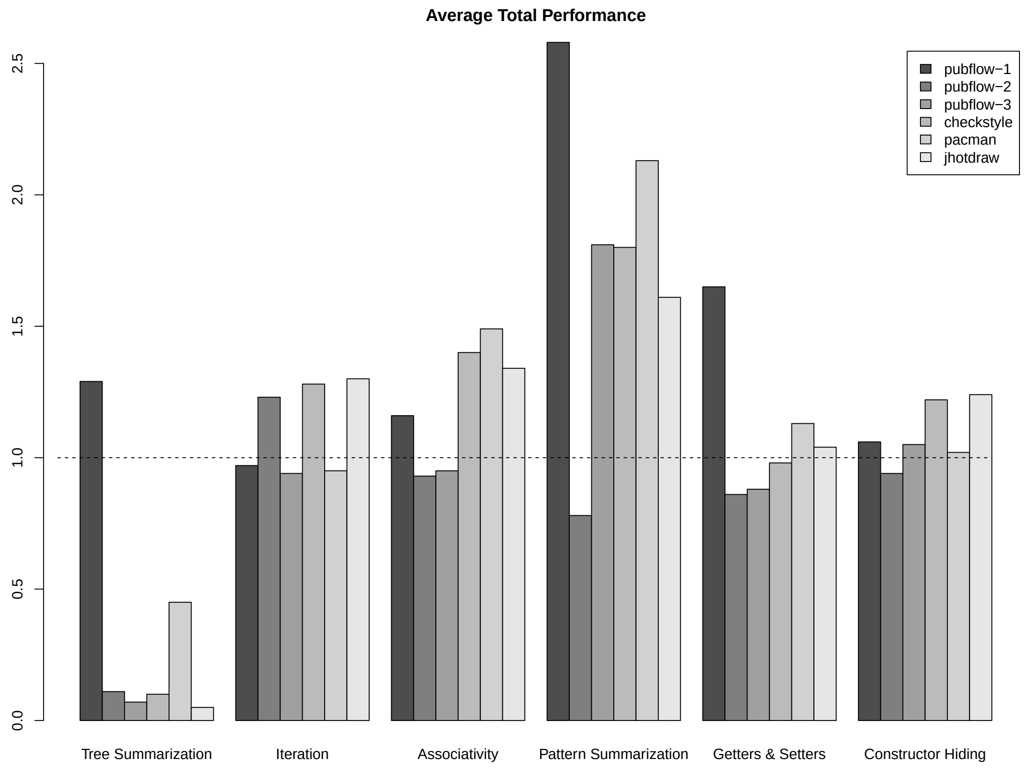


Figure 7.4. Average total performance rate for all techniques and traces

7.5 Threats to Validity

We conducted the evaluation only on one desktop computer, and running ExplorViz with only one worker node. Furthermore, the test set contains only 6 traces with a maximum size of 161088 method calls. Unfortunately, the traces with more method calls provided by Cornelissen does not follow the prescribed pattern.

Furthermore, the measurements for *Pattern Summarization* depend on two attributes, the *distance* in which repetitions are searched, and the *nesting level* that determines how often the trace passes the algorithm. We evaluate this technique only with a distance of 300 and a nesting level of 1. A comparison with the values of the theoretical evaluation shows how much these attributes affect the reduction. With a distance of 300 and a nesting level of 1 the *pubflow-2* trace is reduced by 19%, with a distance of 310 the reduction increases to 34%.

Chapter 8

Related Work

Cornelissen

Cornelissen et al. [2008] inspired this thesis with their list of trace reduction techniques. They implement a few of the techniques in Perl and evaluate them with a selection of traces which differ in size and structure. Their focus lies on the assessment criteria. Therefore, the easiest to implement techniques were chosen. Most of these techniques preserve too less information for the use in ExplorViz.

Mohror

Mohror and Karavanic [2009] describe in their paper *"Evaluating Similarity-based Trace Reduction Techniques for Scalable Performance Analysis"* different similarity methods to compare traces. These methods then are used in a trace reduction technique like pattern summarization.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The use of dynamic analysis in the program comprehension process leads to a large amount of trace data. For storage or live trace visualization these data must be reduced. Addressing this issue numerous trace reduction techniques were developed in the last years. In this thesis we present several trace reduction techniques, and evaluate them based on their reduction rate, their information preservation, their parallelizability, and their performance. With our experimental setup *Pattern Summarization* achieved the best results. With a parallel approach *Tree Summarization* could also be an interesting technique.

9.2 Future Work

The future work should focus on a wider evaluation. All implemented techniques should be tested on a greater test set, and with parallel worker nodes. Furthermore, *Pattern Summarization*, which was the best technique in our evaluation, should be tested with different values for distance and nesting level.

For traces with several millions of method calls a parallelization through building subtraces like described in Section 4.3 is an interesting task. In this case, the reduction filter must be adjusted.

Another task is to analyze the 3 big traces from Cornelissen, and afterwards modify the *CornelissenTraceToExplorVizTransformFilter* to read them correctly.

Bibliography

[Ball 1999] T. Ball. The concept of dynamic analysis. In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pages 216–234. URL: http://dl.acm.org/citation.cfm?id=318773.318944. (Cited on pages 19, 20)

[Brauer and Hasselbring 2013] P. C. Brauer and W. Hasselbring. Pubflow: provenance-aware workflows for research data publication. In: *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP '13)*. 2013. URL: http://www.pubflow.de. (Cited on page 9)

[Chan et al. 2003] A. Chan, R. Holmes, G. Murphy, and A. Ying. Scaling an object-oriented system execution visualizer through sampling. In: *Program Comprehension, 2003. 11th IEEE International Workshop on*. 2003, pages 237–244. (Cited on page 30)

[*Checkstyle*]. Checkstyle. last accessed: 2014-05-11. URL: http://checkstyle.sourceforge.net/. (Cited on page 9)

[Cornelissen et al. 2007] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In: *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*. 2007, pages 213–222. (Cited on pages 24 and 29)

[Cornelissen et al. 2008] B. Cornelissen, L. Moonen, and A. Zaidman. An assessment methodology for trace reduction techniques. In: *Proceedings of the 24th Conference on Software Maintenance*. Sept. 2008, pages 107 –116. (Cited on pages 2, 11, 18, 27, 29, 31, 48, 57, and 69)

[De Pauw et al. 1998] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In: *Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*. COOTS'98. USENIX Association, 1998, pages 16–16. URL: http://dl.acm.org/citation.cfm?id=1268009.1268025. (Cited on pages 11–15)

[Fittkau et al. 2013a] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: the explorviz approach. In: *1st IEEE International Working Conference on Software Visualization (VISSOFT 2013)*. 2013. URL: http://explorviz.net/. (Cited on page 7)

[Fittkau et al. 2013b] F. Fittkau, J. Waller, P. C. Brauer, and W. Hasselbring. Scalable and live trace processing with kieker utilizing cloud computing. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*. CEUR Workshop Proceedings, 2013, pages 89–98. URL: http://explorviz.net/. (Cited on page 9)

[Gargiulo and Mancoridis 2001] J. Gargiulo and S. Mancoridis. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. 2001. (Cited on pages 17 and 26)

[Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In: *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pages 57–76. (Cited on page 5)

[Hamou-Lhadj and Lethbridge 2006] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. 2006, pages 181–190. (Cited on page 29)

[Hamou-Lhadj and Lethbridge 2002] A. Hamou-Lhadj and T. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In: *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. 2002, pages 159–168. (Cited on pages 15–18)

[Hamou-Lhadj et al. 2005] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In: *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*. 2005, pages 112–121. (Cited on pages 22, 23)

[Hamou-Lhadj and Lethbridge 2004] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In: *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '04. Markham, Ontario, Canada: IBM Press, 2004, pages 42–55. URL: http://dl.acm.org/citation.cfm?id=1034914.1034918. (Cited on page 31)

[Jacobson 1992] I. Jacobson. Object-oriented Software Engineering. New York, NY, USA: ACM, 1992. (Cited on page 11)

[*JHotDraw*]. JHotDraw. last accessed: 2014-05-11. URL: http://www.jhotdraw.org/. (Cited on page 9)

[*JPacman-Framework*]. JPacman-Framework. last accessed: 2014-05-11. URL: https://github.com/SERG-Delft/jpacman-framework. (Cited on page 9)

[Kleinberg 1999] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM* 46.5 (Sept. 1999), pages 604–632. (Cited on page 23)

[Kuhn and Greevy 2006] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In: *Software Maintenance, 2006. ICSM '06. 22nd IEEE International Conference on*. 2006, pages 320–329. (Cited on pages 18 and 21)

[Mancoridis et al. 1998] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In: *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*. 1998, pages 45–52. (Cited on pages 18–20)

[Mancoridis et al. 1999] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In: *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*. 1999, pages 50–59. (Cited on page 18)

[Mohror and Karavanic 2009] K. Mohror and K. Karavanic. Evaluating similarity-based trace reduction techniques for scalable performance analysis. In: *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. 2009, pages 1–12. (Cited on page 69)

[*MyBatis JPetStore*]. MyBatis JPetStore. last accessed: 2014-05-11. URL: `http://mybatis.github.io/spring/sample.html`. (Cited on pages 7, 8)

[Reiss and Renieris 2001] S. Reiss and M. Renieris. Encoding program executions. In: *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*. 2001, pages 221–230. (Cited on page 17)

[Rohr et al. 2008] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stoever, S. Giesecke, and W. Hasselbring. Kieker: continuous monitoring and on demand visualization of Java software behavior. In: *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE'08)*. ACTA Press, Feb. 2008, pages 80–85. (Cited on page 9)

[Safyallah and Sartipi 2006] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In: *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on Program Comprehension (ICPC)*. 2006, pages 84–88. (Cited on pages 15, 16)

[Salah and Mancoridis 2004] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. 2004, pages 72–81. (Cited on page 5)

[Van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: design and application of the Kieker framework. Technical report TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009. (Cited on page 9)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. 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 2012)*. ACM, Apr. 2012, pages 247–248. (Cited on page 8)

[Vasconcelos et al. 2005] A. Vasconcelos, R. Cepêda, and C. Werner. An approach to program comprehension through reverse engineering of complementary software views. In: *Proc. Workshop on Program Comprehension through Dynamic Analysis (PCODA)*. 2005, pages 58–62. (Cited on page 31)

[Wong et al. 1999] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In: *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. 1999 IEEE Symposium on*. 1999, pages 194–203. (Cited on page 31)

[Zaidman and Demeyer 2004] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In: *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*. 2004, pages 329–338. (Cited on page 20)

[Zaidman et al. 2005] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In: *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*. 2005, pages 134–142. (Cited on pages 23, 24)