CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abteilung Software Engineering

Diploma Thesis

Automatic Failure Diagnosis based on Timing Behavior Anomaly Correlation in Distributed Java Web Applications

Nina Sophie Marwede

August 14, 2008

First examiner	Prof. Dr. Wilhelm Hasselbring
Second examiner	MIT Matthias Rohr
Advisor	Dipl.-Inform. André van Hoorn
Advisor	MIT Matthias Rohr

Abstract

One approach to handling software failures, especially in large distributed systems, is the monitoring of components for quick reaction and recovery to reduce downtimes and maintenance costs. In previous work, our group has developed tools to monitor and evaluate the timing behavior of Java software systems to detect anomalies, which may be indicators of erroneous behavior.

This diploma thesis enhances the approach to isolate the root cause for failures in distributed systems. Existing technologies are connected and extended to aggregate anomalies to a failure diagnosis. An anomaly correlator is developed that combines timing behavior anomalies using derived dependency graphs. It is then applied to a sample application under the influence of workload generation and fault injection. The results are visualized as colored graphs that can be exported to various image formats.

Contents

1 Introduction **1**

1.1 Example . . . 3

1.2 Goals . . . 3

1.2.1 Anomaly Correlator . . . 3

1.2.2 Case Study . . . 5

1.3 Formalia . . . 5

1.4 Document Structure . . . 6

2 Foundations **7**

2.1 Automatic Failure Diagnosis . . . 7

2.2 Timing Behavior Anomalies . . . 9

2.3 Event Correlation . . . 11

2.4 Distributed Java Web Applications . . . 12

2.5 Related Work . . . 14

3 Approach **15**

3.1 Solution Idea . . . 17

3.1.1 Model building . . . 18

3.1.2 Aggregation . . . 18

3.1.3 Correlation . . . 19

3.1.4 Visualization . . . 20

3.2 Requirements . . . 20

3.3 Data Structures . . . 22

3.3.1 Input . . . 22

3.3.2 Output . . . 22

3.3.3 Structure Classes . . . 24

3.4 Dependency Graph Creation . . . 26

3.5 Analysis . . . 28

3.5.1 Preconditions . . . 29

3.5.2 Strategy . . . 30

3.5.3 Realization . . . 31

3.6 Results . . . 41

3.6.1 Visualization . . . 41

3.6.2 Automation . . . 45

4 Evaluation **47**
4.1 Goals and Metrics . . . 48
4.2 Experiment Setup . . . 49
4.2.1 Application under Analysis . . . 51
4.2.2 Monitoring . . . 52
4.2.3 Workload Generation . . . 52
4.3 Fault Injection . . . 54
4.3.1 Programming Faults . . . 55
4.3.2 Database Connection Slowdown . . . 58
4.3.3 Hard Disk Misconfiguration . . . 60
4.3.4 High System Load . . . 61
4.3.5 CPU Throttling . . . 61
4.4 Experiments . . . 62
4.4.1 Activities . . . 62
4.4.2 Results . . . 63
4.5 Analysis . . . 67
4.5.1 Experiment Selection . . . 67
4.5.2 Examination Activities . . . 68
4.5.3 Default Parameter Selection . . . 69
4.5.4 Three Algorithms . . . 69
4.5.5 In-Out-Relation . . . 71
4.5.6 Edge Weight Methods . . . 71
4.5.7 Mean Calculation Methods . . . 73
4.5.8 Neighborhood Mean Distance Exponents . . . 73
4.5.9 Three Algorithms with New Parameters . . . 77
4.5.10 Cross-Check Appliance to Other Fault Scenarios . . . 80
4.6 Summary . . . 81

5 Conclusions **83**
5.1 Achievements . . . 84
5.2 Limitations . . . 85
5.3 Future Work . . . 89

A Experiment Setup Details 91
A.1 Experiment Activities 91
A.2 Instrumentation of the JPetStore 91
A.3 Preparations for Fault Injection 94

B Correlator Plug-in Usage 95
B.1 Package Content 95
B.2 Tpan Integration 97
B.2.1 Machine Interface 97
B.2.2 Human Interface 98
B.3 Correlator Configuration 98
B.3.1 General 98
B.3.2 Presentation 98
B.3.3 Algorithm 99
B.4 Experiment Instructions 100
B.5 New Algorithms 100

Bibliography 101

Acknowledgement 105

Declaration 107

List of Figures

1.1 Relation between cost and dependability 2
1.2 Anomalies plus dependencies lead to an estimation about failure cause . 4
1.3 Possible distribution of the JPetStore 5

2.1 Error propagation 8
2.2 Kieker architecture 9
2.3 Load-time weaving, a method for Aspect Oriented Programming 11
2.4 Three-tier architecture 13
2.5 Structure of Remote Message Invocation 13

3.1 Overview of the analysis 18
3.2 Example of the first results of the analysis 19
3.3 Data structures produced by Tpan 22
3.4 Structural hierarchy tree 23
3.5 Dependency net 23
3.6 Structural classes of the Correlator 25
3.7 Construction of the dependency graph 27
3.8 Example of anomaly propagation 28
3.9 Example of a histogram of anomaly scores 32
3.10 Overview of the algorithm variants 33
3.11 Construction of the list of distances and weights 36
3.12 Anomaly correlation on operation level 38
3.13 The inhibition and the logistic sigmoid function 40
3.14 Examples of the increase and decrease functions in advanced correlation . 40
3.15 Example of the textual output of the Correlator's results 42
3.16 Class diagram focused on the presentation classes 43
3.17 Example of the graphical result of the Correlator 44
3.18 Example of an experiment batch control file 46

4.1 Conceptual overview of the experiment setup 50
4.2 iBATIS JPetStore 5 demo application 51
4.3 Deployment of the JPetStore components 52
4.4 Database schema for the monitoring of executions 52
4.5 Apache JMeter 2.3 with Markov4JMeter 1.0 53
4.6 Dependency graph of the JPetStore (Part 1) 57
4.7 Dependency graph of the JPetStore (Part 2) 59

4.8 Manipulation of the hard disk transfer mode with hdparm 60
4.9 Example of a Sysstat plot during an experiment run 64
4.10 Example of a response time plot during an experiment run 64
4.11 Example of the result of a simple anomaly correlation 65
4.12 Sysstat plot showing the impact of system load 66
4.13 Charts of the clearness ratings for three algorithms 70
4.14 Charts of the clearness ratings for a set of in-out-relations 72
4.15 Charts of the clearness ratings for two edge weight methods 72
4.16 Charts of the clearness ratings for five mean methods (Part 1) 75
4.17 Charts of the clearness ratings for five mean methods (Part 2) 76
4.18 Charts of the clearness ratings for a set of distance exponents 76
4.19 Comparison of the algorithms with optimized parameters 78
4.20 Comparison of the graphs for the trivial and optimized algorithm 79
4.21 Comparison of the algorithms with other scenarios 80

5.1 Result graph of an analysis using the trivial algorithm 86
5.2 Result graph of an analysis using the advanced algorithm 87

A.1 Activity diagram of the experiments 92

List of Tables

4.1 Summary of programming faults and effects 55
4.2 Source code manipulations applied to the JPetStore 56
4.3 Source code changes to simulate a database slowdown. 59
4.4 Summary of the fault injection scenarios 63
4.5 Summary of the fault injection results 66
4.6 Results of the experimental comparison of three algorithms 70
4.7 Results of the experimental comparison of the in-out-relation 71
4.8 Results of the experimental comparison of the edge weight method 72
4.9 Results of the experimental comparison of mean calculation methods . . 74
4.10 Results of the experimental comparison of the neighbor distance exponent 77
4.11 Comparison of starting and optimized parameters 77
4.12 Results of the advanced algorithm with optimized parameters 78
4.13 Results of the cross-check with other scenarios 80
4.14 Overview of the examination results on five scenarios 81

Chapter 1

Introduction

Software systems exceeding a certain level of complexity are never free of faults. Software failures increasingly affect our lives, bringing us operational outages, expensive maintenance as well as annoyed customers. Developers often work under high pressure of time and money, and the value of their work is rather measured in functionality and design than in reliability and security. Although most professionals know by now that these aspects should be considered early in the development phase, this is rarely achieved to complete satisfaction. With the possibility to easily deploy software patches over the Internet, it has unfortunately become usual to release unfinished software and to fix problems not before they arise, if ever.

Lyu [1996, p. 4] brings it to the point:

> *"The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test, and maintain them. When the requirements for and dependencies on computers increase, the possibility of crises from computer failures also increases. The impact of these failures ranges from inconvenience (e.g. malfunctions of home appliances) to economic damage (e.g., interruptions of banking systems) to loss of life (e.g. failures of flight systems or medical software). Needless to say, the reliability of computer systems has become a major concern for our society.*

There are many approaches to prevent faults right from the beginning, to eliminate them during the development, and to work around them at runtime, thus increasing dependability. The modern architect's possibilities to increase software quality range from promising new programming languages over formal specification techniques and computer-aided development models as well as verification and validation processes up to Extreme Programming – but you can never be sure to catch them all, or to make the system behave sufficiently tolerant in every use case. Even a software that conforms perfectly to its specification does not guarantee to perform perfectly, since the specification itself may contain faults. Although it is possible to produce such *fault-free* software, "it is extremely difficult and expensive. [. . .] Therefore, software developing organizations accept, explicit or implicit, that some faults remain in their software" [Sommerville, 2001, p. 403]. Accordingly, as backed by Figure 1.1, since most products must not be very expensive, the people got used to have faults in their everyday applications.

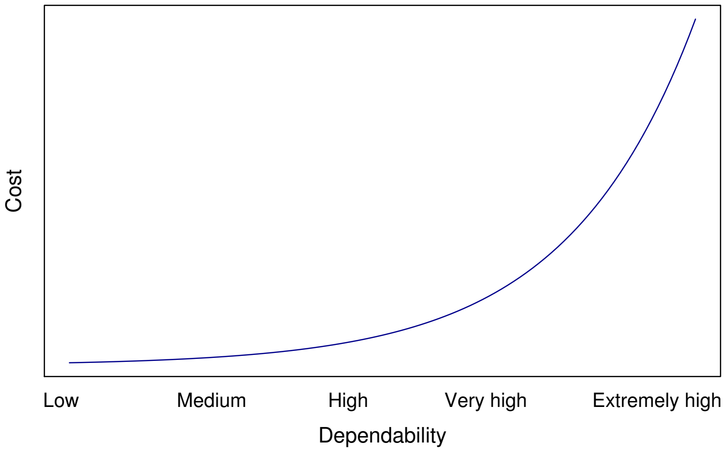


Figure 1.1: Relation between cost and dependability, based on Sommerville [2001, p. 364], who has the opinion that this relation is exponential, and because of that, the reliability of a system could not be 100% proven.

On the other hand, there are large, business-critical software systems, where "restart and recovery" is not the preferred strategy to handle failures. Instead, the problem should be located and solved. Especially in distributed systems, when a failure occurs, the root cause is often hard to identify. Therefore, automated tools and methods for fast diagnosis and efficient fault localization are important to minimize maintenance and incidental costs.

This diploma thesis examines a new method to localize the cause of failures in distributed Java applications. The failure diagnosis is based on the detection of anomalies in the system's timing behavior, and also considers its architecture. Our group's approach is to monitor the system continuously, comparable to operation control centers that are common in industrial environments, e.g. power stations [Giesecke et al., 2006]. Instead of simply comparing the system's behavior with a set of pre-defined "normal" parameters to detect anomalies, our detector relies on statistical information considering different system usages. From this, in combination with dependency graphs that are reconstructed from the monitoring data, the root cause of a failure can be estimated.

For evaluation, we use the Java Web application iBATIS JPetStore[1] which has already been instrumented so that information about runtime behavior is collected and stored into a database [Rohr et al., 2008b]. After a system failure has occurred, our tools can immediately analyze the recorded data and attempt to estimate the causing component.

In the field of dependability, the approach is related to fault avoidance and fault tolerance. More precisely, the localization process is settled between fault detection ("there's a fault somewhere") and fault removal ("got it").

[1] `http://ibatis.apache.org`

1.1 Example

A scenario where the approach can be used may be the following: Imagine a large, business-critical software system. Many users concurrently access it, for example by browsing through the catalog, adding products to their shopping cart, managing their personal data, or doing financial transactions. This software system, deployed on different servers, is crashing after a long runtime, which should be recognized by monitoring tools as significant anomalous behavior.

Typically, if a restart of the system is not sufficient, an administrator would now manually review a large amount of debug and log messages. Fortunately, the monitoring infrastructure has been continuously writing monitoring data into a database. An anomaly detector evaluates the data of a certain time period, and rates every single execution (invocation of a function) whether it was behaving normal, or not.

Now the goal is to build up a model of the software system from this collection, further evaluate the monitoring data, and assign probability estimations to each component, and to the corresponding virtual machines for being the cause of the failure. This provides an alternative to manual failure diagnosis which tends to be time-consuming and error-prone.

1.2 Goals

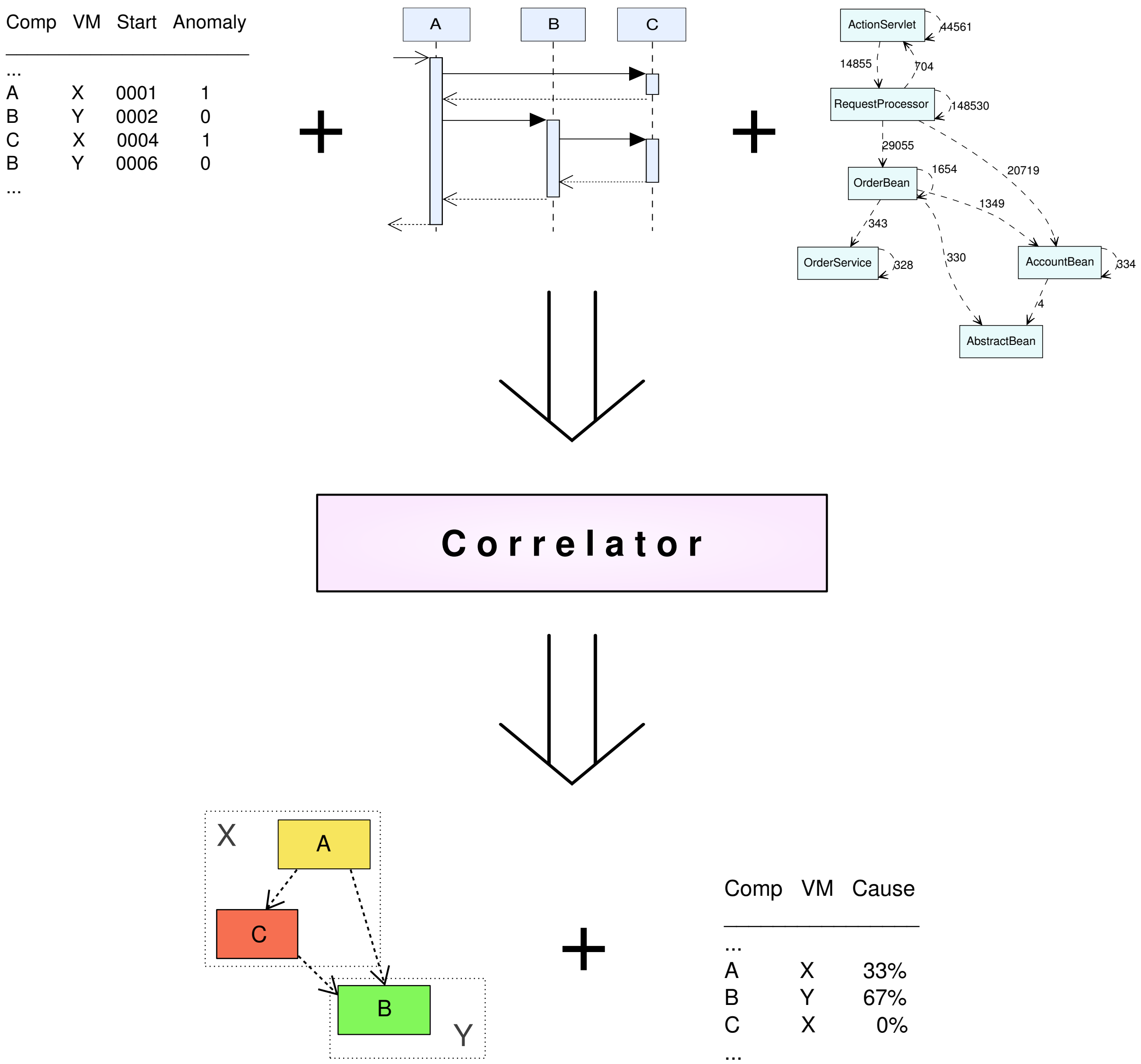
There are two goals for the thesis: (1) An anomaly *correlator* shall be developed, and (2) it has to be evaluated in a *case study*.

1.2.1 Anomaly Correlator

In the thesis, an approach has to be developed and implemented as a prototype that transforms individually detected anomalies into a probability estimation for each component to be the cause of a failure. The collected information about runtime behavior is combined with the component dependencies. It localizes the possible causes of the failure and rates them by their likeliness. Figure 1.2 sketches the context of the “Correlator” to be created.

The main challenge is the choice of the fundamental strategy to generate a valid and clear result. The processing should not be adjusted to a special application or architecture, but rather follow a general concept. Since the algorithms will probably have some parameters, an important secondary goal is to analyze their influence, e.g. whether there exist generally usable values, or whether they have to be found out for each monitored system. Another question is, to what extent the underlying anomaly detector can be trusted.

Another important task is the visualization. Besides a simple textual report, the results have to be presented in graphical form, showing the dependencies and the cause percentages, supported by color.



Comp	VM	Start	Anomaly
...			
A	X	0001	1
B	Y	0002	0
C	X	0004	1
B	Y	0006	0
...			

Comp	VM	Cause
...		
A	X	33%
B	Y	67%
C	X	0%
...		

Figure 1.2: Timing anomalies plus calling dependencies lead to an estimation about failure cause.

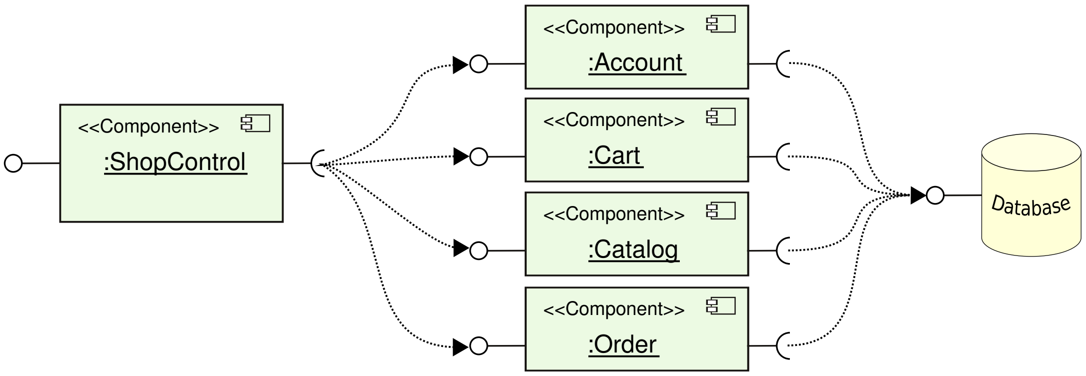


Figure 1.3: Possible distribution of the JPetStore. It has been a monolithic application before, but now its components can run in different deployment contexts.

1.2.2 Case Study

A case study will be performed to expose the concept of the Correlator to reasonably realistic circumstances. The JPetStore has been chosen to be the testing application. It has already been modularized so that it may be deployed over some (virtual) machines, as sketched in Figure 1.3, to match the aspect of distribution. Available tools and methods for fault injection [Rohr et al., 2008b], probabilistic workload generation (*Markov4JMeter*, [van Hoorn et al., 2008]), monitoring (*Kieker*, [Rohr et al., 2008b]), and anomaly detection ([Rohr et al., 2008a]) will be set up. With the distributed deployment, it will be easier to make targeted fault injection without having the components affect each other. For example, high load can be simulated on a function by simply slowing it down.

There are several topics to pay attention to. First, the criteria, where and how to divide the components of the JPetStore, must be decided. Then suitable monitoring points have to be set, and the mechanisms for fault injection have to be decided and applied: Which kind of faults should be injected, and where to put them? One goal of the experiments is whether there are distinct categories of faults that are better detected than others. In order to benchmark the results, a metric has to be developed.

Because architectural migration is a complex task, in technical as well as conceptional respect, this procedure is supported by the advisors of the thesis.

1.3 Formalia

- Important terms when mentioned the first time, or otherwise emphasized words or sentences are in *italic* letters.
- Programming terms such as classes, attributes, packages and the like are in `typewriter` letters.
- Footnotes contain URLs only.

1.4 Document Structure

In Chapter 2, an overview is given of the foundations of our research, and the related work. Chapter 3 documents the functionality and the development of the Correlator prototype. The idea of correlation is illustrated, and the steps of analysis and visualization are explained in detail. A comprehensive description the evaluation experiments and analysis follows in Chapter 4, especially covering fault injection, before the conclusions are presented in Chapter 5.

Chapter 2

Foundations

As motivated in the introduction, failure diagnosis is an important task in the process of maintaining large distributed systems. In the worst case, administrators get flooded by a burst of events. They have to manually sort out the irrelevant ones, analyze the meaning of the rest, and try to connect them to a reason. Automated methods shall help to reduce downtimes, and optimize employees' working time.

Depending on the definition, a *failure* in multi-user Web environments does not always mean the outage of a whole system. It can also mean a partially disabled system or its user interface being unavailable. Maybe the failure affects only a fraction of the users, so that much time is consumed when localizing the causing component, or the incident is not recognized at all by the administration. This motivates efforts to quickly detect possible failures.

Avižienis et al. [2004, p. 13] define these threats to dependability and security:

> "A service **failure**, often abbreviated here to failure, is an event that occurs when the delivered service deviates from correct service. [. . .] Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an **error**. The adjudged or hypothesized cause of an error is called a **fault**."

The following four sections each give an introduction to a technical aspect that the thesis is built upon. Section 2.5 summarizes the most related work.

2.1 Automatic Failure Diagnosis

Automatic failure diagnosis first of all means the collection of information. Usually, this is achieved through *monitoring*, that is, the collection of information from a system during its runtime. In the next step, the raw data has to be processed to finally present a human readable report.

For example, Kiciman and Fox [2005]'s "Pinpoint" detects failures in component-based Internet services by comparing a model of a system's normal behavior with appearing changes which indicate a possible failure. Their approach is limited to the top level in the

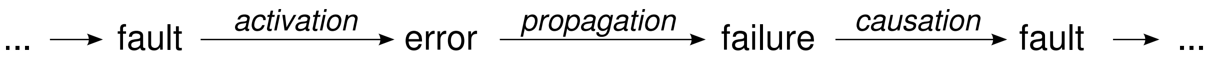


Figure 2.1: Error propagation [Avižienis et al., 2004, p. 21].

software stack (being non-intrusive; they do not look into programming details), neither trying to predict failures *before* they occur, nor trying to explain them.

The authors list three categories of monitors that are used by Internet service operators:

- *Low-level monitors* are watching machine and network components, e.g. HTTP error codes.
- *Application-specific monitors* catch functional aspects like discounts at an e-commerce site.
- *User-activity monitors* compare statistics of pre-defined metrics for user behavior with historical trends.

Attention must be paid to *cascading failures* as a source of *false positives*: Errors can be propagated through a system by its interactions, thus disguising the origin of the failure, as depicted in Figure 2.1. On the other hand, faults do not necessarily result in errors, nor do errors necessarily result in failures, as they may be caught and corrected before a real problem arises, for example through the use of replication techniques.

Monitors can be attached to different structures:

- The *user interface*, where little error messages should be displayed in normal operation, may be parsed for keywords. However, these messages often describe symptoms of a problem instead of its cause – for example, “cannot unlock screen” is displayed instead of “/tmp is readonly”.
- Likewise, the *log files* can be searched, assuming that the developers carefully output all problems there. However, according to Kiciman and Fox [2005] the share of false alarms is relatively high, and hard to filter out.
- Alternatively, special *hooks* can be applied by maintainers, maybe at critical points in the architecture, to report anomalies directly.

Rohr et al. [2008b] present “Kieker” as an approach to continuously monitor Java software systems and to visualize their internal behavior from the monitoring data by creating (amongst others) sequence diagrams, dependency graphs, and execution trace models while being said to have very little overhead. The main components shown in the architecture diagram in Figure 2.2 are the monitoring component *Tpmon* that collects and stores monitoring data, and the analysis component *Tpan* that, by use of its plug-ins, analyzes the data and produces output documents.

One strategy of failure detection is based on the monitoring of timing behavior, which is described in the following section.

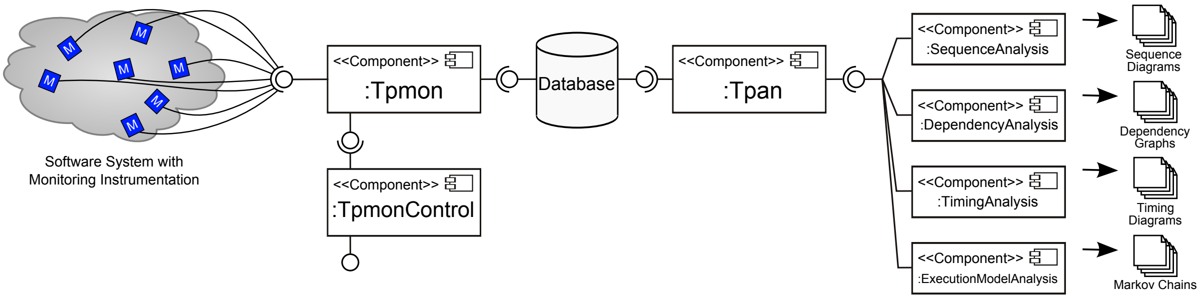


Figure 2.2: Kieker architecture. Tpmon stores monitoring data, and Tpan with its plugins analyze it [Rohr et al., 2008b].

2.2 Timing Behavior Anomalies

Anomalies in timing behavior become important from the view of the users, if they have to *wait* for an answer from their system significantly longer than expected. Even 25 years ago, [Bailey and Soucy, 1983] observed that “response time directly affects the productivity of a user and is one of the main criteria by which an interactive communication system is judged”.

As described by Hoffmann [2006, pg. 240], the typical WWW user gets impatient after only few seconds, and might mistrust the answer. In case of a Web shop, the user might soon abort browsing the catalog and proceed to a competitor’s offerings. From the view of an administrator, such anomalies should be worth some investigation, because they can indicate a spectrum of problems, from avoidable to really annoying.

Timing behavior anomalies can be an indication for a (temporary) overload of the system’s capacity. This might be a peak in the number of concurrent user agents, also known as the *Slashdot Effect* as examined for example by Adler [1999]: The hit rates of Web servers are correlated with the announcement times on high volume news Web sites. Alternatively, the system might be exposed to a distributed denial-of-service attack. On the other hand, these anomalies can be an indication for persistent faults in the system design, architecture, or source code, that could probably be removed easily as soon as they have been localized.

Related work on timing behavior analysis often aims at assessing the performance of a system. For example, Juse et al. [2003] examine the impact of applying Web Service interface technology to a tightly coupled application by measuring, among throughput and latency, the response time. In fact, they notice only a small change of throughput in their experiments, but significantly higher variations in response time under different load situations. This motivates response time to be a sensitive indicator for anomalies.

As noted by Kiciman and Fox [2005, p. 14], anomaly detection can be used for the detection of various types of “bad” behavior – including intrusion detection, code bugs, or violations of invariants – but only if it can be compared to “assumed-good behaviors”, and as long as the system is working correctly most of the time. They admit the possibility of mis-classifying anomalies as “another mode” of the components, in case these fulfill more than one service. That might be a result of their non-intrusive, top-level approach mentioned in Section 2.1.

Rohr et al. [2008a] mention variations of the workload intensity as a major influence on response times: "Anomaly detection with constant thresholds might not achieve optimal results for systems where varying workload intensity occurs that leads to varying response times." They contribute a statistical approach, called *Workload Intensity Sensitive Anomaly Detection* (WISAD), that uses statistical analysis "to detect deviations between current and historical monitoring data". To measure response times, so-called *operations*, i.e. services provided by components, are instrumented with monitoring probes. The anomaly detector relies on a set of training observations, including workload intensity, to decide whether or not each execution of an operation is an anomaly, whereas the thresholds dynamically depend on the *platform workload intensity.*

Overhead through monitoring is an issue when observing response times, because the collected data about timing behavior does not support precise analysis if the monitoring itself has a significant influence on the system's performance.

Other limitations of using time measurements, that are also noticed by our group, are listed by Yilmaz et al. [2008, pg. 4].

- *Imprecision in measurements:* The measurement depends on the resolution of the software/hardware clocks. In distributed systems, we experienced time *synchronization* as a problem.

- *Noise in measurements:* False positives as well as false negatives can arise from undesired activities by the underlying platform. In distributed systems, network traffic is added.

- *Dependency on software/hardware platforms:* Results from measurements on one platform may not be comparable to those on another platform.

Answers to these limitations include the usage of high resolution clocks, the addition of context information, the usage of average values and probability models, and timing abstractions, e.g. percentages instead of actual time values.

Another impact on monitoring is the interference with the source code. One possibility to set monitoring points non-intrusively, thus helping to reduce its influence on performance, is *Aspect Oriented Programming.*

Aspect-Oriented Programming

The concept of Aspect-Oriented Programming (AOP) was first presented by Kiczales et al. [1997] based on the belief that traditional object-oriented programming languages are "ultimately inadequate for many complex systems", because different aspects of a system's behavior "tend to have their own natural form". With AOP, the aspects may be developed separately, and then merged to produce executable code, using a tool they call AspectWeaver.

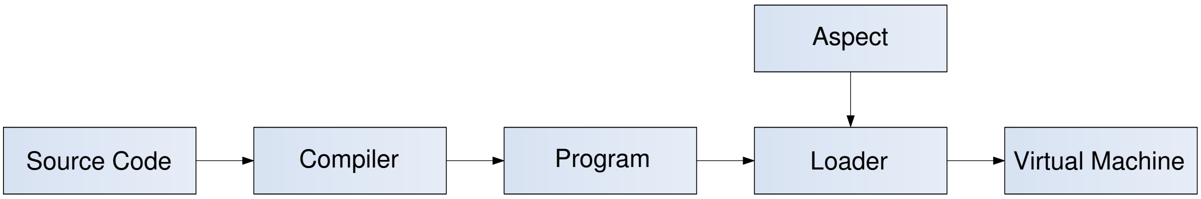


Figure 2.3: Load-time weaving is a method provided by AspectJ for Aspect Oriented Programming [Focke, 2006].

Although AOP has been developed with the thought of letting different languages collaborate, it can also be used to weave additional functionality into an existing program with only minimal changes to the original, using the same language. For example, code can be added at predetermined points, or it can be completely replaced under specific circumstances.

Java Annotations were introduced with Java 5 as a mechanism to enrich Java code with meta data that can be evaluated by external programs, for example to generate additional code. Focke [2006] lists these methods of weaving that are provided by AspectJ[1] (which is an AOP implementation for Java), and that differ in the time when the weaving happens: (1) Compile-time weaving is done by a preprocessor before compilation, (2) Post-compile (binary) weaving is done by a post-processor after compilation, and (3) Load-time weaving (see Figure 2.3) is done not until the program is run in the Virtual Machine. The first two have the disadvantage that the build process of existing applications has to be changed, and their program code has to be compiled and re-deployed when changes occur within the aspects.

In our case, Rohr et al. [2008b] use AOP in the variant of load-time weaving to implement the monitoring of timing behavior. More precisely, their tool *Tpmon* records the call and return times of Java method invocations. With the annotation techniques provided by Java, the goal is achieved to keep the monitoring logic separate from the source code of the software under analysis, and to have little impact on performance.

2.3 Event Correlation

According to Gruschke [1998a], an event correlator first has to reduce the overall number of events to improve the handling, secondly should indicate a problem's cause instead of its symptoms, and thirdly (beyond the scope of this work) it should help to route reports to the right person. As the author states, such mechanisms cannot *prove* the correlation of events, but estimate their connection based on the likelihood of their coincidence, so maybe only a subset of the possible explanations is found. His approach is classified as being a technique for event filtering: A large amount of monitored events is to be reduced to a small amount (if not even a single point) of originating events that evoked all following ones.

[1] `http://www.eclipse.org/aspectj/`

A consequence of the separation of fault detection methods as described in Section 2.1 – the monitoring on different application levels – is that usually all involved software and hardware components, as well as different-level failures, are treated separately. This is an opportunity for event correlation to enhance the structure and relevance of messages by paying attention to the system as a whole.

In a follow-up paper [Gruschke, 1998b], the author proposes an event correlator which processes raw management events that are otherwise shown to the administrator. With the help of a knowledge repository that somehow knows the relations between the events, it shall filter the messages to enhance the fault localization activities. He describes two older concepts of designing the knowledge base: Either a human expert has to teach the correlator through some kind of programming, or a set of specific "cases" is defined which map input events to their probable cause. The author's own approach is based on connecting the correlator to an existing management system, using data already present in it, to construct a dependency graph of the managed objects. The reasons for doing this would be a combination of simplicity and the power of straight-forward consideration through experts looking for "What do they have in common?"

He assumes faults to be *forwarded* as a basic principle – that is, an event is probably not sent by the causing component. For example, when a router in a network fails, then not the device itself, but instead its peers are sending timeout messages. The author even defines dependency by the possibility to provoke a failure in another object (node in the dependency graph).

Agarwal et al. [2004] report their approach for root cause analysis to be utilizing the following combination of concepts: They use dependency information and dynamic runtime performance characteristics to narrow down the root cause. Omitting some categories, like failing hardware components, to traditional management systems, they concentrate on "typical e-business systems" and "non-real-faults", such as a slowdown.

Likewise, Kiciman and Fox [2005, p. 14] report on other approaches to the problem of fault localization to "typically rely on either expert systems with human-generated rules or on the use of dependency models".

2.4 Distributed Java Web Applications

Java Web applications are an implementation of the Client-Server concept. Like many years ago in the mainframe era, applications can be stored and managed at a central site and deployed through the Web as needed. Applications are usually split up into several parts, each handling a particular set of functions.

A three-tier architecture is common for Web-based applications. The *presentation layer* holds the user interface; the *business logic*, or *application layer* contains implements the application logic; and the *data service layer* consists of persistent data. To complete the Web site architecture, there are devices like routers, firewalls, and load balancers.

The distribution aspect is usually a method of scalability to higher loads, allowing the expansion of a service through the use of additional machines, forming a server cluster.

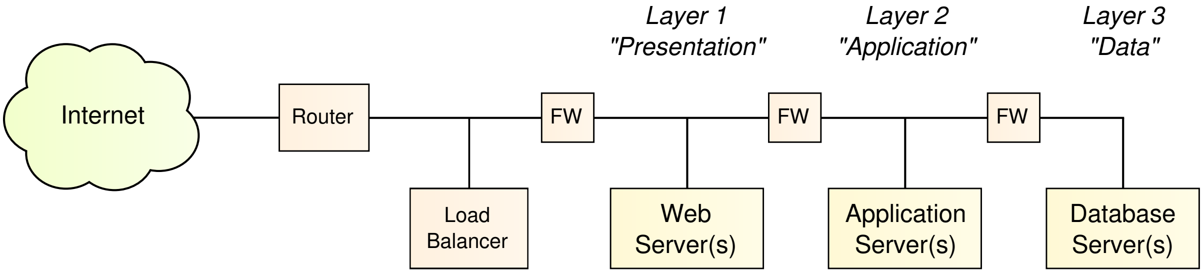


Figure 2.4: Three-tier architecture, based on Menascé and Almeida [2001, p. 159].

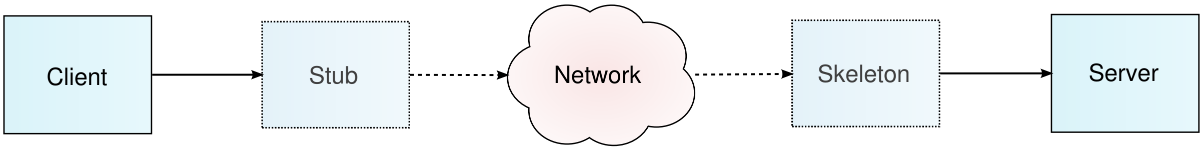


Figure 2.5: Structure of Remote Message Invocation: Method calls from Client to Server are routed over the network through automatically generated proxies called Stub and Skeleton.

In our case, the distribution provides additional possibilities for monitoring, but new sources for faults may arise from building the interfaces, too.

The servers depicted in Figure 2.4 can be separated machines, or services on the same machine, or even several machines for each layer, depending on the requirements. Issues in this context include performance, reliability, security, and maintainability.

Typically, distributed Java Web applications are accessed through HTML Web pages dynamically generated by Java *Servlets*[2]. Servlets are executed within a Servlet container (e.g. *Apache Tomcat*) and provide the presentation of the application logic which may be distributed over a number of application servers.

One possibility of communication between the application components is given by *Remote Method Invocation* as described in the next section.

Remote Message Invocation (RMI)

During the preparation of the current work, we decided to use the Java RMI[3] package from Sun to deal with the communication between the components of our distributed example application, JPetStore. RMI allows objects on one Java Virtual Machine to invoke methods of objects in another Java Virtual Machine. This works almost transparent at client side, with automatically created proxies on both sides (see Figure 2.5). Evolving from a monolithic application, there are only small changes of the source code required. The following paragraphs summarize the steps to apply RMI to an existing program.

An interface extending `java.rmi.Remote` has to be created on the server side for each class to be provided for remote use, containing their respective public methods. The

[2] `http://java.sun.com/products/servlet/`

[3] `http://java.sun.com/javase/technologies/core/basic/rmi/`

methods must be declared to use a `RemoteException` formally, although it will never be used later, because in case of an error, that would be processed on client side.

Each server class must extend `java.rmi.server.UnicastRemoteObject`, that provides (among other things) the ability to leave the server waiting for incoming messages. When instances are created, they have to care about an RMI Registry (e.g. create one), and they have to announce their names in URL format (of the form `//host:port/name`). The server methods are implemented as usual.

The client classes have to obtain references to the server object's interfaces by calling `java.rmi.Naming.lookup()` with the desired server's URL, whose methods can then be called normally, especially concerning synchronous control flow. For the case of an error, e.g. a garbled transmission, `java.rmi.RemoteException` has to be caught.

In order to be transmitted over RMI, objects have to implement the marker interface `java.io.Serializable` – many predefined Java classes already do this. Despite this comfort, there may evolve problems with object handling, e.g. comparisons using "==" may not work, or could have "lost update" effects when working with object copies. Such design flaws cannot be noticed by the compiler, but appear at runtime by throwing an exception in the best case. This also applies to broken transmissions, which have to be explicitly restarted.

However, for the experiments in Chapter 4, another implementation based on Web Services has been used because it works better with our monitoring component.

2.5 Related Work

Similar to our group's studies, Kiciman and Fox [2005]'s "Pinpoint" uses *monitoring* to learn historical reference values, and to detect anomalies in the behavior of an application. For evaluation, the authors use two versions of the Sun Petstore, a predecessor of the JPetStore, one of them distributed across a cluster of machines. Neither do they make use of timing information, nor of the calling dependencies.

The work most related to the approach of this thesis seems to be the one by Agarwal et al. [2004]. Using the dependency graphs is a main element of their They monitor response times, but via non-invasive methods, so their reconstructed graphs may have "imperfections". For fault localization, they use a clustering algorithm combined with a ranking logic to sort the suspicious components. They admit a drawback that sounds very similar to the effect of *propagation*.

Most recent, Yilmaz et al. [2008] present an approach that concentrates on the monitoring of timing behavior. Their "Time Will Tell" is loaded with time spectra from passing and failing runs in order to identify potential causes within the failing runs. Common open source Java applications, instrumented on byte-code level, are used for evaluation. Although the measurements are collected in the form of a call tree, the do not make explicit use of the dependencies to refine their diagnosis.

Chapter 3

Approach

As mentioned in Sections 2.1 and 2.2, Rohr et al. [2008a,b] present an approach to recognizing and evaluating anomalies in the timing behavior of Java applications. Their *Kieker* with the monitoring component *Tpmon* provides functionality to monitor Java software systems at selected points. Tpmon is woven into the application to be monitored using AOP (Aspect Oriented Programming). Raw timing data is collected at runtime and stored into a database or file system.

Kieker's component *Tpan* and its plug-ins read the monitoring data and analyze it. There are plug-ins for visualization, that generate sequence diagrams and dependency graphs, for example. Plug-ins like *PAD* (Plain Anomaly Detection) or *WISAD* (Workload Intensity Sensitive Anomaly Detection) detect anomalies in the timing behavior based on the comparison of the execution's response times with historical or calculated *normal* values. This way, they assign an *anomaly score* to each monitored execution that reflects the degree of its timing to be anomalous – the higher the deviation from what is defined as normal, the higher the score.

However, the output data of Tpan's anomaly analysis algorithms does not support human administrators in performing failure diagnosis very well. Users get raw evaluations of the timing behavior so far, but this might not help much for quickly identifying possible *causes*. There is no aggregation of the large number of anomaly events, that might be distributed over several application parts, and that are not optimized for human readability. Furthermore, there is no correlation: The anomaly events are collected separately, and no connection is drawn between them, thus it would not be recognized if groups of them belong together, and have the same cause. A prevalent example for this is the effect of anomaly *propagation*: A slowdown observed at one component might be a consequence of faults in other components not directly related to the first. Evaluation functionality is needed to *aggregate* the anomaly events, to *correlate* their meanings, to provide the results for further automatic analysis, and to clearly *present* them to human users.

The goal is to combine the anomaly detection with component dependency information that is reconstructed from the monitoring data. Similar to speech recognition or other applications of semantics, somehow abstract input data has to be interpreted, correlated, and given a *meaning* to get a kind of understanding of its content. In terms of the propagation, especially the calling dependencies between the application's components

– in the form of “which method invokes which other methods” – seem to provide useful information for determining a failure’s root cause.

A *correlation algorithm* is created that automatically determines parts of the *system under analysis* (SUA) which are strong indicators to the cause of failure. If the approach is not able to point exactly to *one* part of the application architecture, at least it provides a significant reduction of the search space of possible causes by declaring a large percentage of software elements as *not* being the fault with high probability.

The following requirements for monitoring techniques listed by Kiciman and Fox [2005, p. 2] are relevant for the current approach.

- *High accuracy and coverage:* Without special training, a broad range of failures anywhere in the system should be correctly localized.
- *Few false-alarms:* The rate of false-alarms should be adequately small to not negate the benefit of fault detection.
- *Deployable and maintainable:* No special adjustments to the monitored system should be needed, so that its development or maintenance can easily continue. The evaluation itself should require little configuration effort.

Like with Gruschke [1998a], the results depend on the completeness of the dependency graph, which is reconstructed from the monitoring data. If the monitoring instrumentation is too coarse-grained, i.e. only few monitoring points exist, then the evaluation will be imprecise, in that dependencies are missing, and therefore connections between the affected components cannot be recognized. If it is woven too fine-grained, i.e. most methods are instrumented, the amount of data will increase without being an advantage to the result. To be prepared for all kinds of faults, the *full instrumentation* of the system seems to be necessary: The entry and the exit points of every function or service, which might potentially contain a problem, is annotated. Usually, only those functions or services are annotated that are non-trivial, and that the user of the analysis has control of, but not native operating system calls, for example, or other services that enjoy very high confidence. In other words, it would not be useful to monitor every simple getter and setter method other than producing much overhead, but network transport or algorithmic methods might be good indicators for performance problems.

After a brief overview of the architectural levels of analysis, the following Sections 3.1 and 3.2 present the solution idea and the requirements. Then the most important data structures are described in Section 3.3, before the steps of processing are explained in detail in Sections 3.4 and 3.5. Finally, Section 3.6 gives an overview about the presentation of the results of the analysis.

Architecture

Tpan processes three architectural levels, that form a hierarchy: *operations*, *components*, and *deployment contexts*.

1. An operation is a service offered by the application. Each time it is invoked, an *execution* is created, and logged by Tpmon, if its operation has been instrumented.
2. A component is a group of services, usually sorted by functional relationship.
3. A deployment context relates to the execution environment which hosts the application.

In practice, operations are mapped to Java methods, components are Java classes, and deployment contexts correspond to Java Virtual Machines.

3.1 Solution Idea

The problem has similarities to optimization problems in that an absolutely correct diagnosis is desired, yet not mandatory. Getting clues in the right direction is sufficient, and the procedure need not be deterministic. From this perspective, methods such as genetic algorithms [Goldberg, 1989], neural networks [Haykin, 1999], or swarm intelligence [Bonabeau et al., 1999] come into mind.

On the other hand, these methods require a certain amount of training, that means a number of failures as well as belonging faults have to be induced into the components to have the analysis software learn the relations. This can be classified as a kind of “defect testing”, whose quality highly depends on the proper selection of test cases and test data [Sommerville, 2001, p. 449]. Another argument against these methods is that their ability to efficiently adapt to structural changes (that lead to performance loss, but not necessarily to failures) is of no use in our approach. Since our analysis should not require any training, and no adaption to special architectures, but it should instantly (sufficient monitoring supposed) deliver an adequate result when applied to an “unknown” system, these methods do not seem to be suitable strategies for the analysis after all.

Instead, the idea is to analyze and aggregate the monitored timing behavior data in combination with the dependencies on the three architectural levels in a software tool we call the *Correlator*. Simple logic rules are applied to estimate whether an anomaly has been reported correctly, or in turn, whether one might remain undetected as an effect of error propagation as mentioned in Section 2.3. Ideally, one single element of the application can be highlighted at the end of this process, denoting the likely cause of the failure.

An overview of the four main processing steps and their relations is given in Figure 3.1. They are sketched in the following paragraphs, and explained in detail in Sections 3.4 and 3.5.

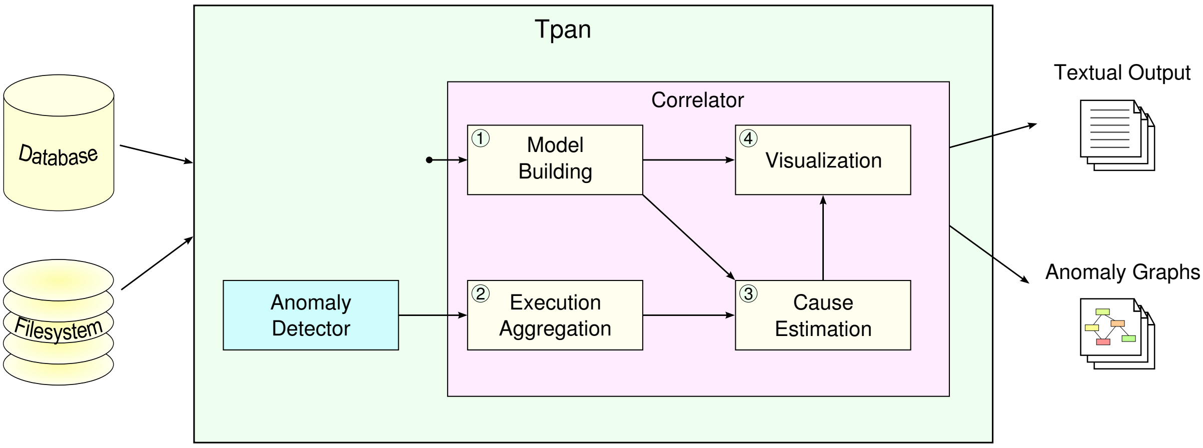


Figure 3.1: Overview of the analysis, centered on the correlation activities. Once loaded and pre-processed by Tpan and an anomaly detector, the Correlator performs further analysis on the data to determine the cause of failure, and prepares the results for presentation in text and graph form.

3.1.1 Model building

Once the required data has been gathered and pre-processed by Tpan, the caller–callee relations of the executions as well as their *anomaly score* evaluations by an anomaly detector, are combined into a model of the application under analysis. This graph-like representation of the application contains both static relations of the elements of the architectural structure, and anomalies in their dynamic behavior.

In contrast to the list of characteristics by Gruschke [1998b, pg. 4], our dependency graphs are exactly the other way round.

- The elements have more than two *states*. Not restricted to faulty or correct, they contain decimal anomaly scores as well as higher level ratings.
- We assume that the model is *complete* in terms of functions that are actually used.
- We assume that the model is *not dynamic* – at least at the current stage of our research.
- More dependencies, thus more *connectivity* in the graphs, are supposed to benefit the results of the analysis.

3.1.2 Aggregation

The anomaly scores of all executions are aggregated by operations in order to get an *anomaly rating* for each operation: A single number that represents the raw summarized timing behavior for an operation. This can be used to get a first graphical impression of the application under analysis – Figure 3.2 shows an example of this step's result. Likewise, for each level in the hierarchy, the ratings of lower levels can be aggregated to

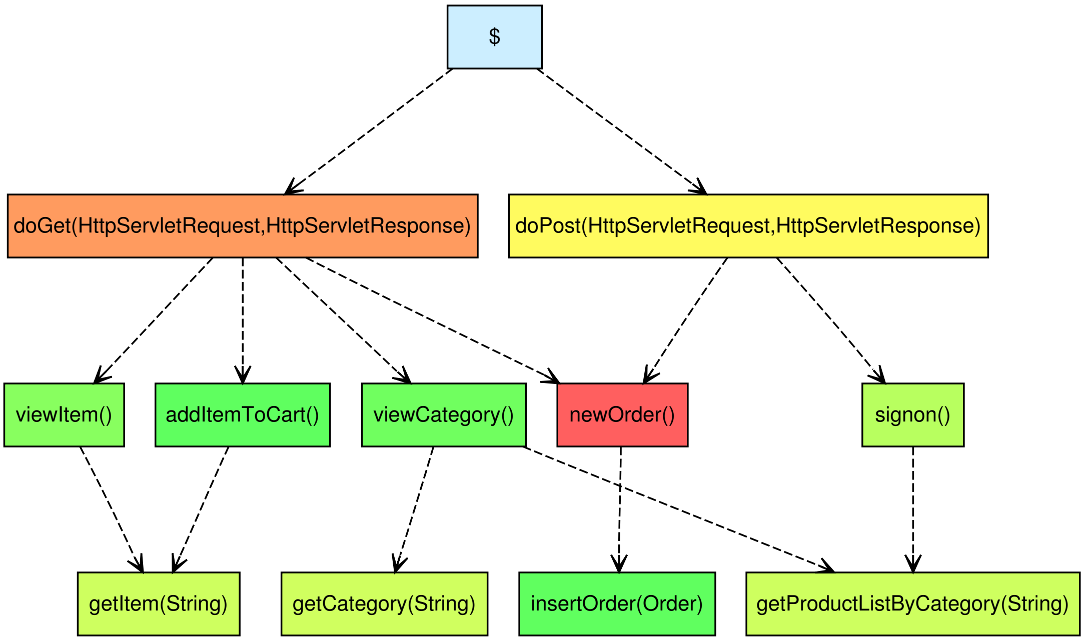

$
doGet(HttpServletRequest,HttpServletResponse)
doPost(HttpServletRequest,HttpServletResponse)
viewItem()
addItemToCart()
viewCategory()
newOrder()
signon()
getItem(String)
getCategory(String)
insertOrder(Order)
getProductListByCategory(String)

Figure 3.2: Example of the first results of the analysis. The system is visualized as a dependency graph. An anomaly rating is assigned to each operation, based on the anomaly events on execution level, displayed in pseudo colors: The darker the red is, the higher are the anomaly ratings.

get high-level ratings. This is suited for an overview of the application as to which parts are most affected by the anomaly: If a component contains a large number of high-rated operations, then there is reasonable suspicion that the cause of the failure is located in this component.

3.1.3 Correlation

Further analysis is needed because these parts where anomalies manifest are not necessarily the causing parts. The dependencies between operations are considered to face the problem of propagation. For example, if an operation shows anomalous behavior, then its callers will probably show anomalous behavior as well, although they are not the cause of the failure.

The idea of correlation is based on the assumption that the effects of timing behavior anomalies are propagated through the anomaly graph in the bottom-up direction. Under some circumstances, anomalies mask with the timing behavior of normal executions, so that their effects disappear in statistical noise. On the other hand, if a faulty operation is executed in a loop, for example, thus having the timing effects sum up for each invocation, its anomaly shall become noticeable, not only at the origin, but also at all dependent services. Ideally, the correlation is able to perform a negation of the propagation effect. In contrast to aggregation, a certain intelligence is needed, i.e. further assumptions about typical propagation behavior.

3.1.4 Visualization

The results of the correlation processing are ordered lists of elements of the application under analysis, each assigned an estimation whether to be the cause of failure. This can be written out in text form in files, or on console interface, and illustrated as pseudo-colored dependency structures.

Pseudo colors are a special case of false colors, and a mechanism to display information that originally has no color at all. For example, the temperatures in an infrared thermography, and the height levels in radar satellite images are represented by different colors from a part of the visible spectrum.

Since most of the processing effectively deals with an abstract mesh of nodes in the dependency graph, the level of the monitored system is unimportant from the perspective of the correlation algorithm. What we call components could be Java classes, or whole distributed applications. Thus, the precision of failure diagnosis depends on the precision of the monitoring: If the instrumentation is limited to components, the Correlator is not able to narrow down the cause to operation level.

A downside in using dependency graphs, as mentioned by Gruschke [1998b], is that program-specific characteristics, which cannot be mapped to dependencies, cannot be considered by the correlation. This example is given: “If event A and event B arrive within thirty seconds, then ignore both.” Since the Correlator to be developed in this thesis is supposed to be activated on demand – after a failure has occurred – and the examined monitoring data is from a short time period, this approach will disregard the *chronology* of the anomalies. It is assumed that all reported anomalies indicate symptoms of a single cause, or are at least somehow related to each other. A separation of anomaly groups and bursts would however become necessary in case of an extension to continuous operation.

3.2 Requirements

The prototype for an anomaly Correlator that is developed as part of the thesis has to satisfy the following requirements:

1. Plug-in for Tpan.

   The Correlator prototype has to be developed in Java as a plug-in for Tpan to use existing functionality for the analysis and the preparation of monitoring data. Input data can be gathered from the file system, or from a database. The plug-in is controlled through the user interface of Tpan, and provides interfaces to retrieve raw correlation results.

   Existing components in Tpan can create *message traces* – ordered lists of caller–callee relations – and anomaly evaluations from monitoring data. These suffice as input for the Correlator.

2. Three levels of aggregation.

   All architectural levels that are available from input data – operations, components, and deployment contexts – are included in the processing, and become part of the visualization.

3. Construction of dependency graphs.

   From the collections of message traces, a hierarchical directed graph is constructed that represents the calling dependencies between operations, components, and deployment contexts.

4. Calculation of anomaly ratings based on anomaly status and dependencies.

   The executions of operations are evaluated by an anomaly detector to judge each one to be anomalous, or not. Combined with the dependency information, these ratings are refined to narrow down the location of the cause.

5. Estimation for higher levels to be the cause of failure.

   The operations' anomaly ratings are conducted into cause estimations on component level as well as on deployment context level. This is again done based on simple aggregation of the ratings rather than based on the dependencies.

6. Configuration through properties files.

   Since there is a bunch of settings available for the algorithm and the output, as long as there is no graphical user interface, a properties file seems to be a better choice than hard-coded values, or a poll on command line interface for each run.

7. Textual output.

   The minimum required output of the Correlator is a list of the components, each assigned a percentage to be the cause of failure, sorted in descending order by this percentage.

8. Graphical output.

   In addition to the textual output from within Tpan, a graphical representation of the dependency structure is created. The cause estimations are displayed through colors as already shown in the example in Figure 3.2 on page 19. The main requirement for the graphics is clearness: A human viewer shall quickly recognize which parts of the application under analysis are probably the cause of a failure, so that further efforts on fault localization can be adjusted accordingly.

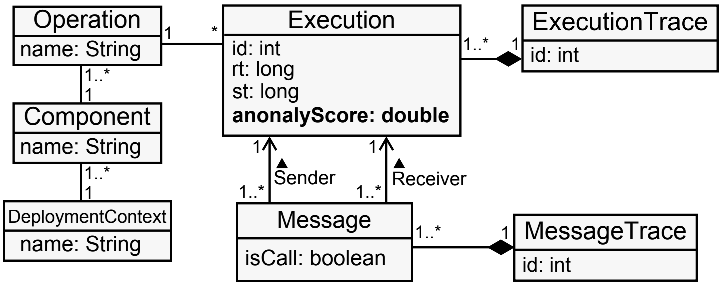


Figure 3.3: Data structures produced by Tpan [Rohr, 2008].

3.3 Data Structures

As described before, and depicted in Figure 3.1, the Correlator is developed as a plug-in for Tpan. Tpan offers functionality to load and process monitoring data. For further processing by the plug-ins, it provides message traces. Additionally, an anomaly detector can be integrated to evaluate the timing behavior of executions, whose results can also be retrieved by plug-ins. The goal of the Correlator plug-in is to gain more information from the available data, especially – assuming the existence of a failure – about possible causes of failure.

3.3.1 Input

Figure 3.3 [Rohr, 2008] shows the data structures that are produced by Tpan, thus being input data for the Correlator. The starting point is the *operation*, which represents a function, method, or procedure – common elements in modern programming languages. The operations are organized in *components* and *deployment contexts*. For timing behavior anomaly detection, the interest lies in the *execution* as denoting the activity of its operation when it is called during runtime. Besides *response time* and *start time*, it contains a decimal *anomaly score* that is a measure of its timing behavior. The score is evaluated by an anomaly detector, and lies in the interval $[-1, 1] \in \mathbb{R}$, where a score of -1 means that the execution has a normal timing behavior, and 1 means that the detector considers this execution to be extremely anomalous. A dependency graph can be calculated from the *message traces*.

3.3.2 Output

The main classes representing the output of the correlator are those specifying the hierarchy structure. The hierarchy can be considered to be three-dimensional: One dimension contains the *architectural* levels – from executions to deployment contexts – whereas two dimensions are needed to form the calling *dependencies*.

The architecture is actually a tree, sketched in Figure 3.4, with the `Application` as root and the `Executions` being the leaves. On the other hand, the dependency structure is a net, sketched in Figure 3.5, where the (directed) edges represent function calls be-

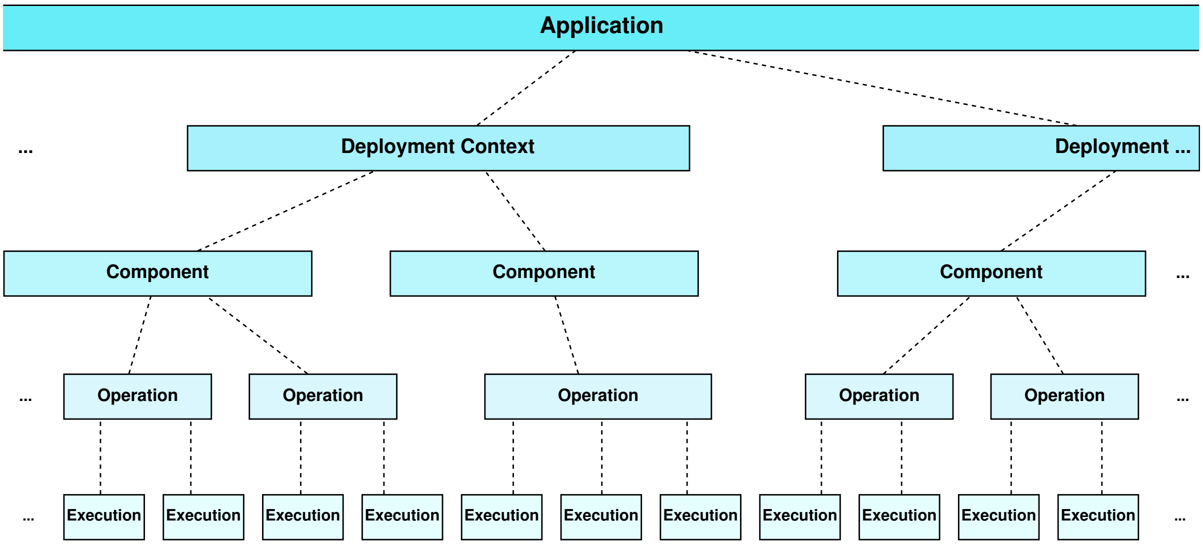


Figure 3.4: Structural hierarchy tree. From `Executions` to `Operations` to `Components` to `Deployment contexts` to the `Application` as a whole.

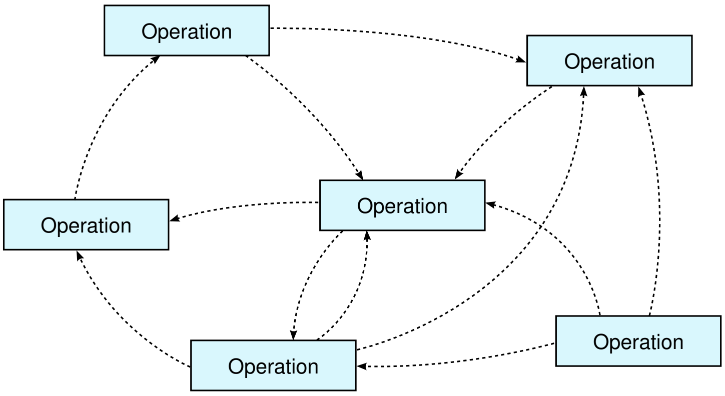


Figure 3.5: Dependency net on operation level. Circles and joins are allowed in the graph.

tween the elements. These edges are only allowed between nodes on the same hierarchy level. The graph may contain circles (loops in the source code, or bottom-up calls) and joins (different methods call one and the same other method). That way, there visually exist separate dependency nets on each hierarchy level. But in fact, because the real interaction takes place on the operation level, the connections on the higher levels are only aggregated through their respective children in the graph. For example, the component-external connections (those that point to a foreign component) of all operations within a component are aggregated into inter-component edges.

3.3.3 Structure Classes

In software design, this above-mentioned collection of interconnected graphs can be reflected by a set of classes each representing one type of node in the structure as depicted in Figure 3.6. Each element in the graph representation is linked with its parent, children, and neighbor elements through object pointers.

Application is the top level class in the hierarchy that is instantiated only once per runtime. It is therefore not connected with any other object of the `Application` class, but instead encapsulates the other hierarchy levels. It provides methods to build up the structure, to initiate the anomaly evaluation, and to fetch data for the presentation of the results. This way, it also serves as a connector to the environment.

StructureElement is the abstract superclass of all classes that are part of the dependency structure. It defines common attributes like the links to parents and children (on higher or lower levels) as well as incoming and outgoing connections to neighbor elements (on the same hierarchy level), including the number of connections. It provides methods to build up the linkage, and helper methods for anomaly evaluation such as counters for various properties, or for creating histograms of anomaly scores.

The major function of the elements besides forming a structure is their anomaly state. Like with the execution level's *anomaly score*, the states of the higher level elements are primarily defined through a decimal number we call *anomaly rating*. The alternative would be a classification to discrete states – a reduction to only "good" or "bad" behavior in the extreme – but that would be hard to judge. Furthermore, additional parameters and thresholds would be needed that have to be specified at first to later analyze their impact in the experiments. Agarwal et al. [2004, pg. 6] mention: "It is very difficult and error prone for the system administrator to configure a threshold for a component without extensive benchmarking experience".

The `StructureElement` class implements the Java `Comparable` interface to allow an easy sorting of the elements.

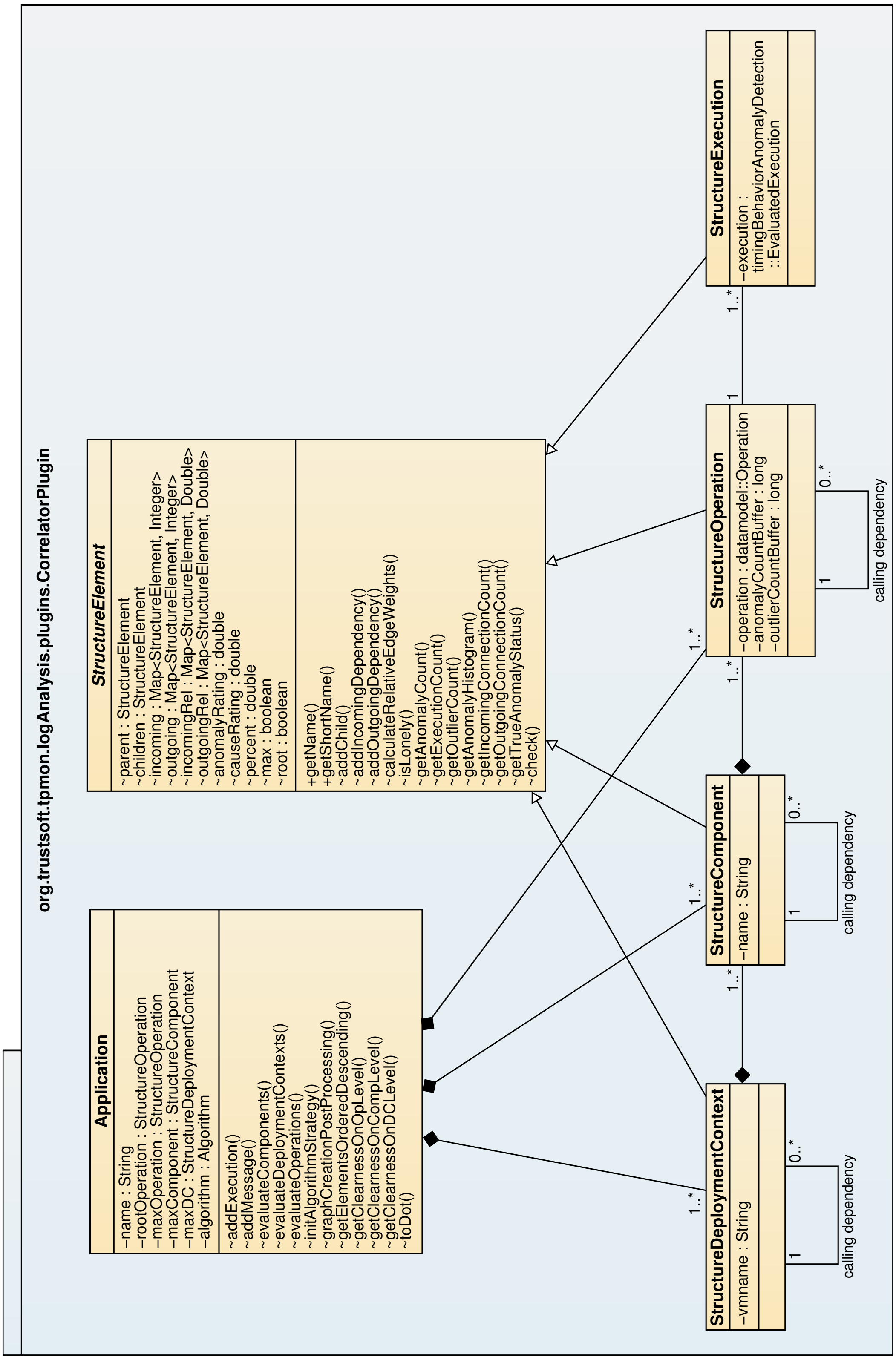

org.trustsoft.tpmon.logAnalysis.plugins.CorrelatorPlugin
Application
−name : String
−rootOperation : StructureOperation
−maxOperation : StructureOperation
−maxComponent : StructureComponent
−maxDC : StructureDeploymentContext
−algorithm : Algorithm
~addExecution()
~addMessage()
~evaluateComponents()
~evaluateDeploymentContexts()
~evaluateOperations()
~initAlgorithmStrategy()
~graphCreationPostProcessing()
~getElementsOrderedDescending()
~getClearnessOnOpLevel()
~getClearnessOnCompLevel()
~getClearnessOnDCLevel()
~toDot()
StructureElement
~parent : StructureElement
~children : StructureElement
~incoming : Map<StructureElement, Integer>
~outgoing : Map<StructureElement, Integer>
~incomingRel : Map<StructureElement, Double>
~outgoingRel : Map<StructureElement, Double>
~anomalyRating : double
~causeRating : double
~percent : double
~max : boolean
~root : boolean
+getName()
+getShortName()
~addChild()
~addIncomingDependency()
~addOutgoingDependency()
~calculateRelativeEdgeWeights()
~isLonely()
~getAnomalyCount()
~getExecutionCount()
~getOutlierCount()
~getAnomalyHistogram()
~getIncomingConnectionCount()
~getOutgoingConnectionCount()
~getTrueAnomalyStatus()
~check()
StructureExecution
~execution :
timingBehaviorAnomalyDetection
::EvaluatedExecution
StructureOperation
−operation : datamodel::Operation
−anomalyCountBuffer : long
−outlierCountBuffer : long
StructureComponent
−name : String
StructureDeploymentContext
−vmname : String
1..*
1
0..*
1
calling dependency
1..*
1..*
1..*
1..*
0..*
1
calling dependency
1..*
0..*
1
calling dependency

Figure 3.6: Structural classes of the Correlator. Getters and setters as well as other trivial methods are omitted.

StructureDeploymentContext is the highest level "real" structure element that is typically represented as an object in the resulting graph. It contains components, and may be linked with neighbor deployment contexts by a *uses* relation. Its only private attribute is the name of its virtual machine.

StructureComponent is the intermediate level structure element. Usually, it has a deployment context as parent, may be linked with neighbor components, and has operations as children, that are again linked among each other. Its only private attribute is its name, for example a qualified Java class name.

StructureOperation is the lowest level regular structure element. It contains executions, and may be linked with neighbor operations. Furthermore, it contains the "original" `Operation` from the `datamodel` package in Tpan that is integrated into the Correlator model in the first step of processing.

StructureExecution is included in this view for practical reasons. Although it is not a functional element that has to be designed and written like the other parts of the application, as being the instance of an operation, it shares many attributes that are useful for anomaly and cause analysis. It contains the "original" `EvaluatedExecution` from the `timingBehaviorAnomalyDetection` package in Tpan that is integrated into the Correlator model in the second step of processing. Most importantly, this provides the results of the anomaly detection such as start and response times, anomaly and outlier status, and the anomaly score – the foundation of all following analysis. Usually, there is no need to link the executions with each other, but it might be useful to order them by time.

3.4 Dependency Graph Creation

In preparation of the analysis, a dependency graph is computed and loaded with required information. This graph is a three-dimensional, hierarchical structure containing all elements of the software under analysis that are linked among each other corresponding to their calling dependencies. The operations each contain a set of executions, they are linked with dependent and depending neighbor operations, and they are clustered into components and deployment contexts. As mentioned in Section 3.3.1, the input data is a collection of message traces, whose context can be seen in Figure 3.3 on page 22. These traces are processed according to the following steps, that are visualized in Figure 3.7.

1. In the `Application`, for each hierarchy level – operations, components, and deployment contexts – a `Map` is created to successively hold instances of the structure elements as soon as they become known. Instances of the original objects – `Operation` objects for operations, and the names of components and deployment contexts – are mapped to the Correlator-internal structure elements.

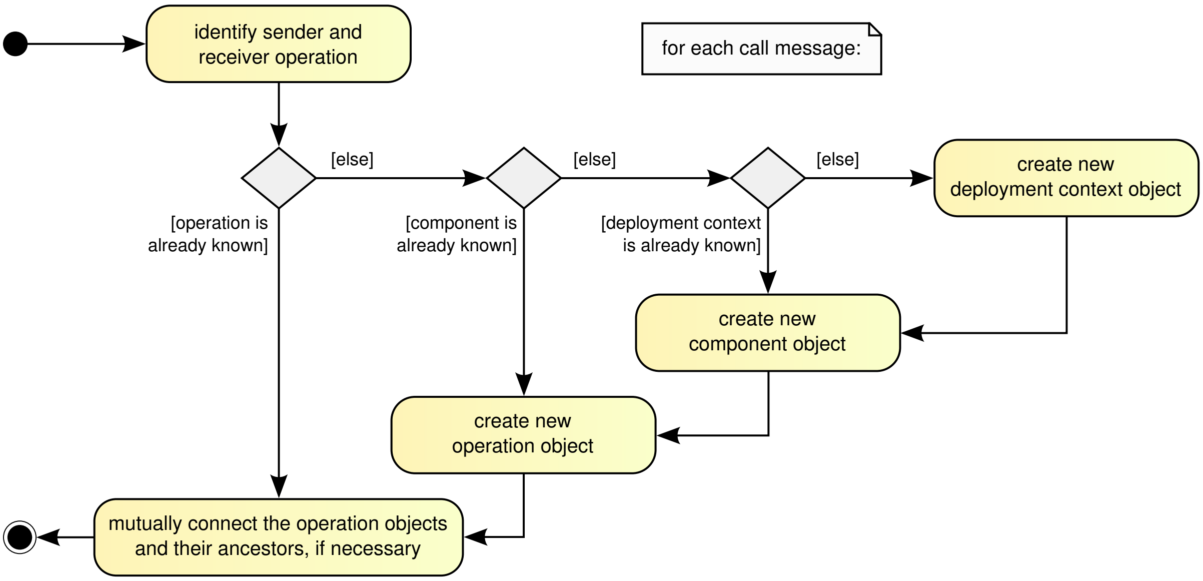

identify sender and receiver operation
for each call message:
[else]
[else]
[else]
create new deployment context object
[operation is already known]
[component is already known]
[deployment context is already known]
create new component object
create new operation object
mutually connect the operation objects and their ancestors, if necessary

Figure 3.7: Construction of the dependency graph: The involved operations of each message from a message trace as well as their components and deployment contexts are created as `StructureElements` and mutually connected where appropriate. The executions are subsequently added to their operations.

2. For each *call* message, the operations are identified that send or receive the message, respectively. The *return* messages are ignored, because it is assumed that they provide no new information, but only a "mirror" of the call messages.

3. Each `Operation` is looked up in the map, and assigned a new `StructureOperation` if necessary. Doing this, the related `StructureComponent` is looked up with its name, and in turn assigned a new `StructureDeploymentContext` if necessary.

4. Once all objects are known, they are connected to each other through object assignments, again using the bottom-up pattern: First, the dependency connections are established bidirectionally on operation level. For each pair of operations, if their parent objects (namely components) are unequal, a connection is established between these two. And if they are located at different deployment contexts, these are also linked with each other.

5. Finally (not shown in the figure), a list of `EvaluatedExecutions` is integrated into the dependency structure by simply encapsulating each one into a `StructureExecution` object and mutually connecting these with the related `StructureOperation`.

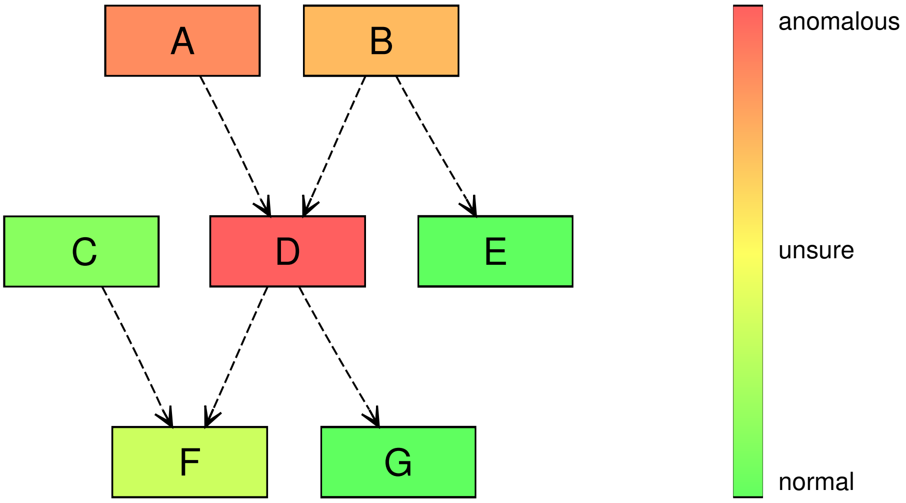


Figure 3.8: Example of anomaly propagation. The anomaly in `D` is propagated to `A` and `B` (that depend on `D`), but not to `C` and `E` (that are neighbors of `D`), and not to `F` and `G` (that `D` depends on). The degree of anomaly is represented by color shades from green to red.

3.5 Analysis

The idea of the correlation approach presented in this section is based on the assumption that timing behavior anomalies *propagate* through an application in the backwards direction of the dependency hierarchy. The effects of this propagation can thus be analyzed to draw conclusions on the origin of the anomalies, and to assign a high probability to the component that contains the cause of failure.

For the processing to be clear and manageable, the Correlator does not examine all available data at once, but instead looks at small groups of adjacent nodes in the dependency graph, from the perspective of one node after another. These *configurations* are tested for pre-defined conditions (e.g. the occurrence of a certain anomaly rating in a certain position) that allow some deduction.

In this context, *element X depends on element Y* if `X` *uses* a service of `Y`, and thus an error (in terms of timing behavior anomalies) in `Y` might be propagated to `X`.

Figure 3.8 shows an example situation of the expected behavior of anomaly propagation. It is expected that the majority of the callers of an anomalous element show some degree of anomalous behavior, too. The neighbor elements – not directly connected to the originating element – are not directly influenced, as well as elements that the anomalous element itself depends on. In the example, the anomaly in element `D` is propagated to the elements `A` and `B` that depend on D, while the neighbor elements `C` and `E` show no significant sign of anomalous behavior. Although `F` and `G` are directly connected to `D` in the dependency hierarchy, so there is the possibility to affect the performance of `D`, it is assumed that timing behavior anomalies can only be propagated against the calling dependencies, thus `F` and `G` are not influenced by an anomaly in `D`.

The essential part of the analysis is the application of an *algorithm* that interprets these and other configurations to draw conclusions about the cause of failure. The mathematical basis of its calculations is the *rating* – a single decimal number in the range of $[-1, 1] \in \mathbb{R}$ that reflects the status of each element of the dependency structure to what extent it is suspected to be the cause of failure. The maximum value of 1 means a significant anomaly, while -1 means perfectly normal behavior, and 0 means that the classification is ambiguous. This applies to the input anomaly score as well as to the resulting cause rating which can easily be converted to a percentage as shown in Equation 3.1 where r is the rating, and p is the resulting percentage.

$$p(r) = \frac{r+1}{2} \cdot 100 \tag{3.1}$$

As described in Section 3.1, the goal is to get a simplified version of the graph. If a single cause cannot be found, the anomalies should at least be less scattered, but concentrated at some regions of the graph to ease further diagnosis.

3.5.1 Preconditions

Although this approach to fault localization is based on anomalies in the *timing* behavior, its role is not to care precisely about timing behavior. Likewise, the task is not the guessing of the circumstances of failures, for example an answer to the question whether a failure actually happened, or whether there are overlaps of more than one effect. It is assumed that other tools, or administrators have recognized the scenario, and decided to initiate the Correlator to help locate the cause of failure, after an anomaly detector has evaluated the timing behavior. The following conditions have to be fulfilled:

There has been exactly one failure in the observation period. The Correlator still presents reasonable results if the input data does not contain any real anomalies, or if they are equally distributed, but this results would have no significance beyond failure diagnosis. For example, it will not make much sense to put the Correlator on pure performance analysis.

The failure has a distinct cause. The analysis is aimed to locate one point of failure, that is one single fault in the application. Although the result might be ambiguous, the Correlator does not actively try to distinguish separate elements of the application as possible faults.

The measurements are correct. Minimal effort is taken to doubt the results of the anomaly detector. It is assumed that the monitoring is accurate, and the anomaly detector has adequately evaluated the meaning and relevance of the timing behavior and thereby handles aspects like statistical outliers. Therefore, the Correlator does not examine the raw response or execution times, but instead relies exclusively on the anomaly score.

3.5.2 Strategy

The challenge is to effectively *reduce* the amount of information in a way to not lose anything important. The calculations shall be concise, but they shall not become imprecise. For example, in theory, every single data set of each execution – including start time, response time, etc. – could directly influence the calculation of the cause of failure on deployment context level. In practice however, an iterative process seems to make more sense.

The first essential simplification is to completely omit the concrete timing information from the evaluation of the executions. It is assumed that the input data is from a short time period just before the failure has occurred, thus the start and response times would not help much, but only complicate things. The interpretation of the anomaly intensity – what can be concluded if an execution takes significantly more or less time than what has been defined as "normal" – that results in the anomaly score, is the task of the anomaly detector.

The anomaly information is evaluated and aggregated separately on each hierarchy level in a bottom-up way. Doing this, the reduction should not happen too "fast". For example, a simple evaluation like "IF value X exceeds threshold Y THEN set rating A to level B" should be set aside. A trade-off is to include the size of the *deviation* of a value from some other value (might be a threshold, or a calculated value) in its evaluation. Thus, instead of a simple, rule-based decision system, the approach is geared more to the concept of *neural networks*: The intensity of signal A changes dependent on the deviation of input X. This way, in a consistent implementation, the algorithm does not need any pre-defined thresholds that would be hard to determine, but instead it needs a set of transformation rules. With these rules, an algorithm examines certain small-scale configurations of anomalies and dependencies, and draws conclusions from them. The conclusions are then embodied in a single decimal *rating* number – the higher the value, the higher the probability to contain the cause of failure – for each element, and for each step of processing, that can be used for presentation, or for further analysis.

A straightforward procedure is to analyze the situation from the *perspective* of each element in the dependency graph. The elements are defined by their properties, and their environment, consisting of parent, children, and neighbor elements. All of these can influence their timing behavior, thus the algorithms should consider them for the calculation of their rating just as well.

Referring to Figure 3.8 on page 28 for example, from the perspective of `A` and `B`, there is at least one element they depend on that behaves anomalous, so it can be concluded that `A` and `B` themselves would probably not be the cause of failure, because their anomalies seem to be propagations from `D`. An evaluation algorithm could therefore lower the ratings of `A` and `B` to reflect their innocence. From the perspective of `C`, neither being connected to an anomalous element nor being anomalous itself, no conclusion can be drawn. Likewise, from the perspective of `E` and `G`, no conclusion can be drawn, because there is only one *incoming* connection each, namely from the dependent elements `B` resp. `D`. Although these behave anomalous, so it is possible that `E` resp. `G` contain the cause, this statement is vague, because from their view, `B` resp. `D` could be the cause themselves, or they could be affected by other elements. From the perspective of `F`, again there is one dependent element that behaves anomalous, but there is another that does not, so it can be concluded that `F` has an only slightly increased probability to be the cause of the anomaly in `D`. However, if `C` would behave anomalous, too, the situation would be more clear, and an algorithm could significantly increase the rating of `F` for containing the cause, no matter if its own timing behavior seems to be unobtrusive.

3.5.3 Realization

There are two procedures of analysis that have been outlined in Section 3.1.

Aggregation For each element on a specified hierarchy level, the anomaly ratings of all subordinate elements are evaluated and combined to get a new single rating. The purpose is to effectively reduce the data to ease follow-up evaluation, yet preserving most of the meaning of the original data. The aggregation is limited to the *local* information from the perspective of the element, that means, the child nodes in the dependency graph, but not their neighbors, or their position in the structure. In practice, this is done by using a mean calculation.

Correlation For each element on a specified hierarchy level, another rating value can be calculated to now include the *environment* information from the perspective of the element. Instead of indicating the distribution of the *effects*, the purpose is to give an estimation about the *cause* of failure. The calculation is based on a set of rules to consider the configurations of anomalies and dependencies as described in Section 3.5.2. Other available information, such as the mappings to higher-level structures, might be considered as well. The definition of the environment as well as the conditions under which a configuration has increasing or decreasing influence on the rating, and in what extent, is up to the specific implementation.

The combination of aggregation and correlation has similarities to the concept of *cellular automata* [Wolfram, 2002], in that (1) a new state is calculated based on the state of a specified element; (2) the state depends on the states of neighbor elements in a distinct environment; (3) the structure of elements remains constant during processing; (4) the calculation is successively executed for all elements; (5) each element has the same rule for updating.

The processing in the prototype is done in the following four steps:

- The first step is the aggregation on operation level. For each operation, the *anomaly scores* of all contained executions are combined into an *anomaly rating*. An example distribution of execution anomaly scores within an operation is shown in Figure 3.9. The result of this step suits as a first overview of anomaly distribution as shown in Figure 3.2 on page 19 and may suffice as help for simple problems, where the anomaly is outstanding, and/or an administrator can instantly deduce the cause.

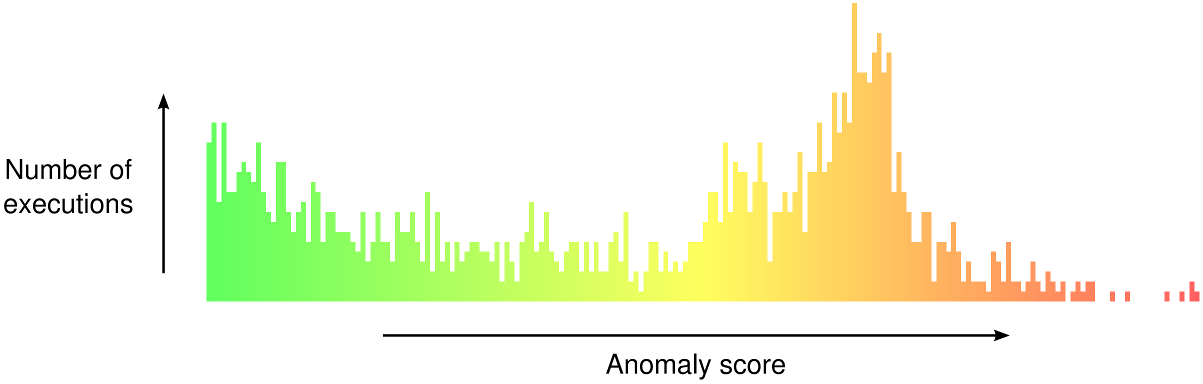


Figure 3.9: Example of a histogram of anomaly scores. The horizontal axis shows the values of the scores from −1.0 (green) to +1.0 (red) while the vertical axis shows the relative number of scores.

- Then an algorithm for correlation is applied on operation level. Additional information is gathered through the application of pre-defined rules. For each operation, the local anomaly ratings as well as the neighborhood ratings and structural informations are examined for certain configurations depending on the complexity of the algorithm to gain a *cause rating*. Again, this might suffice for some problems.

- For this implementation being a prototype, and because the correlation can be a very complex task, and its benefit has yet to be shown, it is only executed on operation level. Therefore, the main purpose of the third and fourth step is aggregation again, namely on component and on deployment context level, to further help the administrator in isolating the cause of failure.

The algorithms itself are loaded through a plug-in mechanism and can be substituted. New variants can be added by extending the abstract `Algorithm` super class, that is additionally prepared to be configured through Java `Properties`. Three implementations are created in the thesis that differ in complexity, and in their possibilities to be customized.

A controversial customization seems to be the method of mean calculation. For example, if there are very few anomalies in a large number of normal executions, it is difficult to decide whether they are part of the statistical noise, or whether they are indicators of the failure. Common methods include the *arithmetic mean*, the *median*, and the *power mean*. The power mean (see e.g. [Cantrell and Weisstein, 2003]) as shown in Equation 3.2 is also called *generalized mean* because it is a superset of other mean variants that differ only through their exponent. The arithmetic mean, for example, is a special case of the power mean with exponent $p = 1$. Similarly, the *root mean square* is equal to the power mean with exponent $p = 2$.

$$M_p(x_1, x_2, ..., x_n) := \left(\frac{1}{n} \sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}} \tag{3.2}$$

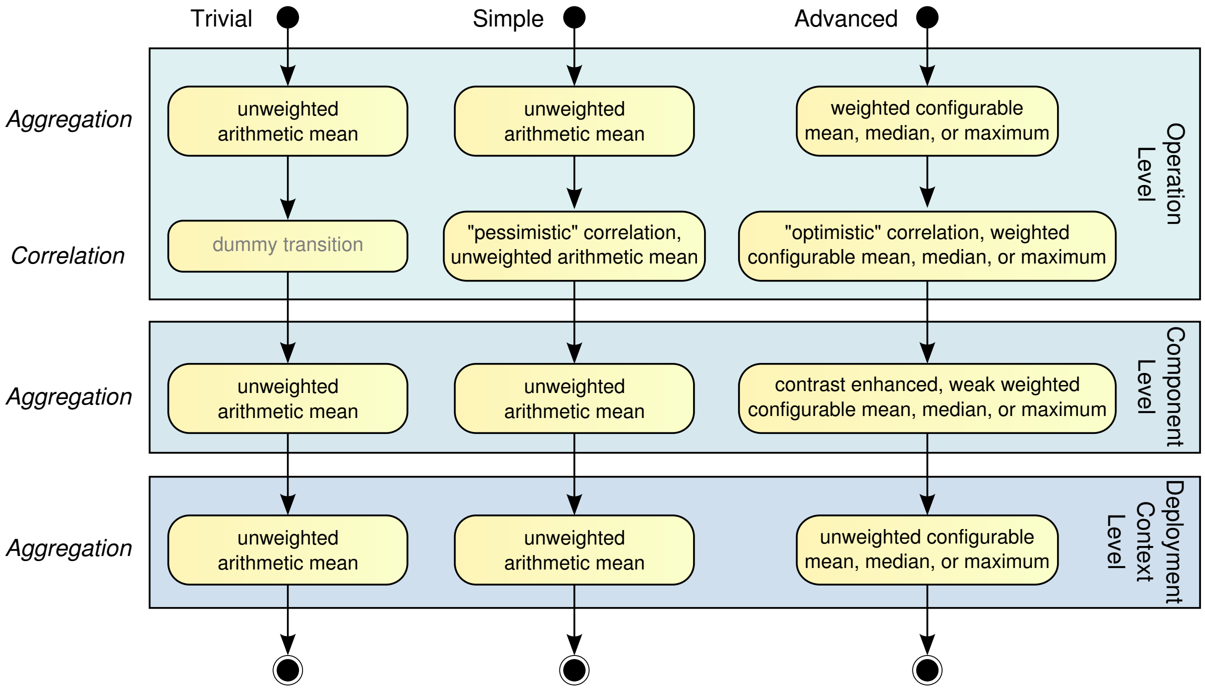

Trivial
Simple
Advanced
Aggregation
Correlation
Aggregation
Aggregation
unweighted arithmetic mean
unweighted arithmetic mean
weighted configurable mean, median, or maximum
dummy transition
"pessimistic" correlation, unweighted arithmetic mean
"optimistic" correlation, weighted configurable mean, median, or maximum
unweighted arithmetic mean
unweighted arithmetic mean
contrast enhanced, weak weighted configurable mean, median, or maximum
unweighted arithmetic mean
unweighted arithmetic mean
unweighted configurable mean, median, or maximum
Operation Level
Component Level
Deployment Context Level

Figure 3.10: Overview of the algorithm variants: There are three ways, differing in complexity, to get from anomalies on execution level to a cause evaluation on deployment context level.

The main difference to the median is the dealing with outliers within the set of samples. The concept of the median implies to cut off outliers, while the outliers are always included in the power mean calculation, depending on the power mean exponent: An exponent smaller than one decreases their influence on the result, an exponent greater than one increases the influence, while an exponent equal to one is actually the arithmetic mean where all samples have equal influence on the result.

Another aspect in the context of mean calculation is the weighting of samples by some criteria. For example, the number of connections between the nodes in the dependency graph could be considered as influence on the calculation of the neighborhood anomaly ratings to reflect the mutual cohesion in relation to other connections. This can be integrated into the mean calculation by assigning a weight factor to each sample in the data set.

Both of these aspects, the mean method and the weighting, are implemented, and are subject of the evaluation in Chapter 4 from page 47.

Figure 3.10 shows an overview of the implementation variants that are explained in detail in the following paragraphs.

Trivial

The trivial algorithm performs aggregation only. In other words, it does no “real” correlation at all, but only forwards the ratings between the aggregation levels. The aggregation is done through an unweighted arithmetic mean calculation on each level as shown in Equation 3.3, where r is the rating of the currently viewed element, and n is the number of subordinate rating samples, from which the mean is calculated. Its main purpose is to have a basis for the evaluation of the efficiency of the other variants, whose results shall not be worse than this variant's results.

$$\overline{r} := \frac{1}{n} \cdot \sum_{i=1}^{n} r_i \tag{3.3}$$

Simple

This variant performs a "pessimistic" correlation. That means that only few rules are used to detect configurations in the anomaly structure that are relatively clear to understand and to implement. Following a "keep it simple" directive, two specific conditions are tested, and an increase or decrease flag is set, respectively. For example, some additional information – like the outlier status of executions – is completely ignored, because their effects are unknown and could be misleading. The cause rating is then derived from the anomaly rating, according to the flags.

More precisely, the rating is increased if the unweighted arithmetic mean of the anomaly ratings of the directly connected callers (upwards in the calling dependency graph) is greater than the anomaly rating of the currently calculated operation plus a tolerance value. This means that this operation is likely to be the cause of failure, because the dependent operations show significant anomalies.

The rating is decreased if the maximum of the anomaly ratings of the directly connected callees (downwards in the graph) is greater than the anomaly rating of the current operation plus tolerance. This means that this operation's rating is likely to be a propagation from another operation it depends on.

Under all other conditions, as well as in special cases such as singular connections, and the root operation, the value of the anomaly rating is forwarded unchanged.

In Equation 3.4, a is the local anomaly rating, c is the resulting cause rating, t is the tolerance, $\overline{a_{in}}$ is the unweighted arithmetic mean of the anomaly ratings of the caller operations (incoming connections) according to Equation 3.3, and max_{out} is the highest anomaly rating of the called operations (outgoing connections).

The function for increase and decrease is chosen for its simple linear curve staying in range $[-1, 1]$. Again, the aggregation is done through an unweighted arithmetic mean calculation on all three levels.

$$c := \begin{cases} \frac{a+1}{2}, & \overline{a_{in}} > a + t \\ \frac{a-1}{2}, & max_{out} > a + t \\ a, & else \end{cases} \tag{3.4}$$

Advanced

Compared to the simple algorithm, this variant has some additional features that do not imply to produce better results, but extend the possibilities for experimentation. It can be called “optimistic” in that it relies on further assumptions that are rather speculative in part. The samples in mean calculations are now *weighted* by different criteria on each hierarchy level. The neighborhood – elements on the same level whose ratings are included in the calculations – is extended to all elements that can be directly or indirectly reached through the graph, thus adding a *distance* to the weight factor in the correlation. Additionally, the following *parameters* can be configured via Java `Properties`:

- Independent methods of mean calculation for each of the three aggregation steps, and for correlation: Median, power mean, or maximum.
- The exponents of the power mean calculation on each of the four steps.
- The method of edge weight calculation in correlation: Absolute (number of executions), or relative (percentage of executions per connection).
- The relation factor between the influence from caller and callee operations.
- The intensity of the influence of distance from indirect neighbor operations.

While the aggregation and correlation functions work basically the same as in the trivial and simple variants, the following extensions are implemented:

The aggregation on operation level does not ignore outliers in the execution anomaly scores, but instead includes them in the calculation with a weight of one tenth related to the other samples. The aggregation on component level uses the number of executions of the operations as weights, but weakens the influence through the computation of a square root. After the mean calculation, the contrast is raised by squaring the rating. The aggregation on deployment context level remains unweighted, but can be varied through the `Properties`.

The calculation of a weighted power mean is shown in Equation 3.5, where r_i are the sample rating values, w_i are the related weights, n is the number of samples, and e_p is the power mean exponent. The modified power function pwr shown in Equation 3.6 is used to retain the sign of the ratings, because the power mean is originally defined for non-negative values only. The signum function sgn shown in Equation 3.7 extracts the sign of a number.

$$\widetilde{r}(e_p) := pwr\left(\frac{\sum_{i=1}^{n} w_i \cdot pwr(r_i, e_p)}{\sum_{i=1}^{n} w_i}, \frac{1}{e_p}\right) \tag{3.5}$$

$$pwr(a, b) := sgn(a) \cdot |a|^b \tag{3.6}$$

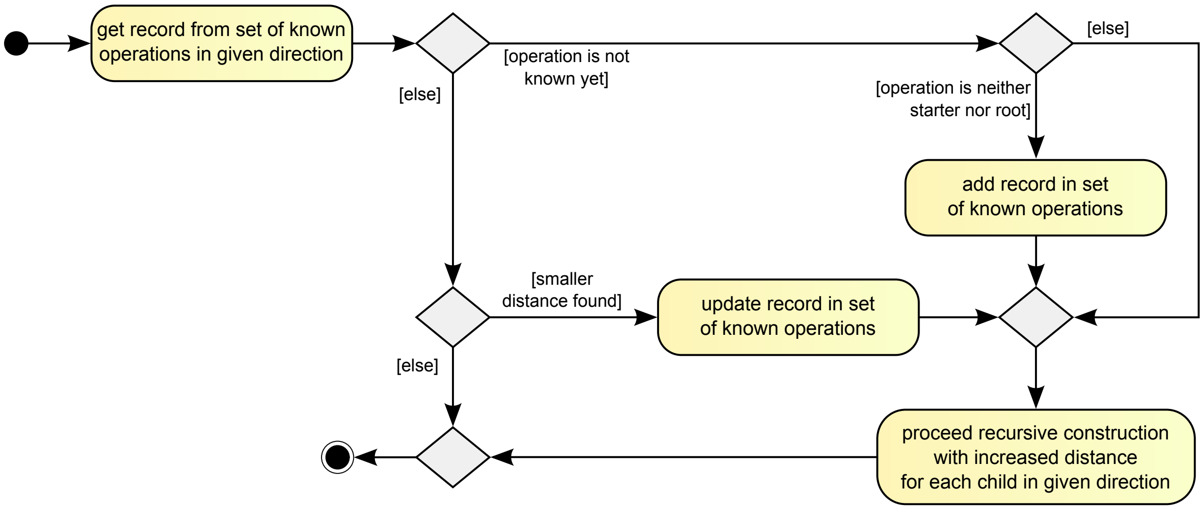

get record from set of known operations in given direction
[operation is not known yet]
[else]
[else]
[operation is neither starter nor root]
add record in set of known operations
[smaller distance found]
update record in set of known operations
[else]
proceed recursive construction with increased distance for each child in given direction

Figure 3.11: Construction of the list of distances and weights. The depicted algorithm is executed recursively for each directly or indirectly connected neighbor operation.

$$sgn(x) := \begin{cases} +1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} \tag{3.7}$$

The correlation function queries the highest rated directly connected callee, like in the simple variant, but additionally includes the mean of directly and indirectly connected operations, callers (predecessors in the graph) and callees (successors in the graph) respectively, weighted by distance and edge weight (absolute or relative execution count), whereas the simple algorithm is restricted to a distance of one “hop”. Figure 3.11 shows the activities to collect the information needed to build up the two tables for each operation: A recursive function traverses the dependency net (example shown in Figure 3.5 on page 23) upwards and downwards in a tree-like depth-first search, storing the length of the shortest path to each directly or indirectly connected other operation, and the number of connections, i.e. the number of executions on that path. It is assumed that a propagated anomaly “looses power” with increasing distance to its origin, so the sample weights w_i are composed of the edge weight divided by the distance as shown in Equation 3.8 where e_i is the edge weight, d_i is the related distance, and e_d is a configurable distance intensity constant.

$$w_i := \frac{e_i}{d_i^{e_d}} \tag{3.8}$$

If a method call is *remote* – i.e. the involved operations are not within the same deployment context – then, compared to local calls, it is likely that the timing measurement is affected by environmental variances (such as network traffic bursts), or that the measurement itself is inaccurate, so that the number of false positives as well as false negatives in the anomaly detection increases. These anomalies may be noticed on higher level examination, but unlike “real” anomalies, they are assumed to be less helpful with the failure analysis, because they probably do not point out software faults, but rather show flaws in the system environment. To take these circumstances into account, the influence of the remote operation’s anomaly rating on the calculation of the cause rating is reduced by simply halving the weight w in the mean calculation.

The transcription from the anomaly rating to the cause rating is not an “either-or” relation with a linear change (like $c = \frac{a+1}{2}$ in the simple variant), but instead depends on the *difference* between the local and (aggregated) neighbor values. In other words, the influence depends on the deviation between the anomaly rating and a reference value, plus optional tolerance. The cause rating is defined as the sum of the anomaly rating plus (or minus) the influence of the caller and callee operations as shown in Equation 3.9, where c is again the resulting cause rating, and a is the local anomaly rating.

$$c := a + influence_{\mathit{callee}} + influence_{\mathit{caller}} \tag{3.9}$$

Figure 3.12 depicts the *conditions* of an increase or decrease, as is also shown in Equations 3.10 and 3.11, where $\widetilde{a_{in}}$ and $\widetilde{a_{out}}$ are the weighted means of the anomaly ratings of all influencing caller and callee operations, respectively, $max(a_{out})$ is the highest anomaly rating of the directly connected callee operations, and $tol(a)$ is an arbitrarily selected tolerance function that reaches zero for the maximum of $a = 1$.

$$influence_{\mathit{callee}} := \begin{cases} decrease_{\mathit{callee}}, & \widetilde{a_{out}} > a + tol(a) \\ increase_{\mathit{callee}}, & max(a_{out}) < a \\ 0, & else \end{cases} \tag{3.10}$$

$$influence_{\mathit{caller}} := \begin{cases} decrease_{\mathit{caller}}, & \widetilde{a_{in}} < a \\ increase_{\mathit{caller}}, & \widetilde{a_{in}} \geq a \end{cases} \tag{3.11}$$

$$tol(a) := \frac{1-a}{4} \tag{3.12}$$

First, if the *mean* of the callees’ anomaly ratings is relatively high, then the cause rating of the currently viewed operation is decreased, because the cause is more likely to be there, not here. Second, if the *maximum* of the callees’ anomaly ratings is relatively small, then the cause rating is increased, because the cause is more likely to be located near the current operation. Third, if the mean of the callers’ anomaly ratings is relatively small, then the cause rating is decreased, because if the current operation had been the cause, that mean would probably be greater. Fourth, if the mean of the callers’ anomaly ratings is relatively great, then the cause rating is increased, because they probably have a common reason, and that would be the current operation. But if there is only one directly connected operation, this increase is very small, because the cause might as well be contained in that other operation. Other configurations cannot be drawn any

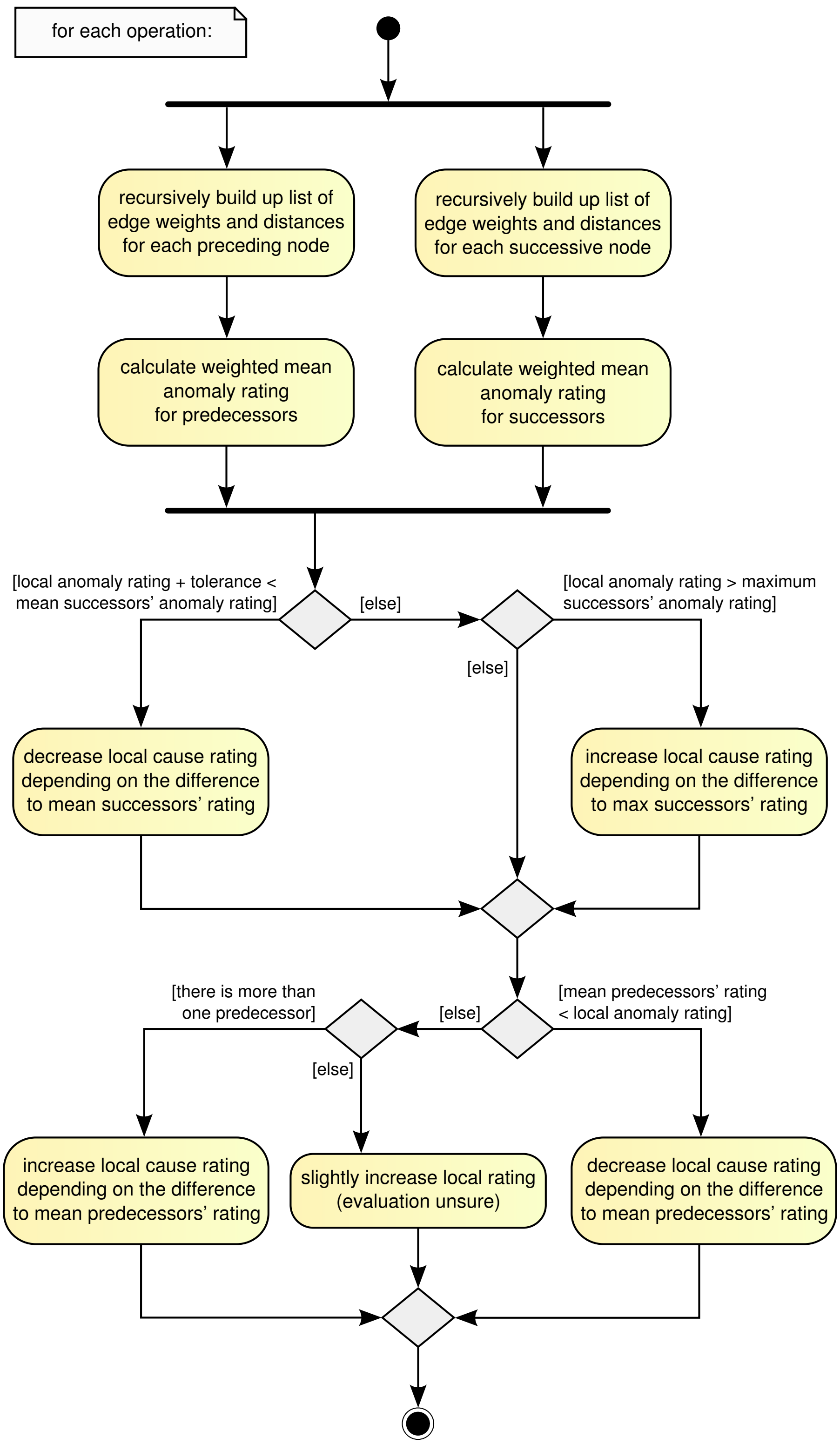

for each operation:
recursively build up list of edge weights and distances for each preceding node
recursively build up list of edge weights and distances for each successive node
calculate weighted mean anomaly rating for predecessors
calculate weighted mean anomaly rating for successors
[local anomaly rating + tolerance < mean successors' anomaly rating]
[else]
[local anomaly rating > maximum successors' anomaly rating]
[else]
decrease local cause rating depending on the difference to mean successors' rating
increase local cause rating depending on the difference to max successors' rating
[there is more than one predecessor]
[else]
[mean predecessors' rating < local anomaly rating]
[else]
increase local cause rating depending on the difference to mean predecessors' rating
slightly increase local rating (evaluation unsure)
decrease local cause rating depending on the difference to mean predecessors' rating

Figure 3.12: Anomaly correlation on operation level. After the weight-and-distance lists are build up and the means are calculated, the data is tested for several configurations that can have influence on the cause rating.

conclusions from, or they must be handled by the algorithm in another processing step, that may be on another hierarchy level, or from another operation's perspective.

Equations 3.13–3.16 show the detailed *actions* that happen, where $count(a_{in})$ is the number of directly connected caller operations, and $relation_{inOut}$ is the configurable relation factor between the influence from caller and callee operations

$$decrease_{callee} := -inh((1+a) \cdot relation_{inOut} \cdot (\widetilde{a_{out}} - a)) \tag{3.13}$$

$$increase_{callee} := \;\; inh((1-a) \cdot relation_{inOut} \cdot (a - max(a_{out}))) \tag{3.14}$$

$$decrease_{caller} := -inh((1+a) \cdot (a - \widetilde{a_{in}})) \tag{3.15}$$

$$increase_{caller} := \begin{cases} inh((1-a) \cdot (\widetilde{a_{in}} - a) \cdot 0.1), & count(a_{in}) = 1 \\ inh((1-a) \cdot (\widetilde{a_{in}} - a)), & count(a_{in}) > 1 \\ 0, & else \end{cases} \tag{3.16}$$

If an increase or decrease of the rating is decided, the amount to add to, or subtract from the rating is determined by the inhibition function that is shown in Equation 3.17. This is a normalized form of the Michaelis-Menten equation $\frac{v_{max} \cdot v}{K+v}$ (see e.g. Voet et al. [2005]) with $v_{max} = 1$ and $K = 1$ that is commonly used in biochemistry to describe the growth under a limiting factor. It realizes a monotonically increasing curve that grows approximately linear at $v = 0$ and asymptotically reaches the capacity K for large values, thus providing an upper bound (or saturation) at $K = 1$. Specifically, it reaches $\frac{2}{3}$ for $v = 2$ which is the maximum that occurs in correlation. The behavior is similar (but easier to calculate) to the positive branch of the logistic sigmoid function (Equation 3.18) that is also used in ecology and in neural networks as proposed for example by Mitchell [1997, pg. 97]. Another (positive branch of a) sigmoid function could be used in the inhibition's place with few effort.

Both the inhibition and the logistic sigmoid function are shown in Figure 3.13. In combination with the limitation factor $(1 + a)$ (that approaches zero for small values in $[-1, 1]$) resp. $(1 - a)$ (that approaches zero for large values in $[-1, 1]$), the inhibition is used in correlation to produce moderated results that again stay in $[-1, 1] \in \mathbb{R}$.

$$inh(v) := \frac{v}{1+v} \tag{3.17}$$

$$sig(t) := \frac{1}{1+e^{-t}} \tag{3.18}$$

Figure 3.14 displays the results of the correlation under the four main conditions, and for a selection of parameters, ignoring the relation factor ($relation_{inOut} := 1$). The set of curves in the top-left graph shows the condition "increase the rating while the rating is smaller than the reference value": The increase of the rating is calculated relatively high if the difference to the compared mean $\widetilde{a_{in}}$ is large, and lowers sigmoid-like with

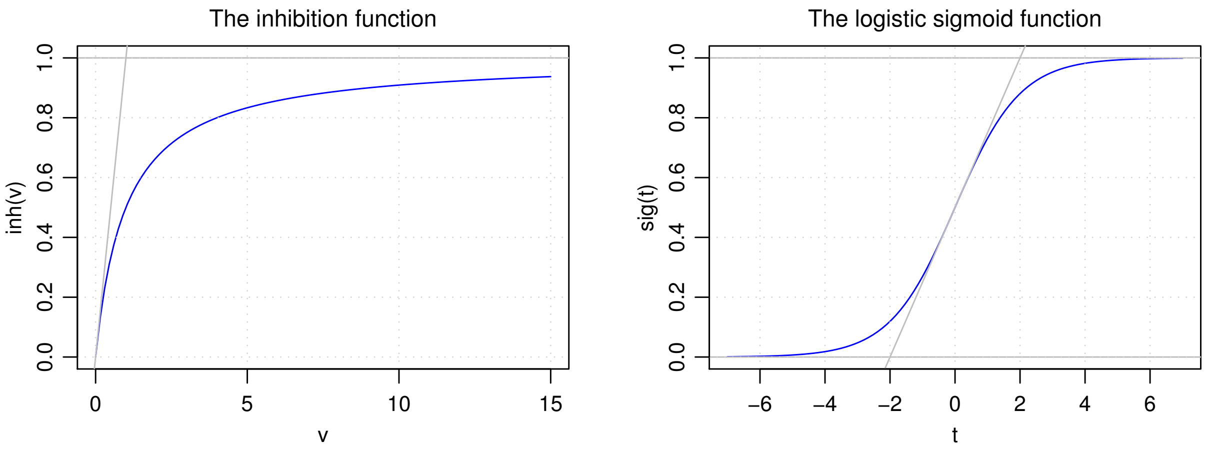


Figure 3.13: The inhibition and the logistic sigmoid function. Both curves approach a linear function at zero, and reach one for large values. The asymptotes are plotted in gray.

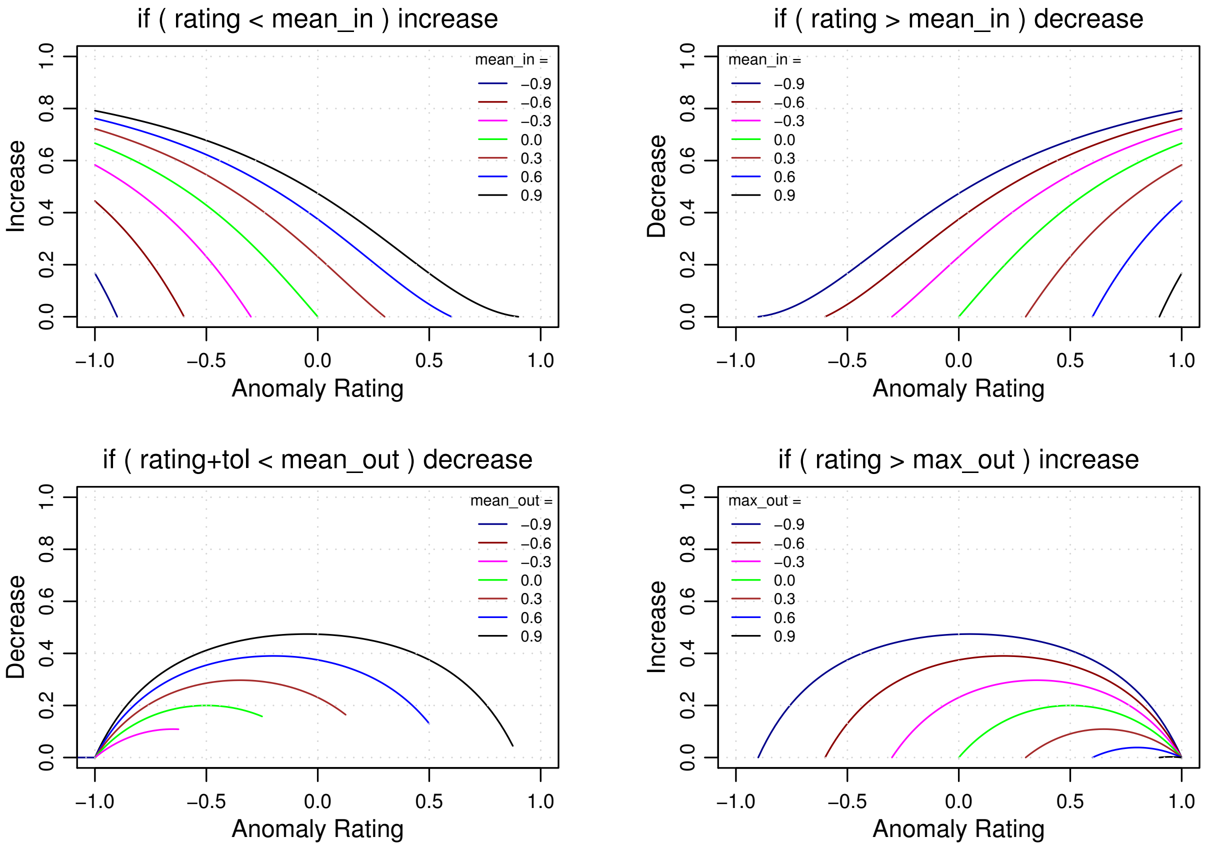


Figure 3.14: Examples of the increase and decrease functions in advanced correlation. Under pre-defined conditions, the cause rating is determined based on the anomaly rating and other factors according to Equations 3.13–3.16.

decreasing difference to not excess the maximum valid rating of 1. Likewise, the curves in the top-right graph show the decrease of the rating if it is greater than the mean, with respect to the lower limit of -1.

In contrast, if a decrease is planned while the rating is known to be relatively small already, the curves become parabolas opening to the bottom, vertically compressed by the inhibition function. This can be seen in the bottom-left graph, where one minimum is approached for a small difference between rating and mean, and the other minimum is approached for a small difference between the rating and its lower limit of -1. Additionally, this condition includes a dynamic tolerance value (given through $\frac{1-a}{4}$) that reflects in the white space where no action is taken. Finally, the set of curves in the bottom-right graph show the equivalent for a rating to be increased that is known to be relatively large already, so the increase has a minimum where the difference between the rating and the upper limit of 1 approaches zero.

3.6 Results

The Correlator plug-in provides several possibilities to fetch the results of the processing – textual and graphical. First, intermediate results and conclusions as well as warnings and errors are printed to the *console interface* at runtime. These messages can also be appended to a *log file*. The detailedness of the messages can be configured in four levels via Java `Properties`. An example of this output is shown in Figure 3.15 for verbosity level 3 (“info”). Other methods allow to write an ASCII-formatted table, or a list of components, ordered by their level of suspiciousness, to a `PrintStream` of the user’s choice as documented in the corresponding `Javadoc` comments. For custom post-processing, three methods are provided that fetch ordered `Lists` of operations, components, and deployment contexts, respectively. The Java method names of all the output functions are included in the class diagram in Figure 3.16 as members of the `CorrelatorPlugin` class.

3.6.1 Visualization

As for the graphical output, a representation of the dependency structure, optionally augmented with anomaly and cause rating details, can be converted to Graphviz *dot* files that can be used to create vector- or pixel-based image files. Dot files are simple text files that contain instructions to build up a directed graph from nodes, edges, and clusters, that can be highly configured [Gansner et al., 2006]. Instead of absolute positions, only the relations are specified. The dot program then creates a directed graph from this that can be written to common formats like PostScript (PS), Portable Network Graphics (PNG), or Scalable Vector Graphics (SVG). Our application elements’ dependencies can easily be displayed as a hierarchical box-and-line-diagram, while the anomaly ratings and cause estimations are represented by colors and textual annotations .

Figure 3.17 shows an example of a graph created by dot. Small variations in the arrangement of the elements within the dot graphs arise from the Graphviz algorithms

```
< Running experiment "20080724-050347-test"
  Properties loaded: plugin.correlator.properties
  Properties loaded: plugin.correlator.presentation.properties
  Override properties accepted.
  Debug and log file initialized.
 Building dependency structure of 40599 message traces ...
  ... found 5 deployment contexts, 16 components, and 35 operations.
 Integrating 277514 evaluated executions ...
  ... found 165242 anomalies ( = 60% ), and 0 outliers ( = 0% ).
 Algorithm initialized: "AlgorithmAdvanced".
 Rating anomalies on operation level (34 operations) ...
   Clearness: 0.3946871123507705
  ... rated sqlmapdao.ItemSqlMapDao.getItemListByProduct(String,int,int) to have the highest anomaly rating of 0,993.
 Rating cause on component level (15 components) ...
   Clearness: 0.5284704309277314
  ... rated sqlmapdao.ItemSqlMapDao to contain the cause with 8,89% probability.
 Rating cause on deployment context level (4 contexts) ...
   Clearness: 1.1076300901979153
  ... rated Virtual Machine 'tier' to contain the cause with 29,50% probability.
  ==================
  Correlator Result:
  ==================
  Application: "Experiment 20080724-050347-test" -- anomalies correlated by "AlgorithmAdvanced"
  ---------------------------
  Deployment Context: "Virtual Machine 'tier'"
    8,89%  Component: "com.ibatis.jpetstore.persistence.sqlmapdao.ItemSqlMapDao"
             [ 4693/ 8360 | +0,382 | +0,250 ]   getItem(String)
             [ 8985/ 8985 | +0,993 | +0,634 ]   getItemListByProduct(String,int,int)
    7,78%  Component: "com.ibatis.jpetstore.service.hessian.server.CatalogService"
             [ 4649/11103 | +0,366 | +0,174 ]   getCategory(String)
             [ 3935/ 8985 | +0,441 | +0,198 ]   getProduct(String)
             [ 5766/12219 | +0,148 | +0,145 ]   getProductListByCategory(String,int,int)
             [ 8985/ 8985 | +0,982 | +0,477 ]   getItemListByProduct(String,int,int)
    ...  < shortened due to space restrictions >
  ---------------------------
  Deployment Context: "Virtual Machine 'scooter'"
    6,03%  Component: "com.ibatis.jpetstore.persistence.sqlmapdao.AccountSqlMapDao"
             [  497/ 1116 | -0,272 | -0,222 ]   getAccount(String,String)
    6,30%  Component: "com.ibatis.jpetstore.service.hessian.server.AccountService"
             [  518/ 1116 | -0,202 | -0,080 ]   getAccount(String,String)
  ==================
  Components sorted by cause rating in descending order:
  ==================
     8,89%  com.ibatis.jpetstore.persistence.sqlmapdao.ItemSqlMapDao
     7,78%  com.ibatis.jpetstore.service.hessian.server.CatalogService
     6,91%  com.ibatis.jpetstore.persistence.sqlmapdao.ProductSqlMapDao
     6,83%  com.ibatis.jpetstore.presentation.OrderBean
     6,71%  org.apache.struts.action.ActionServlet
      ...  < shortened due to space restrictions >
     6,03%  com.ibatis.jpetstore.persistence.sqlmapdao.AccountSqlMapDao
  ==================
< Checking correlator consistency ... done. Plug-in finished without error.
< Creating file /home/nina/logAnalysis/tempdata/correlator/demo-080724.dot ... done.
< Creating file /home/nina/logAnalysis/tempdata/correlator/demo-080724.dot.ps ... done.
< Creating file /home/nina/logAnalysis/tempdata/correlator/demo-080724.dot.svg ... done.
< Creating file /home/nina/logAnalysis/tempdata/correlator/demo-080724.dot.png ... done.
```

Figure 3.15: Example of the textual output of the Correlator's results, to be found on command line interface, and in log files, for verbosity level 3 ("info"). The listing has been carefully shortened due to space restrictions.

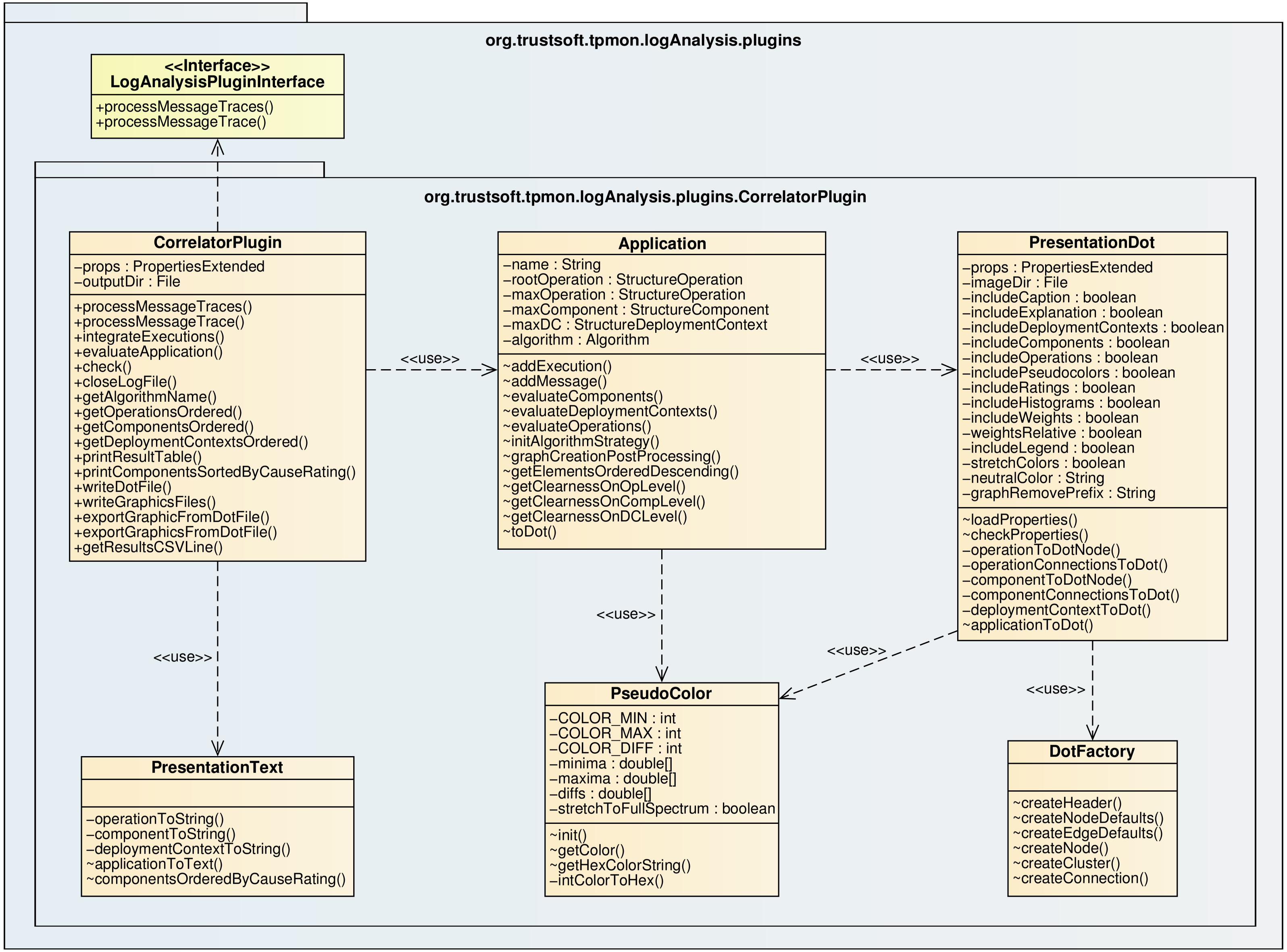


Figure 3.16: Class diagram focused on the presentation classes.

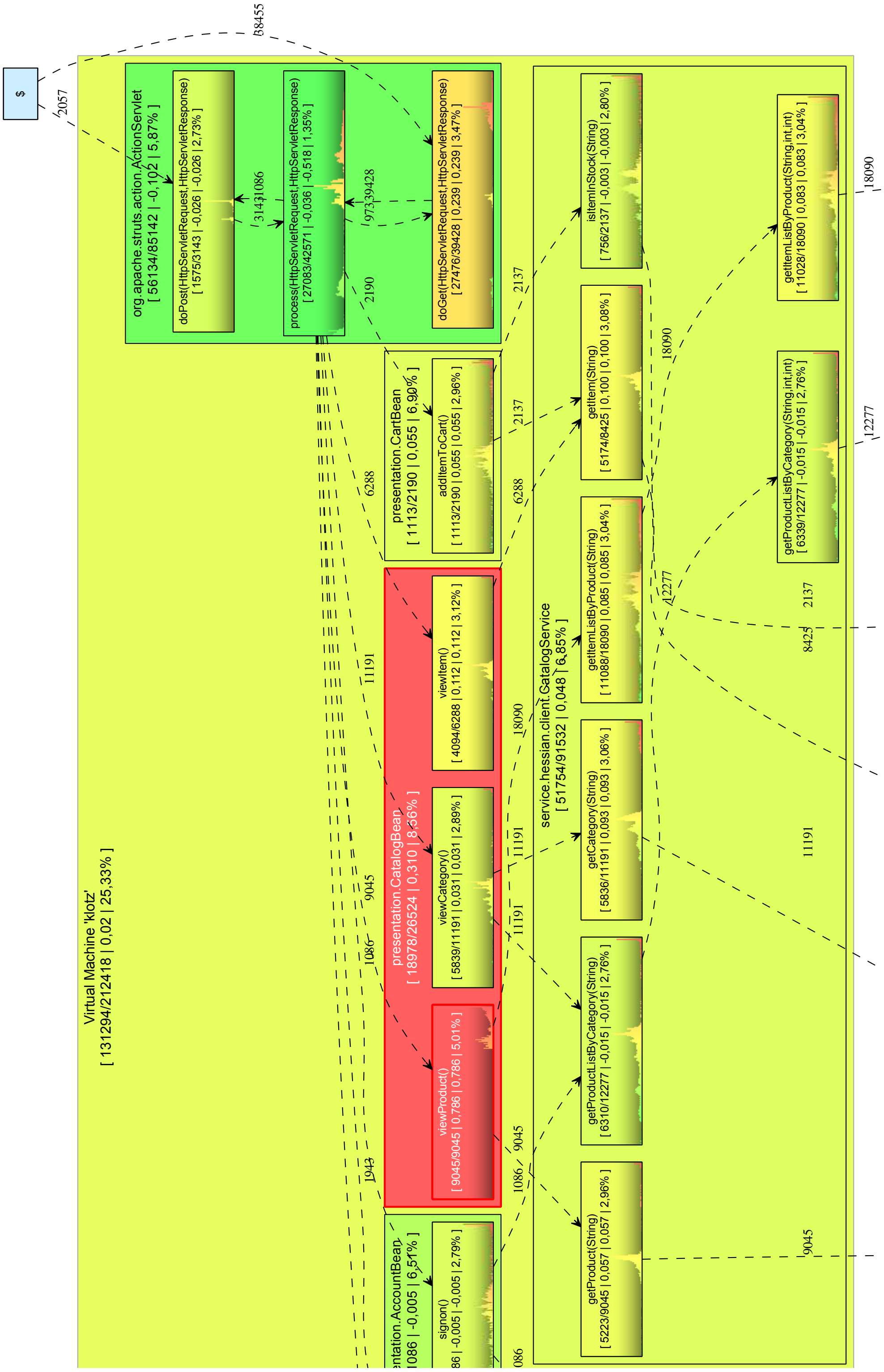


Figure 3.17: Example of the graphical result of the Correlator. Due to space restrictions, some parts of the graph are omitted. There are four components visible in one of the four deployment context. The percent values, augmented with colors, represent the probability for the elements to be the cause of failure.

and can not be controlled by Tpan or the Correlator plug-in. The backbone of the graph is the net of operations. The operations are connected among each other, based on their calling dependencies, and grouped into components and deployment contexts. Each element is labeled with a distinct name, and optionally annotated with debugging information about its executions and the individual ratings. As usual in Tpan, the "root operation" is represented by the Dollar symbol $ – this is not a real application element, but denotes the starting point for all execution paths, that may be a human user, or a workload driver. The connections are annotated with the number or percentage of method interactions on their respective paths. The ratings are expressed through colors, shaded from green (normal) over yellow (ambiguous) to red (anomaly). The highest rating is highlighted by special coloring by default. The percentage values are accumulated to 100% on each hierarchy level. Although executions are organized in the internal structure very similarly to the other elements, it is not feasible to visualize them in the same way. Instead, they are depicted as anomaly histograms as already shown separately in Figure 3.9 on page 32. Technically, the histograms are independent graphic files, usually in PNG format, that are linked into the dot code to be included as background images for graph elements.

All features can be configured in detail via Java `Properties`. The three hierarchy levels, the histograms, the rating details, the edge weights, as well as additional graph elements such as a caption, an explanatory text, and a legend can all be individually switched on or off. The remaining graph adapts to some changes. For example, if the operations are hidden, the components are connected with each other, and show a histogram each. Additional options include the adjustment of font families, colors, and sizes for different graph elements, as well as histogram dimensions and image types.

3.6.2 Automation

Beyond the ability to be configured through Java `Properties`, the Correlator plug-in has two features to allow the automated control of experiment evaluation. This way, the automated evaluation of a dataset with different parameter sets is supported without having to restart Tpan, or reload any data.

Parameter Override

The plug-in comes with the `Experiment` class that manages a set of experiment parameters that are supposed to override those specified in the default `Properties` files. There is no separation between `Properties` for the plug-in, for correlation, and for presentation. Additionally, in case a fault has intentionally been injected into the application under analysis – e.g. for evaluating the Correlator itself –, information about the level and the position of the fault may be specified that is later used to rate the success of the analysis.

```
# This is an experiment batch file for the Correlator plug-in.
# Layout: experiment name; property=value [; property=value] ...

20080612-001; correlatorAlgorithm = AlgorithmSimple; exportFileTypes ↩
    = ps:svg:png; debugLevel = 3
20080612-002; injectionLevel = comp; injectionElement = ↩
    com.ibatis.jpetstore.presentation.CatalogBean
20080612-003; graphIncludeLegend=0; graphFileName=graph; ↩
    graphStretchPseudocolorsToFullSpectrum=0

###   <-  this is an EOF marker

20080612-004; correlatorAlgorithm = AlgorithmAdvanced; ↩
    operationAggregationPowerMeanExponent = 2.0
```

Figure 3.18: Example of an experiment batch control file. A set of `Properties` can be specified per experiment to override the default `Properties`.

The `Experiments` can be created through a distinct CSV file, specifying one experiment per line. Alternatively, a single `Experiment` can be specified in an entry in the Tpan `Properties` file. Figure 3.18 shows an example of an experiment batch control file. The lines are processed successively, running an independent correlation analysis each.

Results Table

In addition to the log file, and the dot graph exports, a text file can be written that contains a summary of the analysis performed on an `Experiment` group in CSV format. By default, it contains the experiment name, and clearness ratings for each hierarchy level. Outside of the plug-in code, a success rating is generated if the real fault's position has been specified. Further information can be added easily when needed.

Chapter 4

Evaluation

To evaluate the applicability of the approach presented in Chapter 3 in practice, an *explorative case study* is performed. By analyzing and interpreting the output of the Correlator for a series of *scenarios*, we want to learn about the plausibility of its estimation, and maybe discover possibilities for further improvements in the whole process from monitoring over anomaly detection to event correlation when applied to distributed systems. In contrast to a *controlled experiment* as described by Koziolek [2005, p. 2], the input data is not strictly under control, but determined by a probabilistic process.

We choose the iBATIS JPetStore[1], which is an implementation of the Sun Java Pet Store Reference Application[2], a sample online shopping store, to be the *application under analysis* (AUA). It has been used in our working group for similar examinations before, so its architecture is well known, and workload models as well as monitoring strategies have already been developed and applied [Focke, 2006; Stöver, 2007; Rohr et al., 2008b]. To match the demand for a distributed Java Web application, the JPetStore has recently been divided into several parts [Rohr et al., 2008a].

The significance of the correlation function is evaluated by using reasonably realistic but also randomly generated behavior of sample users. A single fault per scenario is injected at selected points, that could be e.g. a source code bug, or a hardware misconfiguration. Ideally the output exactly pinpoints the cause.

After a short explanation of the goals and metrics in Section 4.1, an overview of the experiment setup is given in Section 4.2, covering the monitoring, and the workload generation. Section 4.3 describes the selected methods of fault injection in detail, while Section 4.4 documents the experiment activities to gain timing data for the following examinations. In Section 4.5, the analysis of five fault injection scenarios is presented that are examined with different algorithms developed in the previous chapter, combined with various settings and parameters. Finally, Section 4.6 gives a summary of the examination results and an impression of performance of the analysis.

[1] `http://ibatis.apache.org`

[2] `http://java.sun.com/developer/releases/petstore/`

4.1 Goals and Metrics

The main goal of the evaluation is to find out whether the Correlator plug-in developed in Sections 3.3–3.5 works as desired. It should assign a high cause probability to the application element where the fault has actually been injected. The plug-in should also show no erroneous behavior itself, though the tolerance to faulty input data is not explicitly tested.

The secondary objective is the examination of the algorithm variants and parameters of the analysis. It is expected that the *advanced algorithm* performs better than the *trivial algorithm*. Parameters like mean calculation method, edge weight calculation method, and distance intensity have to be systematically tested for different fault injection scenarios whether, and to what extent they affect the quality of the analysis.

The application hierarchy level of the fault injection is relevant in that the result will be at the same level. For example, if a fault is injected on component level, a meaningful result on operation level cannot be expected.

In order to evaluate the quality of the analysis, first the *clearness* is calculated for each level, based on the rank of the elements, ordered by their rating to be the cause of failure, and based on the relation between consecutive ranks. Provided a list of cause ratings, translated to percentages, and sorted in descending order, the clearness function $clear \in \mathbb{R}^+$ is given through Equation 4.1: It increases with increasing contrast of the first value to the other values, and decreases with increasing similarity of the values.

$$clear(r_{1,\ldots,n}) := \frac{r_1}{\sum\limits_{i=2}^{n} \frac{r_i}{i} + 1} \tag{4.1}$$

For example, provided the ordered ratings vector $\langle 40, 35, 25 \rangle$, the benchmark is given through $clear(\langle 40, 35, 25 \rangle) = \frac{40}{\frac{35}{2}+\frac{25}{3}+1} = 1.49$ which is not very clear compared to $clear(\langle 60, 30, 10 \rangle) = \frac{60}{\frac{30}{2}+\frac{10}{3}+1} = 3.10$. The results of *clear* are comparable to each other, independent of the number of ratings, i.e. the ratings vector length, as can be seen in $clear(\langle 60, 30, 5, 1, 1, 1, 1, 1 \rangle) = 3.23$. Furthermore, the ratings neither need be percent values, nor need to sum up to 100. Instead, the values can be in any range, since their relations are calculated only.

The distribution of the clearness may also give an indication of the estimated fault's hierarchy level: For instance, if the clearness on component level is significantly higher than the clearness on operation and deployment context level, then this indicates that on component level, there is probably one element that is more emphasized compared to its sibling elements, while on the other levels, the analysis is not that clear.

The indicator for the *success* of the analysis among different experiments is given through the quotient of the rating (again translated to percentages) of the element where the fault has actually been injected, and the highest rating on the appropriate level, as shown in Equation 4.2 where $r^* \in \{r_1, \ldots, r_n\}$ is the "true" fault's element's rating. Thus, the result of the *success* function is a decimal in $[0, 1]$, reaching 100% if r^* is equal to the highest rating.

$$success(\{r_1, \ldots, r_n\}) := \frac{r^*}{max(\{r_1, \ldots, r_n\})} \tag{4.2}$$

4.2 Experiment Setup

Figure 4.1 shows all elements that contribute to the experiment setup, and their relations as well as sketches of their respective output.

- In the center is the JPetStore, our application under analysis (AUA), that has been distributed over four machines plus one machine for the database. With the use of AOP as described in Section 2.2 on page 9, monitoring points are woven into the components, and connected to the monitoring tool *Tpmon* [Rohr et al., 2008b] that writes monitoring data to the file system of the respective host.

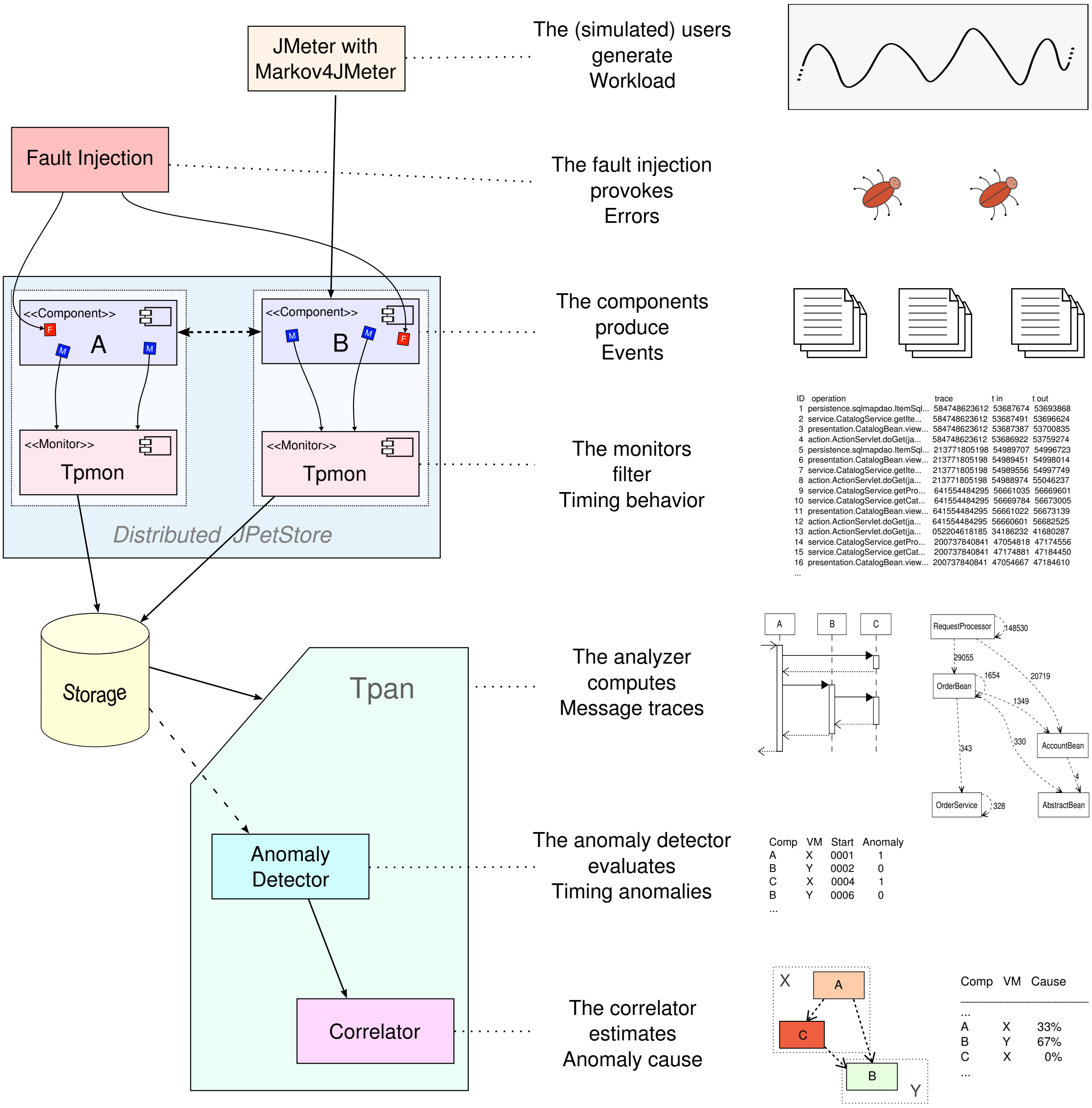
- In contrast to Kiciman and Fox [2005], who explicitly run deterministic workload to verify their results with MD5 hashes from fault-free runs, we want to expose the AUA to probabilistic workload that can be generated by Apache *JMeter*[3] with the extension *Markov4JMeter* [van Hoorn et al., 2008].

- The fault injection is not a software component, but a set of selected manipulations that provoke failures of different complexity in the AUA. Scenarios are created, each containing one fault of a certain kind on a specified architecture level. For example, there are programming faults on operation level, and hardware faults on deployment context level.

- *Tpan* [Rohr et al., 2008b], introduced in Section 2.1 on page 7, analyzes the monitoring data through its plug-ins. The system behavior can be reconstructed in terms of several graphs and diagrams. The component dependencies are essential for the further processing.

- *PAD* [Rohr et al., 2008a] (Plain Anomaly Detector), one of the plug-ins, also reads the monitoring data, but without considering dependencies, it analyzes the timing behavior and determines anomalies within.

- The remaining step is for the *Correlator* plug-in developed in Chapter 3 to combine the gained information about dependencies and timing behavior anomalies, and to make an estimation about the cause of each failure, whose quality will be rated in the experiment to improve the correlation algorithm.

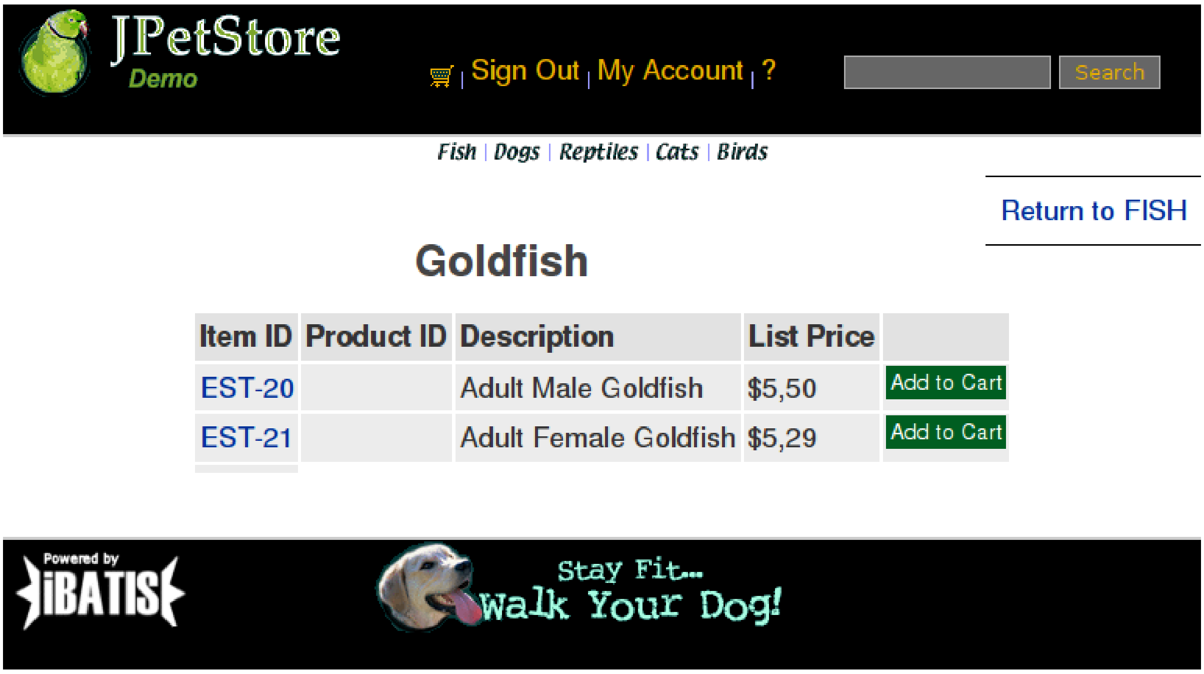

JMeter with Markov4JMeter
The (simulated) users generate Workload
Fault Injection
The fault injection provokes Errors
<<Component>>
A
F
M
M
<<Component>>
B
M
M
F
The components produce Events
<<Monitor>>
Tpmon
<<Monitor>>
Tpmon
The monitors filter Timing behavior
Distributed JPetStore
Storage
Tpan
The analyzer computes Message traces
A
B
C
RequestProcessor
148530
29055
20719
1654
OrderBean
1349
330
AccountBean
343
4
OrderService
328
AbstractBean
Anomaly Detector
The anomaly detector evaluates Timing anomalies
Correlator
The correlator estimates Anomaly cause
X
A
C
B
Y

ID	operation	trace	t in	t out
1	persistence.sqlmapdao.ItemSql...	584748623612	53687674	53693868
2	service.CatalogService.getIte...	584748623612	53687491	53696624
3	presentation.CatalogBean.view...	584748623612	53687387	53700835
4	action.ActionServlet.doGet(ja...	584748623612	53686922	53759274
5	persistence.sqlmapdao.ItemSql...	213771805198	54989707	54996723
6	presentation.CatalogBean.view...	213771805198	54989451	54998014
7	service.CatalogService.getIte...	213771805198	54989556	54997749
8	action.ActionServlet.doGet(ja...	213771805198	54988974	55046237
9	service.CatalogService.getPro...	641554484295	56661035	56669601
10	service.CatalogService.getCat...	641554484295	56669784	56673005
11	presentation.CatalogBean.view...	641554484295	56661022	56673139
12	action.ActionServlet.doGet(ja...	641554484295	56660601	56682525
13	action.ActionServlet.doGet(ja...	052204618185	34186232	41680287
14	service.CatalogService.getPro...	200737840841	47054818	47174556
15	service.CatalogService.getCat...	200737840841	47174881	47184450
16	presentation.CatalogBean.view...	200737840841	47054667	47184610
...				

Comp	VM	Start	Anomaly
A	X	0001	1
B	Y	0002	0
C	X	0004	1
B	Y	0006	0
...			

Comp	VM	Cause
...		
A	X	33%
B	Y	67%
C	X	0%
...		

Figure 4.1: Conceptual overview of the experiment setup. The application under analysis is exposed to simulated workload, and fault injection. Tpmon monitors the timing behavior, which is then evaluated by Tpan with an anomaly detector, and the Correlator plug-in.



Goldfish

Item ID	Product ID	Description	List Price	
EST-20		Adult Male Goldfish	$5,50	Add to Cart
EST-21		Adult Female Goldfish	$5,29	Add to Cart

Figure 4.2: iBATIS JPetStore 5 demo application. Screenshot of the catalog view in a Web browser.

4.2.1 Application under Analysis

The JPetStore is an open source J2EE application that runs in a Servlet Container – we use Apache Tomcat. As being an example of a common Web shop, its only user interface is through an HTML browser as shown in the screenshot in Figure 4.2. It allows to manage a personal account, to browse a catalog of items, to add items to a virtual shopping cart, and to simulate a purchasing process. The account and catalog data is persistently stored into a database management system – we use MySQL. The JPetStore has a classic three-tier architecture: presentation, application, database, as described in Section 2.4.

In order to get a more realistic scenario, the AUA has been divided into four parts. The distribution is mainly influenced by the application services that are grouped into four categories: account, cart, catalog, and order. We analyzed some monitored profiles of example usage as well as the dependency structure, and decided to deploy the application to five machines as depicted in Figure 4.3, each running an Apache Tomcat Servlet container: One for account service and catalog service each, one for cart and order combined – these three represent the *application* layer –, one for the database, and another machine for the presentation layer. A sixth machine contains the controlling system that hosts the workload generator, and stores the results.

[3] http://jakarta.apache.org/jmeter/

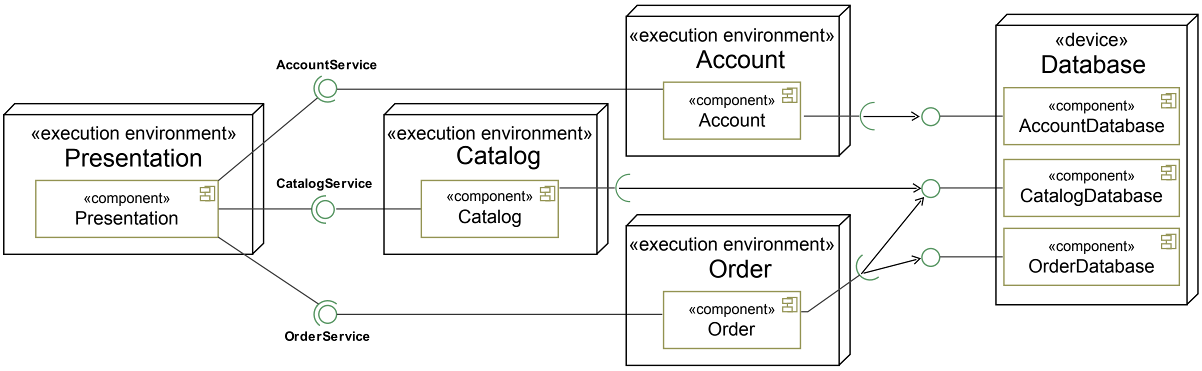


Figure 4.3: Deployment of the JPetStore components [Rohr et al., 2008a]. Referring to the first concept shown in Figure 1.3 on page 5, the functionality of the `Cart` has been merged into the `Order` context.

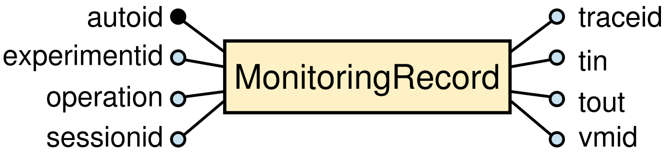


Figure 4.4: Database schema for the monitoring of executions [Rohr et al., 2008b].

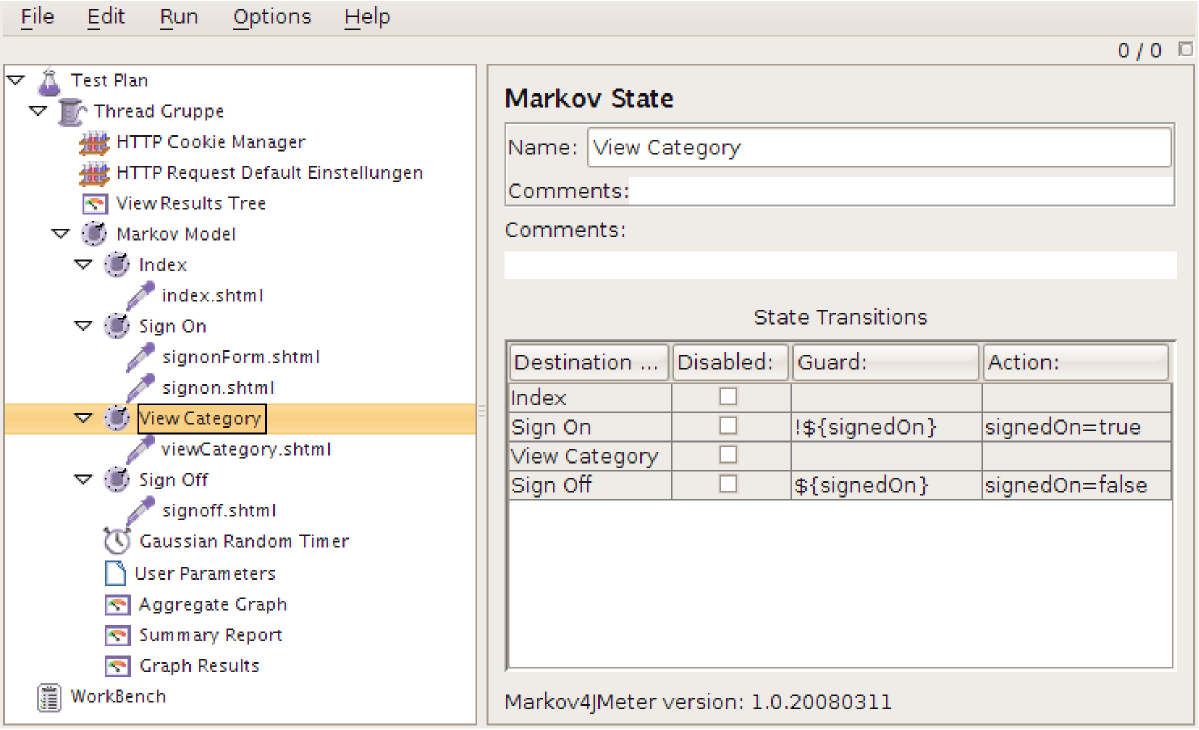
4.2.2 Monitoring

The AUA is instrumented with monitoring probes provided by Kieker [Rohr et al., 2008b] as described in Section 2.2. The start and exit timestamps of a subset of the Java method executions are recorded. Figure 4.4 shows the database schema for the monitoring data. The choice of methods is adopted from Rohr et al. [2008a] who “carefully determined” a subset of 34 operations that are listed in Section A.2. In fact, more than 50 operations are instrumented, but some of them turn out to be never called during the experiments.

4.2.3 Workload Generation

JMeter is an open source Java tool to execute load tests for functional and performance evaluation mainly for Web applications. Many concurrent users can be simulated by simultaneous threads. Individual workload profiles can be configured in detail, and the functionality can be extended by plug-ins. In the current setup, JMeter simulates a number of users that browse the JPetStore using HTTP `get` and `post` requests. Technically, JMeter acts as an HTTP user agent from the view of the Web application. It has the ability to follow links, to complete HTML forms, and to activate specified buttons.

Markov4JMeter [van Hoorn et al., 2008] is a plug-in for JMeter that allows to define probabilistic usage profiles based on Markov chains. For example, a user behavior could be designed that, from browsing the catalog, (1) inspects an item with a probability of 40 percent, (2) selects another category with 30 percent, (3) returns to the main menu with 15 percent, (4) switches to the shopping cart with 10 percent, and (5) leaves the



Markov State

Name: View Category

Comments:

Comments:

State Transitions

Destination ...	Disabled:	Guard:	Action:
Index	☐		
Sign On	☐	!${signedOn}	signedOn=true
View Category	☐		
Sign Off	☐	${signedOn}	signedOn=false

Markov4JMeter version: 1.0.20080311

Figure 4.5: Apache JMeter 2.3 with Markov4JMeter 1.0. Screenshot of the state transition editor.

shop with a probability of 5 percent. The time that a user remains in a certain state can be configured through a timing function that may also be probabilistic. Additionally, the number of active concurrent users can be varied over time using another function. Parts of both JMeter and Markov4JMeter can be seen in the screenshot in Figure 4.5.

Since the whole anomaly detection process is highly experimental and under current work, care must be taken to select a workload curve that produces reasonable data so that the analysis gets significant results. For the following experiments, a constant workload is used that uses up to 15 threads, far less than the overall system capacity of about 80 threads as observed in preparative experiment runs. Two different user profiles are used concurrently: A “browser” that is mostly browsing the catalog and viewing the products of the store without purchasing anything, and a “buyer” that actually logs in, adds items to the shopping cart, and completes a buying process.

4.3 Fault Injection

To put the whole fault localization approach to the test, a set of manipulations is systematically applied to the AUA that are or that simulate faults on different levels, and of various character. Since it is obviously impossible to construct and experiment with *all* theoretically possible situations, some representatives have to be selected, and the results will be statistically evaluated. Avižienis et al. [2004, pg. 5] list a comprehensive taxonomy of faults, grouped in development faults, physical faults, and interaction faults. Our experiments shall cover all of these. The fault injection is focused on software implementation, because, as Kiciman and Fox [2005, p. 7] state, hardware and low level operating system faults typically do not affect the application level without being noticed otherwise. However, one experiment is performed that simulates a broken CPU cooling system.

These are the requirements for the choice of faults to be injected:

1. There has to be some noticeable effect, thus a failure can be detected, either automatically or manually. For example, this effect might be bad responsiveness, or incorrect HTML output, that a user can notice, and report to the administrator.

2. It should be an effect that typically does not spawn a direct administrative message, as is assumed for hardware outages, or software exceptions.

3. There should be a diversity in the position of the fault in the dependency structure, because of the assumption that faults propagate from the bottom to the top, against the direction of the dependencies.

4. All hierarchy levels – operations, components, and deployment contexts – shall be tested individually.

5. For the nature of the influence on the timing behavior, “realistic” faults shall be used as well as exact and repeatable values.

6. The timing behavior should be influenced in both directions, either increasing or decreasing the response times of the executions.

As described by Schwenkenberg [2007, p. 14], software implemented fault injection can happen by embedding the injector to the target system, or by adding a software layer between application and the operating system. He also surveys tools that test the stability of processes by stressing them with random input values, or by intentionally violating communication protocol specifications. Other approaches base on the injection of faults directly into the memory used by running processes. Virtualization and simulation offer further possibilities for fault injection while retaining the integrity of the testing environment. On the other hand, the author points out the demand of resources by injection tools, which especially affects statements about timing behavior.

Since we want to influence the AUA in detail, and we monitor timing behavior, we decided to *not* use any general-purpose tools for fault injection. Instead, the following five manipulation variants are manually applied.

Programming fault	Functional effect	Effect on response times
Omission of negation symbol	Function abortion	Decreased, because method is exited earlier
Wrong method call	None	Insignificant
Boolean flip	None	Insignificant
Omission of a method call	None	Insignificant
Parameter change	None	Insignificant
Condition flip	Exception	Insignificant, but changed calling hierarchy
Loop condition placement	Exception under certain condition	Insignificant
Change of method call sequence	Action has to be initiated twice	Increased (no explanation given by the author)
Change of loop condition symbol	Exception	Decreased
If/else flip	Exception	Decreased
Assignment omission	Exception	Decreased
Condition inversion	HTML output change (item list empty)	Insignificant

Table 4.1: Summary of programming faults manually injected into the JPetStore by Schwenkenberg [2007] and their effects on functional and timing behavior.

4.3.1 Programming Faults

A large variety of source code manipulations can be injected. Kiciman and Fox [2005, p. 7] state their fault injection to "include those that a programmer building a system should expect, might expect, and likely would not expect." Java exceptions can be used to emulate a range of faults, because they cover different effects, from hardware faults to programming issues. However, not all simple programming bugs – like incorrect assignments, mistaken symbols, or exchanged calls or blocks – result in exceptions, and thus have to be regarded separately, although not all of these are expected to result in timing behavior anomalies.

In the experiments performed by Schwenkenberg [2007, p. 69], several fault variants are selected as being "realistic" programming faults, and their effects are demonstrated for one exemplary method each. Table 4.1 summarizes the results. The problem is that most of the modifications that actually cause timing behavior anomalies are also throwing an exception, which disqualifies them for the current experiments. The only modification that results in increased response times is not documented adequately. In the light of this, we decide not to use the specific examples, since they do not meet our requirements for faults that significantly influence the timing behavior while not directly leading to a clear error message as would be the case with Java exceptions.

In preparation for the fault injection experiments, to minimize the influence on the existing experiment cluster, that is also used for other research, a copy of the JPetStore has been deployed – Section A.3 on page 94 lists the technical activities.

Deployment context:	Presentation – Host `jpet2` (“klotz”)
Package:	`com.ibatis.jpetstore.presentation`
Class file:	`CatalogBean.java`, line 142.
Method:	`viewProduct()`
Modification:	`catalogService.getItemListByProduct(String);`
	is called twice, one shortly after the other.

Deployment context:	CatalogService – Host `jpet4` (“tier”)
Package:	`com.ibatis.jpetstore.persistence.sqlmapdao`
Class file:	`ItemSqlMapDao.java`, line 47
Method:	`getItemListByProduct(String,int,int)`
Modification:	`queryForList(String,Object,int,int);`
	is called twice, one directly after the other.

Deployment context:	Presentation – Host `jpet2` (“klotz”)
Package:	`com.ibatis.jpetstore.service.hessian.client`
Class file:	`CatalogService.java`, line 66
Method:	`getItemListByProduct(String,int,int)`
Modification:	`new java.util.ArrayList(0);`
	is executed instead of
	`catalogService.getItemListByProduct(String,int,int);`

Table 4.2: Source code manipulations successively applied to three points of the JPetStore. Double code execution (to increase response times) is performed on presentation and on application layer. An empty list is returned on presentation layer to decrease response time.

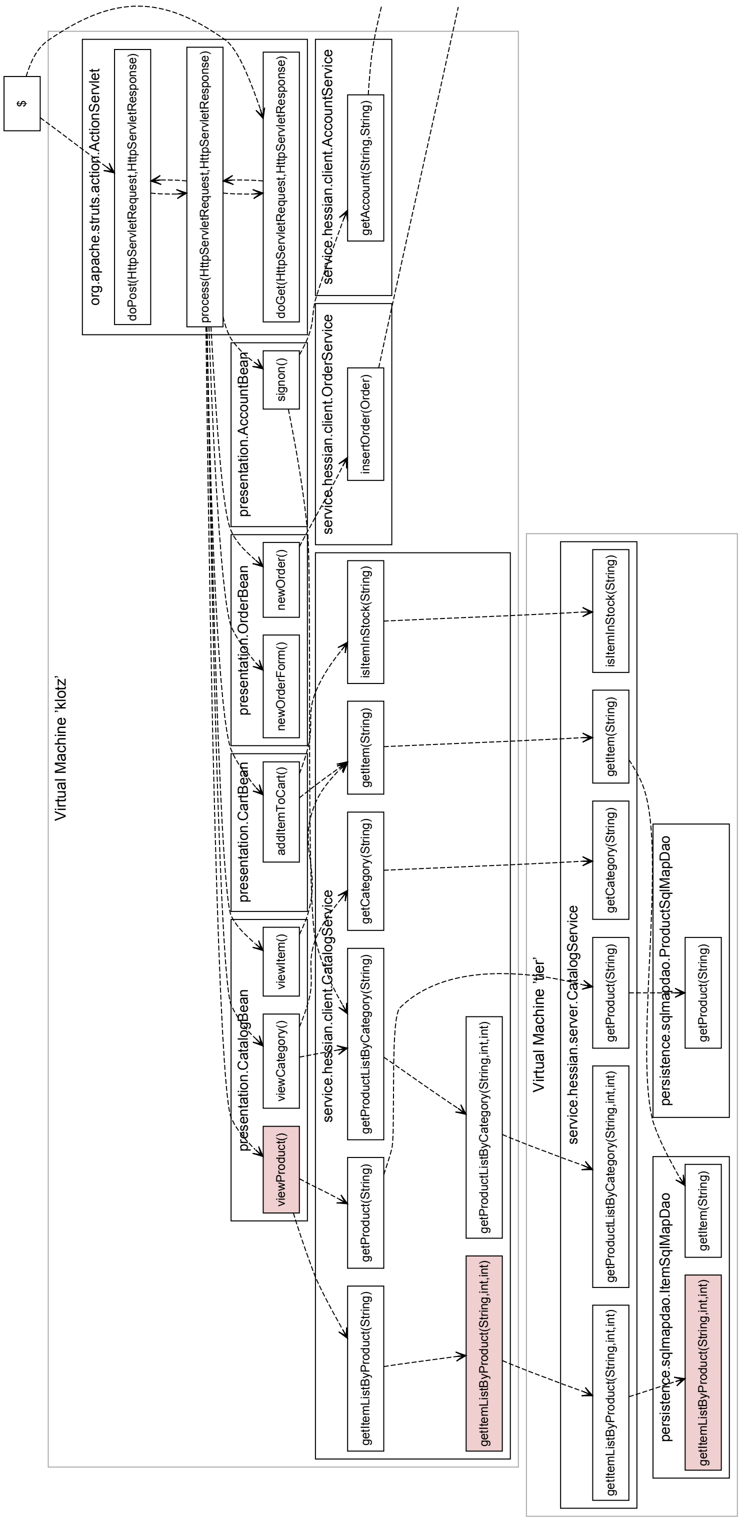

$
Virtual Machine 'klotz'
org.apache.struts.action.ActionServlet
doPost(HttpServletRequest,HttpServletResponse)
process(HttpServletRequest,HttpServletResponse)
doGet(HttpServletRequest,HttpServletResponse)
service.hessian.client.AccountService
getAccount(String,String)
service.hessian.client.OrderService
insertOrder(Order)
presentation.AccountBean
signon()
presentation.OrderBean
newOrderForm()
newOrder()
presentation.CartBean
addItemToCart()
presentation.CatalogBean
viewProduct()
viewCategory()
viewItem()
service.hessian.client.CatalogService
getItemListByProduct(String)
getProduct(String)
getProductListByCategory(String)
getCategory(String)
getItem(String)
isItemInStock(String)
getItemListByProduct(String,int,int)
getProductListByCategory(String,int,int)
Virtual Machine 'tier'
service.hessian.server.CatalogService
getItemListByProduct(String,int,int)
getProductListByCategory(String,int,int)
getProduct(String)
getCategory(String)
getItem(String)
isItemInStock(String)
persistence.sqlmapdao.ItemSqlMapDao
getItemListByProduct(String,int,int)
getItem(String)
persistence.sqlmapdao.ProductSqlMapDao
getProduct(String)

Figure 4.6: Dependency graph of the JPetStore (Part 1), automatically created by the Correlator plug-in. Due to space restrictions, the deployment contexts for `AccountService` and `OrderService` are omitted. Three operations are colored to mark the positions for the injection of programming faults.

Three source code manipulations that are detailed in Table 4.2 and depicted in Figure 4.6 are selected to emulate programming faults. They are successively applied in an own experiment each.

- Two manipulations are supposed to increase the response times by executing some non-trivial method twice. This is probably not noticed by a user. They differ in position not only in the application's architecture level, but also in the implementation: One manipulation is to execute a whole method (a one-liner) twice, the other doubles only one line of the enclosing method.

- The third manipulation aims at a decrease of the response times by instantly returning an empty list of items instead of fetching them from the database. This can be clearly recognized by a user who lists any of the product's items. While the HTML code is still correct in this point, the lists are all empty, as if no items were available at all.

These might not be typical representatives for programming faults, but (1) the JPetStore does not have much "interesting" code (much of the functionality is handled by the underlying iBATIS framework), (2) complex code manipulations to test the fault detection quality would be out of scope of this work, and (3) at least the listed modifications are non-trivial to the extent that none of them is detected by FindBugs[4] [Ayewah et al., 2007], an open source static analysis tool for Java programs, although it detects other potential bugs in the distributed JPetStore, even from the performance area.

This choice of faults fulfills the points 1, 2, 3, and 6 of the requirements listed at the beginning of Section 4.3. Due to the nature of programming faults, their location can be tracked down on *operation level* (point 4), a suitable monitoring assumed.

4.3.2 Database Connection Slowdown

To test the Correlator's capability of localizing faults at the bottom of the calling dependency structure, namely at the connection to the database, manipulations are applied at two classes that are closest to the SQL layer provided by iBATIS. In their experiments, Agarwal et al. [2004] simulate higher response times by using a special tool that periodically locks database tables. Since the complexity of such action is out of scope of the current work, as well as the setup of a high load network, we simulate a slow database connection by adding some `sleep(long)` commands to the Java code. Table 4.3 and Figure 4.7 show the details of the modifications. Compared to the injection of programming faults, concrete values for timing misbehavior are used. Being about three times the typical response times of these methods that are about 3 milliseconds, a sleep of 10 milliseconds is supposed to have a significant influence that is detected with high probability. To match the demand for a fault on *component level*, one experiment is performed with a slowdown in `persistence.sqlmapdao.ItemSqlMapDao`, which turns out to be the

[4] `http://findbugs.sourceforge.net/`

Deployment context:	AccountService – Host `jpet3` (“scooter”)
Package:	`com.ibatis.jpetstore.persistence`
Class file:	`AccountSqlMapDao.java`, line 24.
Method:	`getAccount(String,String)`
Modification:	`Thread.sleep(10);` is added before the method returns.

Deployment context:	CatalogService – Host `jpet4` (“tier”)
Package:	`com.ibatis.jpetstore.persistence.sqlmapdao`
Class file:	`ItemSqlMapDao.java`, lines 47 and 55
Method:	`getItemListByProduct(String,int,int)` and `getItem(String)`
Modification:	`Thread.sleep(10);` is added each before the methods return.

Table 4.3: Source code changes successively applied to the JPetStore to each simulate a slowdown of the database connection. A call to `Thread.sleep(long)` is performed to let the execution wait the specified number of milliseconds.

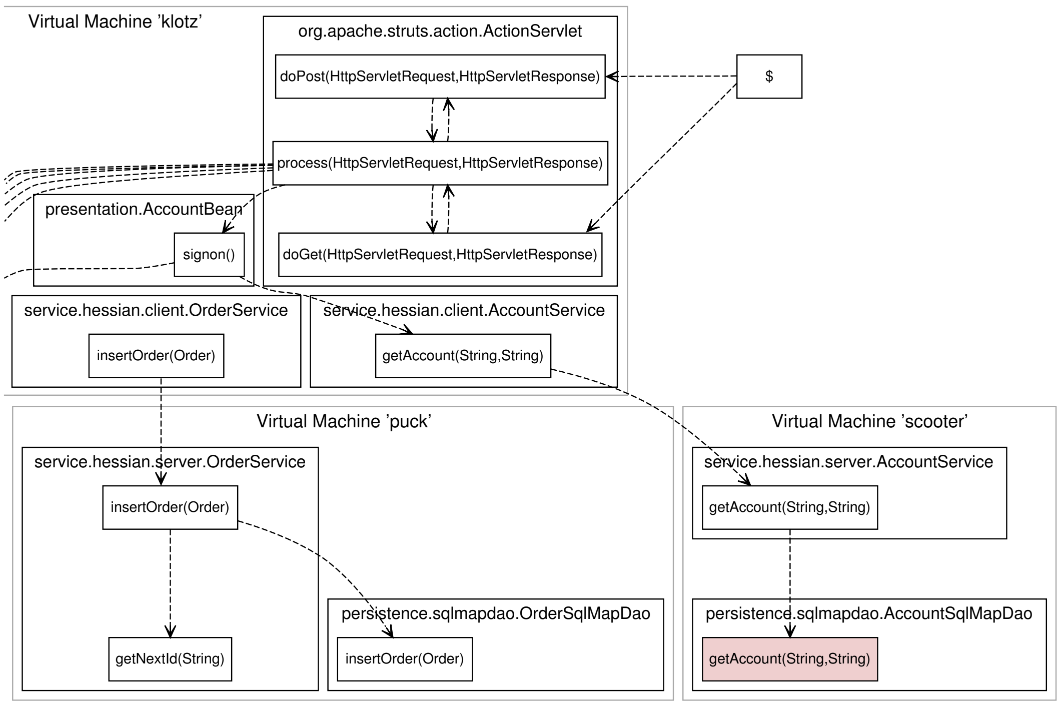

Virtual Machine 'klotz'
org.apache.struts.action.ActionServlet
doPost(HttpServletRequest,HttpServletResponse)
$
process(HttpServletRequest,HttpServletResponse)
presentation.AccountBean
signon()
doGet(HttpServletRequest,HttpServletResponse)
service.hessian.client.OrderService
insertOrder(Order)
service.hessian.client.AccountService
getAccount(String,String)
Virtual Machine 'puck'
service.hessian.server.OrderService
insertOrder(Order)
getNextId(String)
persistence.sqlmapdao.OrderSqlMapDao
insertOrder(Order)
Virtual Machine 'scooter'
service.hessian.server.AccountService
getAccount(String,String)
persistence.sqlmapdao.AccountSqlMapDao
getAccount(String,String)

Figure 4.7: Dependency graph of the JPetStore (Part 2). Due to space restrictions, some parts are omitted. One operation is colored to mark the position for one of two injections of a database connection slowdown.

```
klotz:~# hdparm -X udma5 /dev/hda
  /dev/hda:
   setting xfermode to 69 (UltraDMA mode5)
klotz:~# hdparm -tT /dev/hda
  /dev/hda:
   Timing cached reads:         716 MB in  2.00 seconds = 357.73 MB/sec
   Timing buffered disk reads:  172 MB in  3.02 seconds =  56.94 MB/sec
klotz:~# hdparm -X mdma1 /dev/hda
  /dev/hda:
   setting xfermode to 33 (multiword DMA mode1)
klotz:~# hdparm -tT /dev/hda
  /dev/hda:
   Timing cached reads:         704 MB in  2.00 seconds = 352.03 MB/sec
   Timing buffered disk reads:   32 MB in  3.17 seconds =  10.08 MB/sec
klotz:~#
```

Figure 4.8: Manipulation of the hard disk transfer mode with `hdparm`. First the mode is set to *Ultra DMA 5* where the device delivers 57 MB per second. Then the mode is reduced to *Multiword DMA 1* where the transfer rate is cut to effectively 10 MB per second.

only component containing more than one operation that accesses a database, and that is instrumented as well as actually called in the experiments.

4.3.3 Hard Disk Misconfiguration

With hard disk drives, a broken wire would cause much trouble on operating system level, so that the origin can be quickly localized. A badly configured or deactivated DMA mode however would probably not produce any erroneous output, but slow down the I/O transfer rate and (depending on the transfer mode) may increase the CPU load. Affecting a whole machine, this fulfills the demand for a fault on *deployment context level.*

As demonstrated in Figure 4.8 on the host `jpet2` ("klotz"), a call of `hdparm -X mdma1 /dev/hda` (requiring super-user privileges) reduces the hard disk transfer mode from *Ultra DMA 5* to *Multiword DMA 1.* As a result, the buffered disk read performance drops significantly from about 57 MB/sec (udma5) to 10 MB/sec, although the ATA-2 standard allows 13.3 MB/s according to Kozierok [2001]. Since slower modes cause system instability on klotz's disk drive (Maxtor 6Y080L0), and the PIO mode in particular (that is known to have a big impact on overall system performance) cannot be set, the CPU is even less used than normal. Precisely, the CPU load during disk access drops from about 7% to 2%.

The hosts `jpet3` to `jpet5` – "scooter", "tier", and "puck" – all have a slower disk drive of the same type (WDC WD800AB-00CBA0) that drop from about 40 MB/sec (udma5) to 10 MB/sec (mdma1). The database host `jpet5` ("sam") has a similar drive (WDC

WD800JD-75JNC0) that performs at about 55 MB/sec by default, but gives I/O errors when accessed with hdparm, so this drive is resigned from the experiments.

4.3.4 High System Load

To extend the evaluation beyond the foregoing “simple” software faults with a distinct cause, this scenario allows to study the behavior of the distributed JPetStore when the respective machines are exposed to high system load. In reality, this could be evidence e.g. for a misconfiguration of the operating system, a resource consuming fault in another application, or malicious usage.

Like with Rohr et al. [2008a], *resource intensive processes* are executed on operating system level to increase overall system load. These processes stress the CPU, the main memory, and the hard disk drive, classifying this manipulation to *deployment context level*. Instead of changing some part of the application or the machines for the whole experiment time, this “anomaly injection” can be switched spontaneously during runtime. Besides start time and duration, the host name can be configured as well as one of five levels of “severity”, each providing a set of activities with a different influence on the overall performance. To observe the effect of the load on the system’s resources, the CPU and memory consumption are separately monitored using Sysstat 8.0.4, a collection of performance monitoring tools for Linux by Godard [2008].

Preparation experiments indicate that an adequate challenge for the Correlator plug-in is provided not until severity level 5. The influence of lower levels on timing behavior is noticeable, but it tends to disappear in statistical noise.

4.3.5 CPU Throttling

With active CPU cooling, it can be assumed that the fan rotation speed is monitored, or even logged in a business environment. Thus, a detaching of the connector will lead to an entry in the log, and perhaps a warning message is instantly sent to an administrator. The same applies to network connections, firewall misconfigurations, or other connections in hardware: The adjacent components will quickly perceive and classify the failure.

On the other hand, a breakage at the retention module of the CPU heat sink will probably not induce an instant warning message. Depending on the configuration and system load, the temperature will rise slowly. At certain thresholds, the CPU is throttled, i.e. idle cycles are added. Before the device is eventually switched off to avoid damage, the decreasing performance capacity should have a significant influence on the system’s timing behavior. Like the disk drive misconfiguration, this fault has an impact on *deployment context level*.

This experiment is inspired by a famous on-line video by Tom’s Hardware Team [2001] where the removal of a CPU cooler during a 3D game lets the speed (in frames per second) fall off, but throttling via the mainboard prevents overheating of the CPU, and by personal experience with a broken retention module that caused the system to become unstable.

CPU throttling can also be activated by software via ACPI interface to save power independent of frequency scaling as documented e.g. by Brodowski and Henschel [2002, 2004] and Torres [2005]. To not risk a damage on our testing environment, which is needed for further studies, we decide to keep the CPU cooler intact, and instead use the software throttling method to simulate a broken cooling system.

Since our testing environment (based on an Intel Pentium 4 "Willamette") does not support the direct access through the `acpi-cpufreq` kernel module, we load the module `p4-clockmod` and install the package `cpufrequtils` that allows to activate certain performance levels through "governors", or to select specific clock frequencies which is well explained e.g. by ArchWiki [2008]. For the first Pentium 4 generation, the frequency change command by the usermode software is converted into one of eight throttling modes by the CPU driver. According to Intel Corporation [2008, chap. 13.5], instead of using the STPCLK pin, or the HLT instruction, or changing the clock frequency, the throttling implementation of the P4 modulates the clock duty cycle of the CPU.

After the results of the system load experiments turn out to be discouraging, and a similar behavior could be expected for throttling, the `Account` context on the host `jpet3` ("scooter") is chosen for this manipulation only. Precisely, the CPU is throttled from 1600 MHz default to 800 MHz, resulting in a duty cycle of 50%.

4.4 Experiments

Preparation experiments show that repeated experiments under the same conditions produce very similar results. Thus, no effort is made to average the raw measured values, because a method for doing this would have to be developed yet. Instead, the experiments are run three times each, and the results are visually compared. In case of a significant difference within the triplet, the experiment is repeated.

Table 4.4 shows a summary of the fault injection scenarios. Out of five manipulation variants, 14 scenarios are defined. With three executions each, this results in a total of 42 experiments. Additionally, to get "historical" timing behavior that is assumed to be free of anomalies, thus to make a basis for the training of the anomaly detector, three experiments are run with a fault-free scenario.

The duration of the measurements is exactly 20 minutes each, as configured via a JMeter script. Including reboots of the machines, a warm-up phase, and the automatic post-processing, the experiments have a duration of about 25 minutes per run. Thus, the total experiment time reaches about 19 hours for the 45 runs.

4.4.1 Activities

Once the environment is set up, the main experiment steps are:

1. Activation and configuration of the respective faults for each experiment run.

2. Re-deployment of the distributed JPetStore to the machines after faults have been injected into, or removed from the source code.

Manipulation	Quantity	Position

Programming Faults
(Operation Level)
	3	`presentation.CatalogBean`
		`service.hessian.client.CatalogService`
		`persistence.sqlmapdao.ItemSqlMapDao`

DB Conn. Slowdown *(Component Level)*	2	`persistence.AccountSqlMapDao` `persistence.sqlmapdao.ItemSqlMapDap`

HDD Misconfiguration
(Depl. Context. Level)
	4	`Presentation`
		`Catalog`
		`Order`
		`Account`

High System Load	4	`Presentation`
		`Account`
		`Catalog`
		`Order`

CPU Throttling	1	`Account`

Table 4.4: Summary of the fault injection scenarios. 14 scenarios are defined out of five manipulation variants.

3. On the controlling host, the experiments are executed in a shell script that calls an Ant script in a loop. Debug messages are appended to a log file.

4. The Ant script first resets the four JPetStore machines. They are rebooted, the Servlet containers are started, and a mechanism for time synchronization is initialized.

5. The main routine then runs a short warm-up workload on the distributed system, and the monitoring for the response times and the hardware is enabled.

6. JMeter executes the 20 minutes workload. The test plan is identical for all runs, varied only through the probability functions.

7. The controlling host fetches the results and generates some reports.

8. Depending on the fault variant, the fault injection has to be manually reset.

Figure A.1 on page 92 provides a more thorough diagram of all activities around the experiments.

4.4.2 Results

45 datasets have been collected that take up 4.3 GiB of disk space. Each experiment provides about 370,000 executions, making up for more than 16 million executions in total. No inconsistencies within the triplets have been noticed, so the first result of each experiment triplet is chosen for evaluation. Thus, more than 5 million executions are to be evaluated.

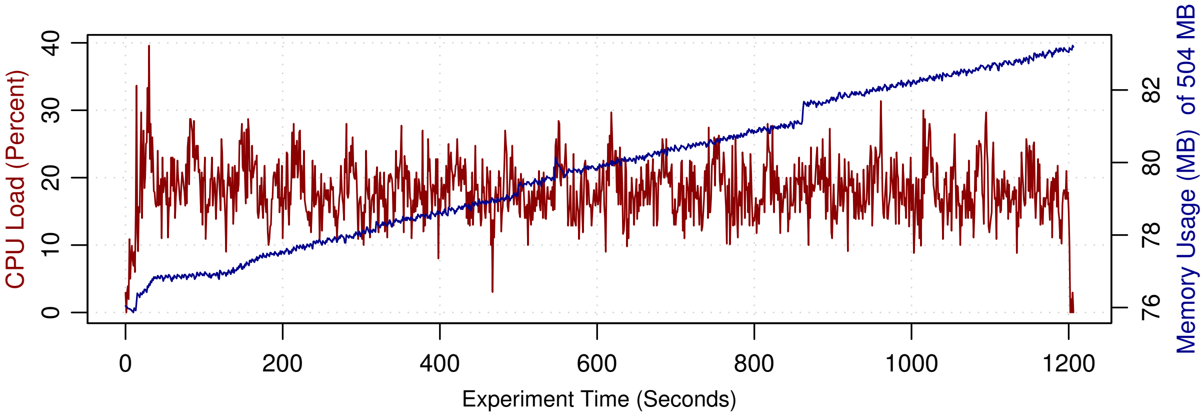


Figure 4.9: Example of a Sysstat plot during an experiment run. CPU and Memory consumption are plotted against experiment time. This example shows `jpet4` ("tier") during a fault-less run.

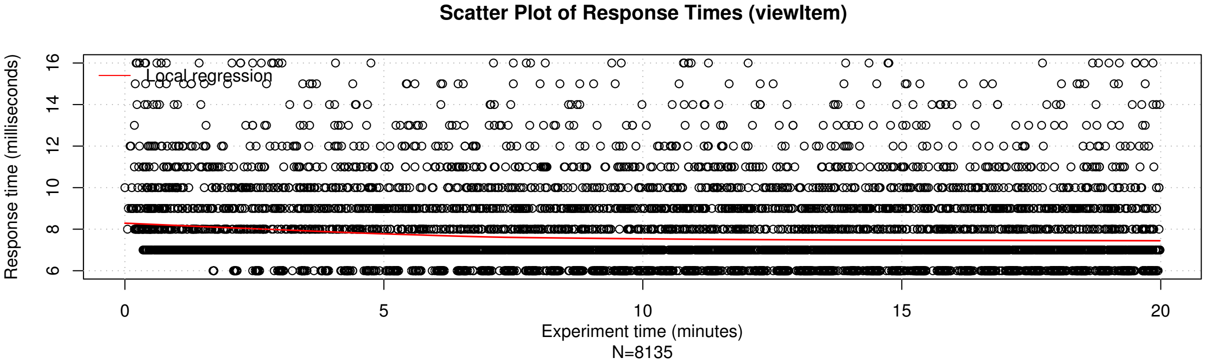


Figure 4.10: Example of a response time plot during an experiment run. 8135 response times have been collected from the Web server logs for the service `viewItem`.

The Sysstat plots automatically created after each experiment run confirm that the system's capacity is not reached. None of the curves get near the limit of the respective resource. Figure 4.9 shows the resource usage of `jpet4` ("tier", hosting the `Catalog` context), whose CPU is the resource with the highest load, matching the observation of Rohr et al. [2008a] who identify this CPU as the "bottleneck resource".

The response time plots show irregularities for some methods at the beginning of each run – Figure 4.10 depicts a typical representative where the timing behavior during the first few minutes differs from that during the rest of the plot. Apparently, the dedicated warm-up phase of 35 seconds is too short. To keep this from affecting the evaluation, we ignore the first 5 minutes of each experiment run, considering them as warm-up, too, thus reducing the evaluated timespan to 15 minutes per experiment.

Another observation is that, regardless of the injection variant and position, the anomaly detector *always* classifies a considerable fraction of the executions as behaving anomalous. This is reflected in red peaks in the anomaly score histograms as can be seen in Figure 4.11. While there are some anomalies in every element, this effect is

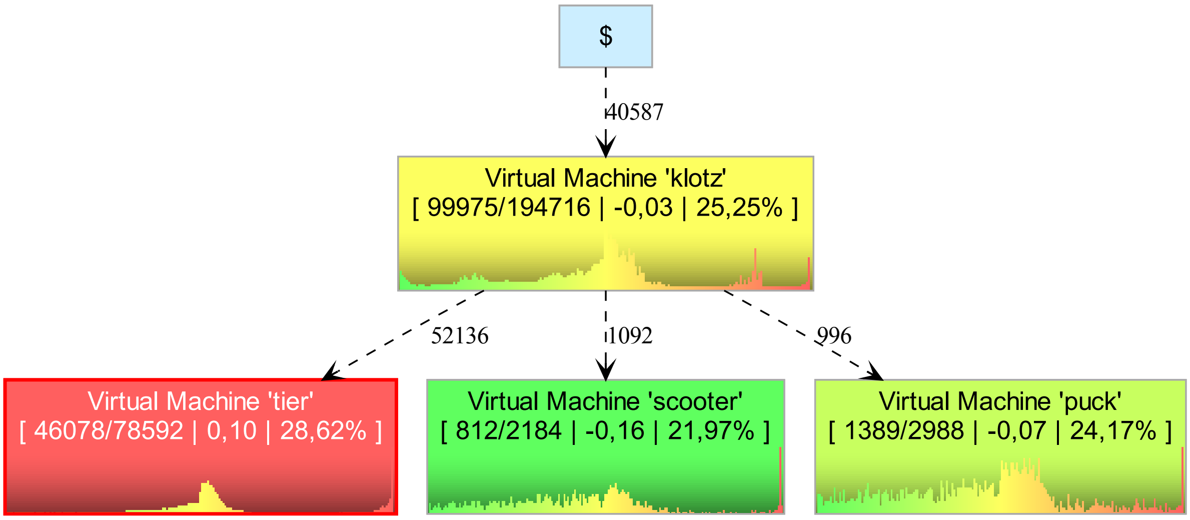


Figure 4.11: Example of the result of a simple anomaly correlation. Although the fault has been injected in "puck", that clearly shows a red peak, and a propagation to "klotz" can be assumed, the anomalies detected in "tier" are the strongest.

particularly manifested within the `Catalog` deployment context hosted on jpet4 ("tier"). Consequently, a fault's impact has to be quite strong to outperform the permanent anomaly at the `Catalog` context.

Apart from that, the dependency graphs of the five scenarios, simply aggregated using the "trivial" algorithm with default parameters, show different results. Table 4.5 gives an overview.

1. The impact of the programming faults is clearly visible. The scatter plots show larger response times that are noticed by the anomaly detector, thus the histograms created by the Correlator plug-in show a "red shift". However, the fault localization is not perfect in this stage: In some cases, the known-faulty element is deep red, but other elements are flagged as being possible causes.

2. The results for the database connection slowdown are even more clear. The elements are almost correctly flagged, and a strong effect of propagation can be noticed.

3. Although not unexpected, the hard disk experiments are rather disappointing. Regardless of where the fault injection happened, the cause is always classified to be within "tier", hosting the `Catalog`. Apparently, the hard disk is a resource that is not much used by the distributed JPetStore.

4. Although the impact of the high system load on CPU usage is clear on all four machines as can be seen in Figure 4.12 for `jpet3`, the effect on response times is nonuniform. While the `Account` context seems to be sensitive to system load, the

Manipulation	Result
Programming fault 1 – Double code	Clearly visible, no propagation, not perfect.
Programming fault 2 – Double code	Perfectly visible, nice propagation.
Programming fault 3 – Empty list	Clearly visible, some propagation, not perfect.
DB conn. slowdown 1 – `Account`	Clearly visible, some propagation, not perfect.
DB conn. slowdown 2 – `Catalog`	Clearly visible, much propagation, nearly perfect.
HDD misconfig. 1 – `Presentation`	Completely wrong.
HDD misconfig. 2 – `Account`	Completely wrong.
HDD misconfig. 3 – `Catalog`	Insignificant.
HDD misconfig. 4 – `Order`	Completely wrong.
High system load 1 – `Presentation`	Completely wrong.
High system load 2 – `Account`	Clearly visible.
High system load 3 – `Catalog`	Insignificant.
High system load 4 – `Order`	Completely wrong.
CPU throttling – `Account`	All too clear.

Table 4.5: Summary of the fault injection results. Some scenarios have to be taken out of the analysis, because in the `Catalog` context, in many cases an accumulation of anomalies is found that disguises the real cause of failure.

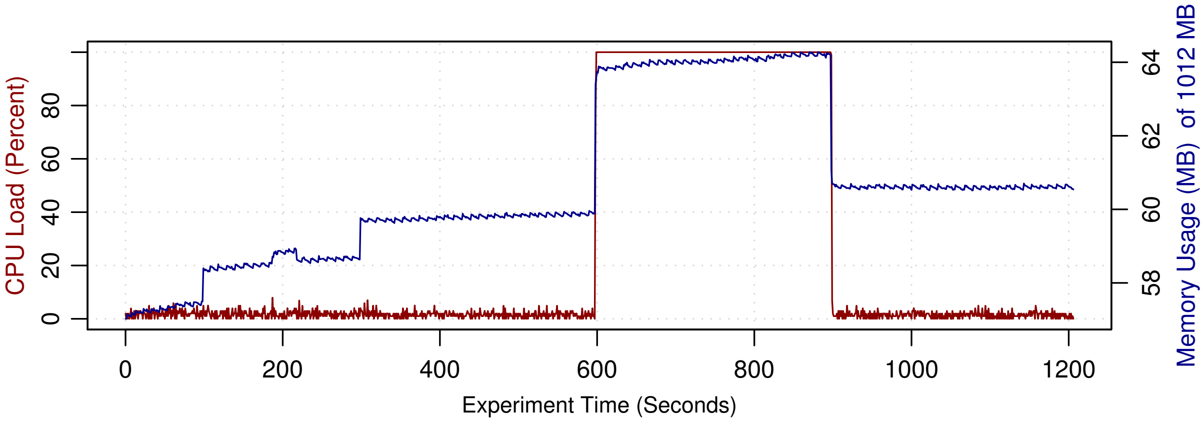


Figure 4.12: Sysstat plot showing the impact of system load on `jpet3` “scooter”: CPU and memory usage are increased during a timespan of 5 minutes.

Order context is not far behind, and the Catalog context is out of the game (as noticed above), the Presentation context shows very little reaction.

5. The throttling of the CPU of jpet3 is clearly visible in the Sysstat diagrams: The CPU load is doubled from about 2% to 4%. Although this is still far away from the system's capacity, and an influence on the timing behavior is not necessarily to be expected, there is a definite impact in that the anomaly detector evaluates *every* single execution in the affected deployment context as behaving anomalous.

The three injection variants on deployment context level have the same problem: Either the injection's effects are so strong that the result is undoubtful, or they are so weak that they are easily hidden by other (false) anomalies. In both cases, further analysis does not seem to make much sense, thus the following examinations focus on operation level and component level.

4.5 Analysis

To evaluate the abilities of the Correlator, a series of seven examinations is performed with varying settings and parameters. As a basis, those values are used for the parameters that have been estimated during development. They are varied independently to each find a value that produces the best result. Then, the optimized values are applied in combination, and the results compared with those from the beginning of the analysis. Finally, the performance of the algorithms is evaluated in a cross-check examination with data that the parameters have not been trained for.

The formulas for *clearness* and *success* presented at the beginning of this chapter are used to compare the results. Additionally, the augmented dependency graphs are visually compared and checked for irregularities and special characteristics.

The "historical" data that is assumed to be fault-free is used to train the anomaly detector. The training data is written to a file that is later read for all further examinations. Since WISAD is still under development, and we use constant workload (while WISAD's focus is to work with varying workload intensity), we decide to use its predecessor PAD (Plain Anomaly Detection) for the analysis.

4.5.1 Experiment Selection

Two representative scenarios are selected out of the data that was collected in the 42 experiments. One that looks quite clear with default parameters and the trivial algorithm, thus not really needing further analysis, and another scenario that points to the right direction, but does not locate the right fault. It makes no sense to examine scenarios that clearly propose a wrong element as the cause, because in these cases, the injected fault is too weak to have a significant effect, the anomaly detector has failed, or there has been some problem with the experiment.

1. The result of the second database connection slowdown experiment, where a `Thread.sleep(10)` command has been injected into every monitored database call of the component `persistence.sqlmapdao.ItemSqlMapDao`, is nearly perfect. All elements related to the fault are deep red and get high ratings. The highest ratings on their respective levels are assigned to the correct elements. A strong propagation effect is visible up to the `Presentation` layer. The remaining goal for non-trivial correlation algorithms is to improve the contrast, to emphasize the cause of failure even more, and to level down other elements.

2. In case of the third programming fault experiment, where an empty list is created instead of fetching a list through an access to the database in the method `getItemListByProduct(String,int,int)` in the `CatalogService` on `Presentation` layer, the pure numbers of the result are not perfect, but should allow an experienced administrator to draw the correct conclusions. The causing element receives a high rating, and a propagation to subsequent elements is visible, but other elements are rated as being the highest each, in fact on every hierarchy level. Hence, the main goal for further analysis is to correct the result to show right cause as clearly as possible.

4.5.2 Examination Activities

The analysis is automated to a large extent. Tpan takes several parameters that can be passed through command line parameters, or listed in `Properties` files. The use of `Properties`, the class loading functionality, and the batch override mode mentioned in Section 3.6.2 can save much time by allowing to run the Correlator plug-in multiple times on the same data without having to restart Tpan, or to reload the data.

The remaining activities are:

Configuration of the Tpan control script. Menu entries are pre-selected, e.g. to remove 300 seconds each. This has to be done only once.

Selection of the data source. For each examination, one of the two chosen experiments is selected to import the raw timing behavior data from.

Selection of the examination parameters. According to the five examination criteria, the algorithms and their parameters are specified as well as additional output configurations.

Figure 3.1 on page 18 gives an overview of the data flow within Tpan while the lower part of Figure A.1 on page 92 lists the specific activities of the automated analysis. The following steps are performed after the data has been loaded and pre-processed by Tpan, and the Correlator plug-in is activated:

- The experiment configuration is loaded either from the Tpan `Properties`, or from a special CSV file as introduced in Section 3.6.2. This may contain parameter overrides as well as the position of fault injection. If no instructions are found, a default experiment is created that does not override any parameters.

- For each experiment, a complete and independent analysis is performed by the Correlator. After constructing a model of the application under analysis from message traces and evaluated executions, the specified algorithm for correlation is loaded and applied on the data as detailed in Section 3.5.
- The results are printed to the command line interface, and written to the specified graphical output formats as described in Section 3.6.
- Finally, for all experiments, a summary file is written in CSV format.

4.5.3 Default Parameter Selection

The trivial and the simple algorithm do not have any parameters to vary. Most of the starting parameters for the advanced algorithm are set rather neutral: The "in out relation" factor, and the "neighborhood mean distance exponent" are set to 1.0, eliminating any special influence. The mean calculation methods for correlation on operation level, and for aggregation on deployment context level are switched to simple arithmetic mean. The aggregation on operation level is done using a root mean square (power mean with exponent 2.0) by default, while the aggregation on component level relies on a maximum function. Finally, the edge weight method is switched to absolute mode that includes the number of executions per element into the calculations.

The following sections contain descriptions of the seven examination variants that have been applied: Five variations of settings and parameters, and two applications of the new found parameters.

4.5.4 Three Algorithms

The Correlator plug-in brings along three different implementations to process the data as described in Section 3.5.3. In short, the *trivial algorithm* performs pure aggregation using unweighted arithmetic mean calculations on each hierarchy level only. The *simple algorithm* takes some effort to produce a better result using few simple rules to detect two special configurations in the application element structure that are indicators to increase or decrease the ratings. The *advanced algorithm* is based on some speculations on the effects of anomaly propagation. It is much more complex than the other algorithms in that it uses additional parameters and calculations.

The goal of this first examination is to check whether the correlation result improves with increasing complexity of the algorithms, and to have a basis for the other examinations that vary different parameters of the advanced algorithm.

Regarding the numbers shown in Table 4.6, the *success* of 1.0 confirms that for the database connection slowdown, every algorithm identifies the correct cause. For both

Experiment	**Algorithm**	**success**	**clear$_{\text{op}}$**	**clear$_{\text{comp}}$**	**clear$_{\text{dc}}$**	**CR$_{\text{max}}$**
DB conn. slowdown	Trivial	1.0	0.3615	0.5934	1.2420	1.0
Fault scenario No. 5	Simple	1.0	0.3793	0.7081	1.3542	1.0
(Component level)	Advanced	1.0	0.3641	0.5363	1.2288	0.7018
Programming fault	Trivial	0.99996	0.4690	0.4854	1.0573	0.9948
Fault scenario No. 3	Simple	0.99996	0.4988	0.4938	1.0770	0.9948
(Operation level)	Advanced	1.0	0.3857	0.4780	0.9363	0.4880

Table 4.6: Results of the experimental comparison of three algorithms in two scenarios. The *clearness* function is given for the three hierarchy levels – operation, component, and deployment context. The last column contains the highest cause rating on the level where the fault injection has been applied.

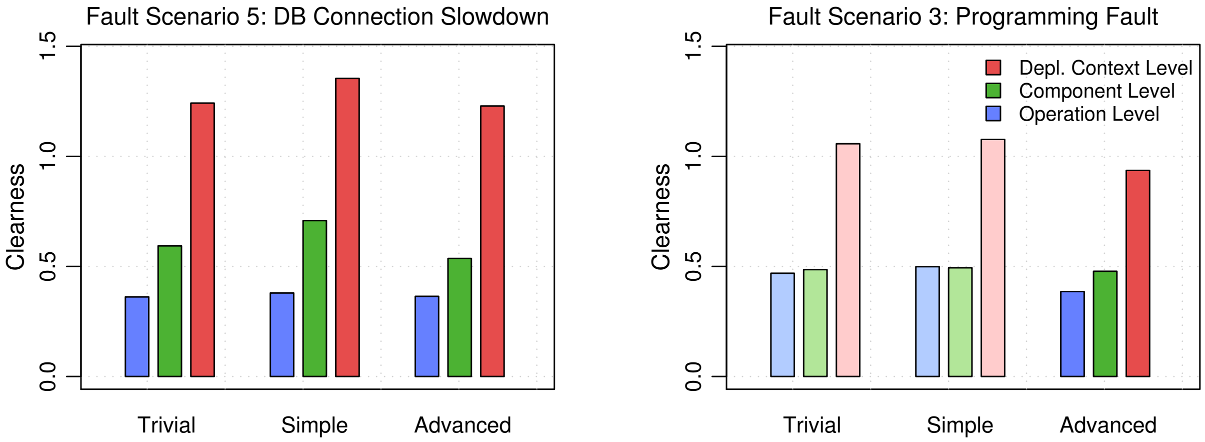


Figure 4.13: Charts of the *clearness* ratings for three algorithms in two scenarios. The colors are pale where the *success* does not reach 100%, thus the *clearness* values are less relevant.

Experiment	**Relation**	**success**	$\textbf{clear}_{\textbf{op}}$	$\textbf{clear}_{\textbf{comp}}$	$\textbf{clear}_{\textbf{dc}}$	$\textbf{CR}_{\textbf{max}}$
DB conn. slowdown	0.1	1.0	0.3638	0.5389	1.2377	0.7018
Fault scenario No. 5	0.2	1.0	0.3640	0.5387	1.2367	0.7018
(Component level)	0.3	1.0	0.3641	0.5384	1.2357	0.7018
	0.5	1.0	0.3642	0.5387	1.2336	0.7018
	1.0	1.0	0.3641	0.5363	1.2288	0.7018
	2.0	1.0	0.3636	0.5333	1.2203	0.7018
	3.0	1.0	0.3628	0.5302	1.2116	0.7018
	5.0	1.0	0.3608	0.5222	1.1936	0.7018
	10.0	0.9876	0.3579	0.5110	1.1578	0.7232
Programming fault	0.1	1.0	0.4000	0.4968	0.9341	0.4880
Fault scenario No. 3	0.2	1.0	0.3989	0.4958	0.9328	0.4880
(Operation level)	0.3	1.0	0.3974	0.4943	0.9314	0.4880
	0.5	1.0	0.3942	0.4901	0.9285	0.4880
	1.0	1.0	0.3857	0.4780	0.9363	0.4880
	2.0	0.9304	0.4044	0.5126	0.9551	0.5993
	3.0	0.8753	0.4226	0.5529	0.9698	0.7000
	5.0	0.8188	0.4403	0.6029	0.9895	0.8174
	10.0	0.7661	0.4551	0.6571	1.0072	0.9424

Table 4.7: Results of the experimental comparison of the in-out-relation in two scenarios.

experiments, and for every hierarchy level, the *clearness* is the highest with the simple algorithm, as can also be seen in Figure 4.13. While the advanced algorithm reaches a 100% *success* for the programming fault injection, the highest cause rating drops significantly for both experiments.

Note: Some of those values that seem to be identical in fact differ in the 15th decimal place – that might be due to rounding errors inherent to the runtime environment.

4.5.5 In-Out-Relation

In this examination, the relation between the influence of incoming and outgoing connections in the dependency structure during correlation on operation level that has been introduced in Equations 3.13 and 3.14 on page 39 is varied in the range of $[0.1, 10.0]$.

For the database connection slowdown experiment, Table 4.7 and Figure 4.14 show few variations in the results. There is a slight tendency that smaller values work better, i.e. the influence of the outgoing (callee) connection is weakened. For very large values, the results become incorrect ($success < 1.0$).

The second experiment supports that: For small values, there is no significant change, while large values spoil the quality of the results.

4.5.6 Edge Weight Methods

This short examination is about different methods to calculate the edge weight used for correlation on operation level. The “absolute” method uses the number of connections between the elements, while the “relative” method uses their percentages.

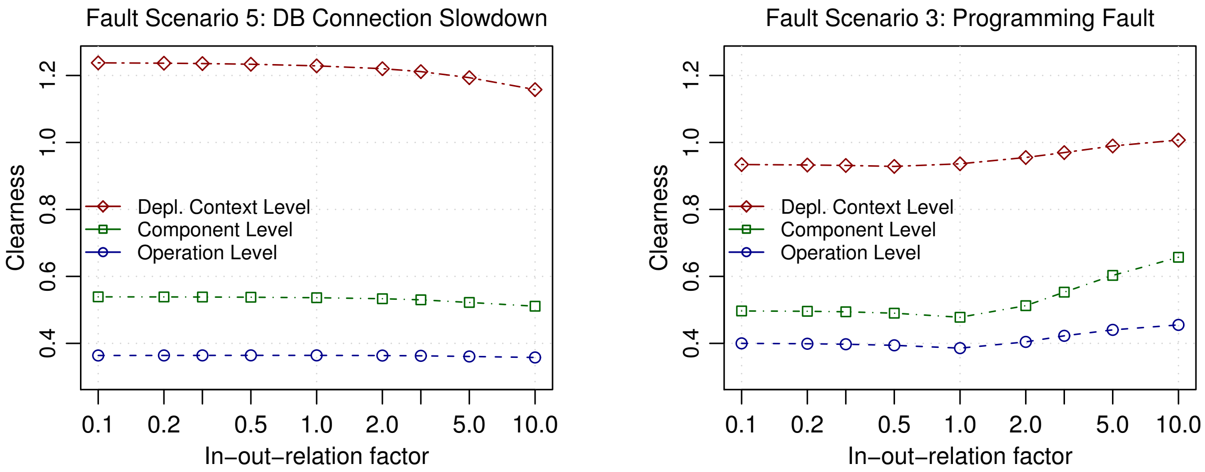


Figure 4.14: Charts of the *clearness* ratings plotted against a number of values for the relation between incoming and outgoing connections in two scenarios. In the second scenario, the good performance for large values is misleading as long as the related *success* rating is smaller than 1.0.

Experiment	**Method**	**success**	**clear**$_{\mathbf{op}}$	**clear**$_{\mathbf{comp}}$	**clear**$_{\mathbf{dc}}$	**CR**$_{\mathbf{max}}$
DB conn. slowdown	Absolute	1.0	0.3641	0.5363	1.2288	0.7018
Fault scenario No. 5	Relative	1.0	0.3693	0.5612	1.3156	0.8354
Programming fault	Absolute	1.0	0.3857	0.4780	0.9363	0.4880
Fault scenario No3̇	Relative	1.0	0.4323	0.5349	0.9538	0.5653

Table 4.8: Results of the experimental comparison of the edge weight method in two scenarios.

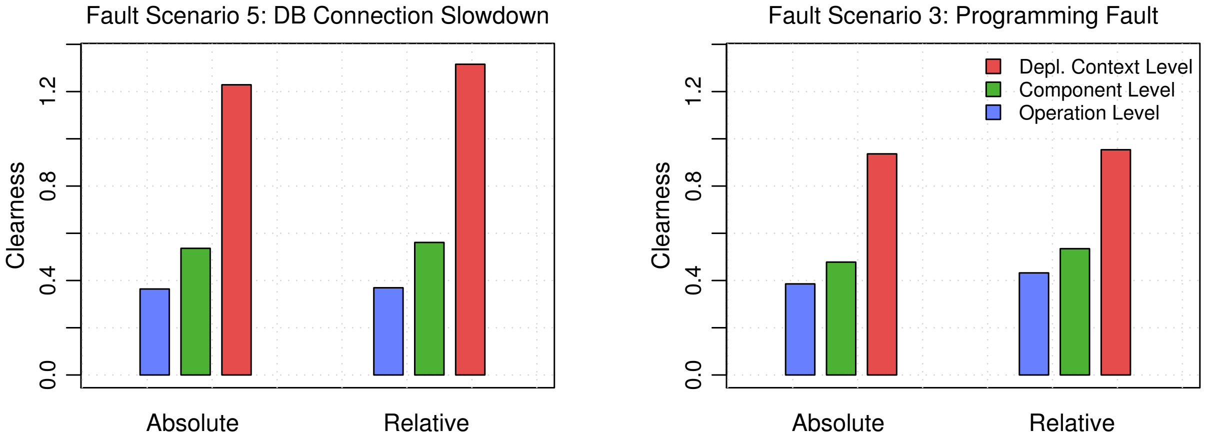


Figure 4.15: Charts of the *clearness* ratings for two edge weight methods in two scenarios.

For both experiments, the numbers in Table 4.8 that are also plotted in Figure 4.15 reveal a distinct advantage of the "relative" method: All *clearness* ratings as well as the highest ratings are increased. However, for the programming fault experiment with the "relative" method, a false deployment context is marked with the highest probability of containing the cause as a consequence of the correct context receiving a lower rating.

4.5.7 Mean Calculation Methods

By default, as described in Section 4.5.3, the Correlator makes use of three different methods in different situations to each combine many values into one value: The median function, the maximum function, or the power mean function. The latter, provided with a variable exponent, is a superset of several mean functions, including the arithmetic mean (exponent 1.0), and the root mean square (RMS, exponent 2.0). With increasing exponent, the influence of outliers is also increased.

Traditionally, the median is used to handle outliers in the samples. In the correlator's input data however, first, because of the pre-processing by the anomaly detector, and because of the limitation of the anomaly score to $[-1, 1]$, there are not any real outliers, and second, the extreme values are not considered "contaminations" but should instead get special respect. Therefore, it is expected that the root mean square performs better than the median.

40 passes of the Correlator plug-in are performed to analyze the influence of the mean calculation methods on the result: For each of the four situations, and for both scenarios, five methods are applied. Regarding the numbers in Table 4.9 as well as the charts in the Figures 4.16 and 4.17, for both scenarios, for aggregation on operation level, the power mean with exponent 0.5 seems to perform the best, but the advance is small. The maximum method is far behind. For correlation on operation level, the median seems to perform well for the first scenario, but the result is wrong for the second one. The same applies to the "weakening" power mean. Instead, the maximum method performs the best. For aggregation on component level, the trend goes to weakening the outliers for the first scenario. For the second one, the results are less clear, but the maximum method performs slightly better. For aggregation on deployment context level, the trends are in opposite directions: While for the first scenario, the weakening of outliers is favored, for the second scenario, it seems to be better to emphasize them – thus there is no winner on this level.

4.5.8 Neighborhood Mean Distance Exponents

Introduced in Equation 3.8 on page 36, the neighborhood mean distance exponent is used to emphasize ($e_d > 1.0$) or weaken ($e_d < 1.0$) the influence of distance during the correlation on operation level. It is part of the edge weight calculation, where the influence of connected elements decreases with increasing distance to the element currently under processing.

(a) Fault scenario 5: “Database connection slowdown”.

Calculation	**Mean Method**	**success**	**clear$_{\mathbf{op}}$**	**clear$_{\mathbf{comp}}$**	**clear$_{\mathbf{dc}}$**	**CR$_{\mathbf{max}}$**
Operation aggregation	Median	1.0	0.3668	0.5414	1.1667	0.7008
	Power Mean 0.5	1.0	0.3770	0.5359	1.1181	0.5311
	Arithmetic Mean	1.0	0.3722	0.5398	1.1487	0.6130
	Root Mean Square	1.0	0.3641	0.5363	1.2288	0.7018
	Maximum	1.0	0.2913	0.4079	0.8997	0.9949
Operation correlation	Median	1.0	0.3822	0.5936	1.4163	1.0
	Power Mean 0.5	1.0	0.3646	0.5316	1.2118	0.6474
	Arithmetic Mean	1.0	0.3641	0.5363	1.2288	0.7018
	Root Mean Square	1.0	0.3657	0.5468	1.2598	0.7512
	Maximum	1.0	0.3631	0.5640	1.4042	1.0
Component aggregation	Median	1.0	0.3641	0.6019	1.1702	0.7018
	Power Mean 0.5	1.0	0.3641	0.5935	1.1360	0.6245
	Arithmetic Mean	1.0	0.3641	0.5836	1.1498	0.6256
	Root Mean Square	1.0	0.3641	0.5717	1.1582	0.6279
	Maximum	1.0	0.3641	0.5363	1.2288	0.7018
Deployment context aggregation	Median	1.0	0.3641	0.5363	1.2937	0.7018
	Power Mean 0.5	1.0	0.3641	0.5363	1.2513	0.7018
	Arithmetic Mean	1.0	0.3641	0.5363	1.2288	0.7018
	Root Mean Square	1.0	0.3641	0.5363	1.2337	0.7018
	Maximum	1.0	0.3641	0.5363	1.2279	0.7018

(b) Fault scenario 3: “Programming fault”.

Calculation	**Mean Method**	**success**	**clear$_{\mathbf{op}}$**	**clear$_{\mathbf{comp}}$**	**clear$_{\mathbf{dc}}$**	**CR$_{\mathbf{max}}$**
Operation aggregation	Median	1.0	0.3780	0.4764	0.9501	0.4966
	Power Mean 0.5	1.0	0.3984	0.4923	0.9210	0.4782
	Arithmetic Mean	1.0	0.3914	0.4865	0.9280	0.4800
	Root Mean Square	1.0	0.3857	0.4780	0.9363	0.4880
	Maximum	1.0	0.2985	0.4086	0.8998	0.9940
Operation correlation	Median	0.9441	0.3726	0.4687	0.9464	0.4595
	Power Mean 0.5	0.9927	0.3752	0.4597	0.9361	0.4269
	Arithmetic Mean	1.0	0.3857	0.4780	0.9363	0.4880
	Root Mean Square	1.0	0.4214	0.5271	0.9694	0.5908
	Maximum	1.0	0.4977	0.7692	0.9751	0.9947
Component aggregation	Median	1.0	0.3857	0.4269	0.9477	0.4880
	Power Mean 0.5	1.0	0.3857	0.4305	0.9484	0.4880
	Arithmetic Mean	1.0	0.3857	0.4301	0.9482	0.4880
	Root Mean Square	1.0	0.3857	0.4264	0.9443	0.4880
	Maximum	1.0	0.3857	0.4780	0.9363	0.4880
Deployment context aggregation	Median	1.0	0.3857	0.4780	0.9288	0.4880
	Power Mean 0.5	1.0	0.3857	0.4780	0.9340	0.4880
	Arithmetic Mean	1.0	0.3857	0.4780	0.9363	0.4880
	Root Mean Square	1.0	0.3857	0.4780	0.9951	0.4880
	Maximum	1.0	0.3857	0.4780	1.0861	0.4880

Table 4.9: Results of the experimental comparison of mean calculation methods in two scenarios.

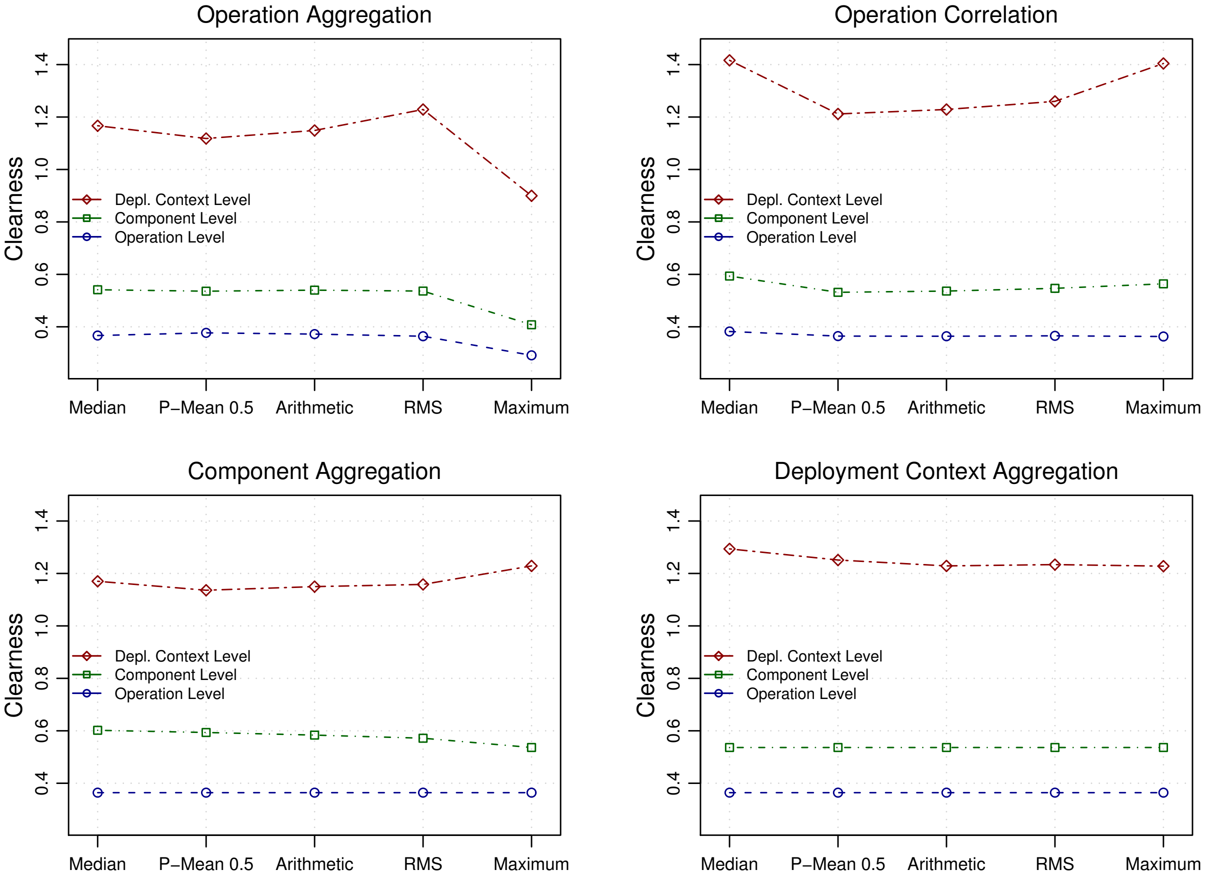


Figure 4.16: Charts of the *clearness* ratings for five mean calculation methods in the "database connection slowdown" scenario.

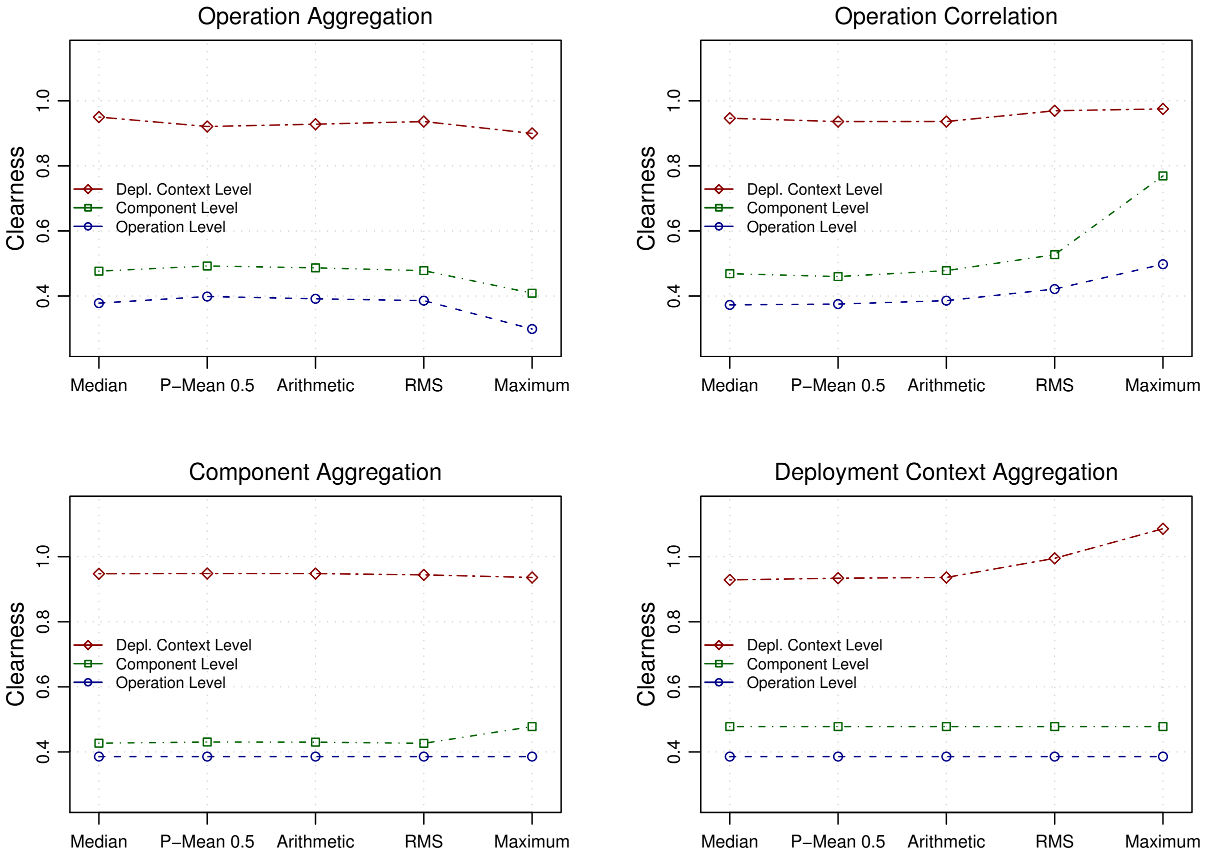


Figure 4.17: Charts of the *clearness* ratings for five mean calculation methods in the "programming fault" scenario.

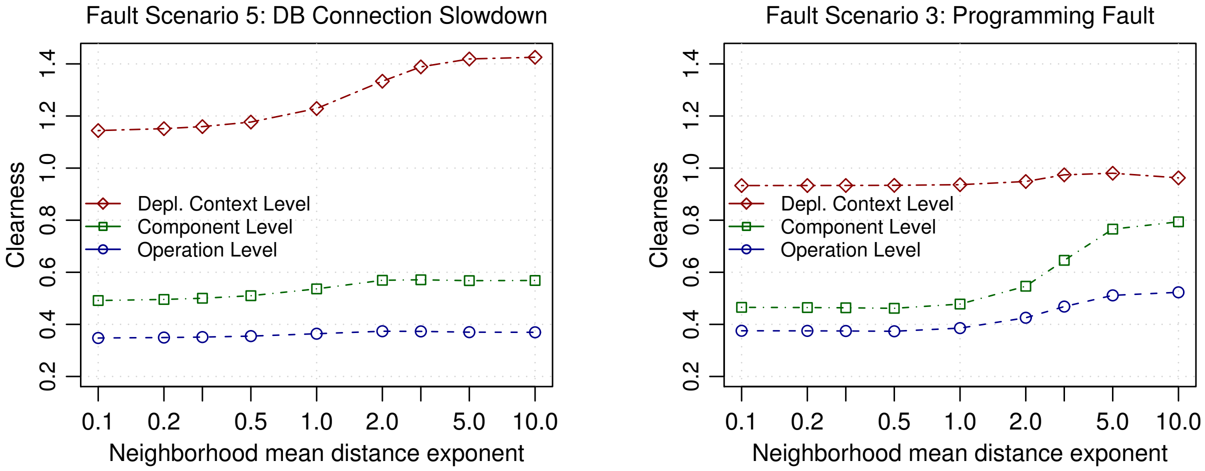


Figure 4.18: Charts of the *clearness* ratings plotted against a number of values for the neighborhood mean distance exponent in two scenarios.

Experiment	**Exponent**	**success**	**clear$_{\text{op}}$**	**clear$_{\text{comp}}$**	**clear$_{\text{dc}}$**	**CR$_{\text{max}}$**
DB conn. slowdown	0.1	1.0	0.3477	0.4915	1.1441	0.5093
Fault scenario No. 5	0.2	1.0	0.3494	0.4958	1.1514	0.5264
(Component level)	0.3	1.0	0.3512	0.5003	1.1593	0.5449
	0.5	1.0	0.3550	0.5101	1.1768	0.5854
	1.0	1.0	0.3641	0.5363	1.2288	0.7018
	2.0	1.0	0.3737	0.5693	1.3336	0.9016
	3.0	1.0	0.3731	0.5712	1.3887	0.9783
	5.0	1.0	0.3701	0.5679	1.4190	0.9992
	10.0	1.0	0.3696	0.5685	1.4256	1.0
Programming fault	0.1	0.9754	0.3755	0.4654	0.9330	0.4465
Fault scenario No3̇	0.2	0.9801	0.3751	0.4646	0.9332	0.4456
(Operation level)	0.3	0.9852	0.3746	0.4637	0.9334	0.4446
	0.5	0.9970	0.3735	0.4617	0.9339	0.4424
	1.0	1.0	0.3857	0.4780	0.9363	0.4880
	2.0	1.0	0.4254	0.5465	0.9486	0.6435
	3.0	1.0	0.4683	0.6459	0.9742	0.8143
	5.0	1.0	0.5114	0.7657	0.9803	0.9667
	10.0	1.0	0.5232	0.7939	0.9626	0.9945

Table 4.10: Results of the experimental comparison of different values for the neighborhood mean distance exponent in two scenarios.

The numbers in Table 4.10 as well as the charts in Figure 4.18 clearly show a trend that larger values perform better in both scenarios. For values smaller than 1.0, the result even becomes incorrect for the "programming fault" scenario.

4.5.9 Three Algorithms with New Parameters

To check how much the results of the advanced algorithm compared to the simpler algorithm variants can be improved when applying the parameter values that are found out to provide better results in the preceding examinations, another examination is performed that uses the combined new settings listed in Table 4.11.

Parameter	**Starting setting**	**Optimized setting**
In-out-relation	1.0	0.2
Edge weight method	Absolute	Relative
Mean calculation methods		
Aggregation on operation level	Root mean square	Power mean 0.5
Correlation on operation level	Arithmetic mean	Maximum
Aggregation on component level	Maximum	Arithmetic mean
Aggregation on depl. context level	Arithmetic mean	Root mean square
Neighborhood mean distance exponent	1.0	5.0

Table 4.11: Comparison of starting and optimized parameters.

Experiment	**Algorithm**	**success**	**clear$_{op}$**	**clear$_{comp}$**	**clear$_{dc}$**	**CR$_{max}$**
DB conn. slowdown	Trivial	1.0	0.3615	0.5934	1.2420	1.0
Fault scenario No. 5	Simple	1.0	0.3793	0.7081	1.3542	1.0
(Component level)	Advanced	1.0	0.3641	0.5363	1.2288	0.7018
	Optimized	1.0	0.3740	0.7461	1.3466	1.0
Programming fault	Trivial	0.99996	0.4690	0.4854	1.0573	0.9948
Fault scenario No. 3	Simple	0.99996	0.4988	0.4938	1.0770	0.9948
(Operation level)	Advanced	1.0	0.3857	0.4780	0.9363	0.4880
	Optimized	1.0	0.5176	0.4257	0.9027	0.9947

Table 4.12: Results of the advanced algorithm with optimized parameters, compared to the results from the first comparison of the three algorithm variants.

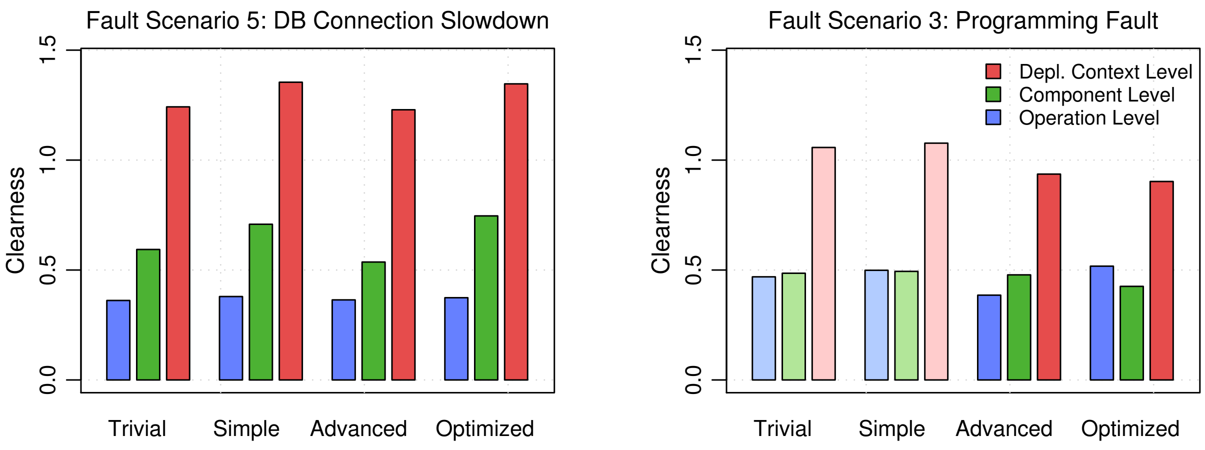
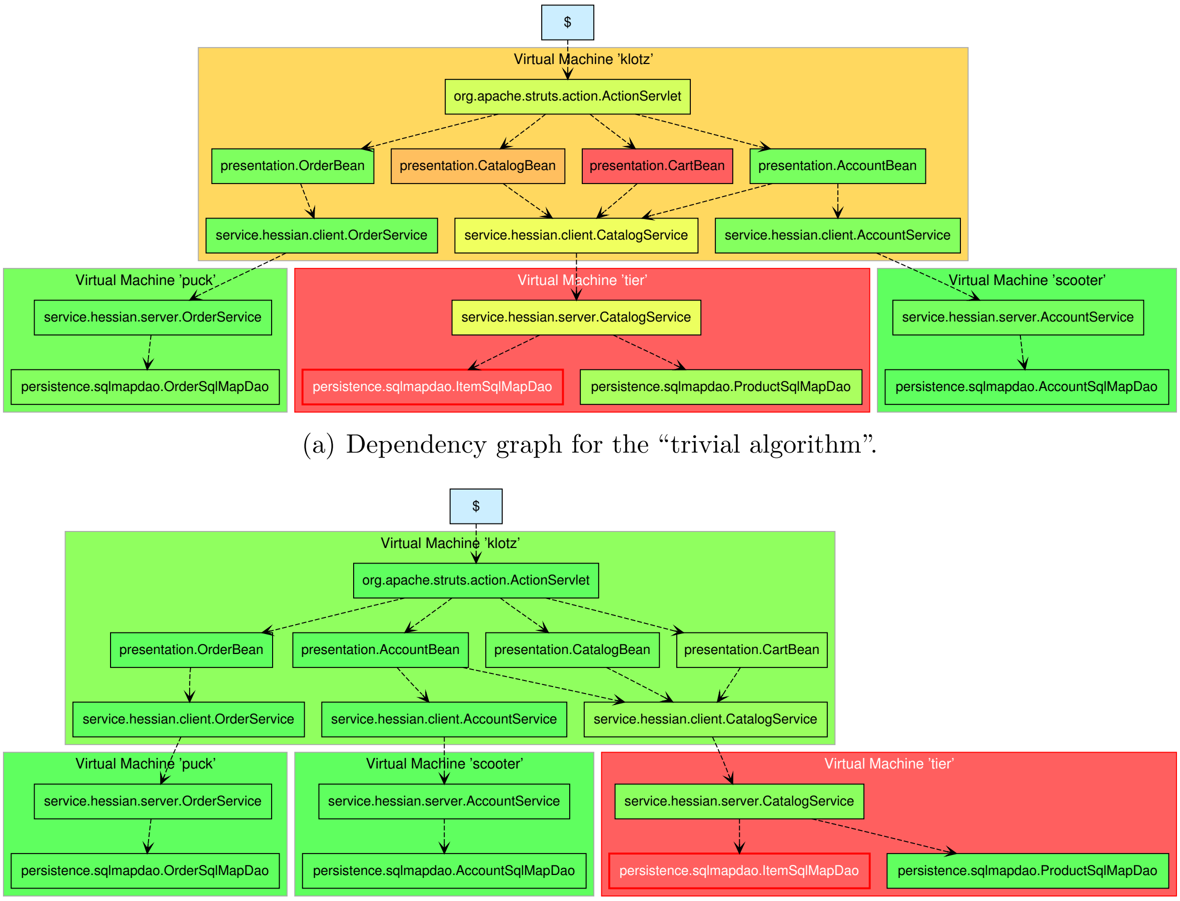


Figure 4.19: Charts of the *clearness* ratings for the advanced algorithm with optimized parameters, compared to the results from the first comparison of the three algorithm variants.


$
Virtual Machine 'klotz'
org.apache.struts.action.ActionServlet
presentation.OrderBean
presentation.CatalogBean
presentation.CartBean
presentation.AccountBean
service.hessian.client.OrderService
service.hessian.client.CatalogService
service.hessian.client.AccountService
Virtual Machine 'puck'
service.hessian.server.OrderService
persistence.sqlmapdao.OrderSqlMapDao
Virtual Machine 'tier'
service.hessian.server.CatalogService
persistence.sqlmapdao.ItemSqlMapDao
persistence.sqlmapdao.ProductSqlMapDao
Virtual Machine 'scooter'
service.hessian.server.AccountService
persistence.sqlmapdao.AccountSqlMapDao
$
Virtual Machine 'klotz'
org.apache.struts.action.ActionServlet
presentation.OrderBean
presentation.AccountBean
presentation.CatalogBean
presentation.CartBean
service.hessian.client.OrderService
service.hessian.client.AccountService
service.hessian.client.CatalogService
Virtual Machine 'puck'
service.hessian.server.OrderService
persistence.sqlmapdao.OrderSqlMapDao
Virtual Machine 'scooter'
service.hessian.server.AccountService
persistence.sqlmapdao.AccountSqlMapDao
Virtual Machine 'tier'
service.hessian.server.CatalogService
persistence.sqlmapdao.ItemSqlMapDao
persistence.sqlmapdao.ProductSqlMapDao

(a) Dependency graph for the "trivial algorithm".

(b) Dependency graph for the "optimized algorithm".

Figure 4.20: Comparison of the colored dependency graphs for correlation with the trivial and optimized algorithm for the "database connection slowdown" scenario.

As can be seen in Table 4.12 and Figure 4.19, for the "database connection slowdown" scenario, the optimized advanced algorithm performs better than the other variants, especially on component level, where the fault injection happened, and better than the non-optimized variant. The simple algorithm provides non-significantly higher values on operation and deployment context level though.

The results for the "programming fault" scenario are similar: The other variants provide single values that are slightly better than those of the optimized variant, but on the essential level where the injection took place – operation level in this case –, the result is correct ($success = 1.0$), and the *clearness* rating is good in comparison.

The advantage becomes more clear when looking at the colored dependency graphs. For example, Figure 4.20 depicts the results for the "database connection slowdown" scenario, for the trivial and the optimized algorithm, reduced to component and deployment context level. The correct elements are highlighted in both cases, and in deep red color, while the other elements are mostly green with a high contrast for the optimized

Experiment	Algorithm	success	clear_{op}	$\text{clear}_{\text{comp}}$	clear_{dc}	CR_{max}
Programming fault	Trivial	1.0	0.4620	0.5952	1.0563	0.7863
Fault scenario No. 1	Simple	1.0	0.4643	0.4991	1.0663	0.7863
(Operation level)	Advanced	0.9129	0.3486	0.4693	0.9735	0.5044
	Optimized	1.0	0.3437	0.4091	0.8965	0.2593
Programming fault	Trivial	1.0	0.4350	0.5817	1.2132	0.9928
Fault scenario No. 2	Simple	1.0	0.4383	0.5877	1.2272	0.9928
(Operation level)	Advanced	1.0	0.3947	0.5285	1.1076	0.6340
	Optimized	1.0	0.4763	0.5034	1.0187	0.9705
DB conn. slowdown	Trivial	1.0	0.4199	0.5484	1.5667	1.0
Fault scenario No. 4	Simple	1.0	0.4334	0.5884	1.6524	1.0
(Component level)	Advanced	1.0	0.3413	0.4408	1.0174	0.1997
	Optimized	1.0	0.4380	0.5947	1.5206	1.0

Table 4.13: Results of the cross-check examinations of three other fault injection scenarios, analyzed with four algorithm variants.

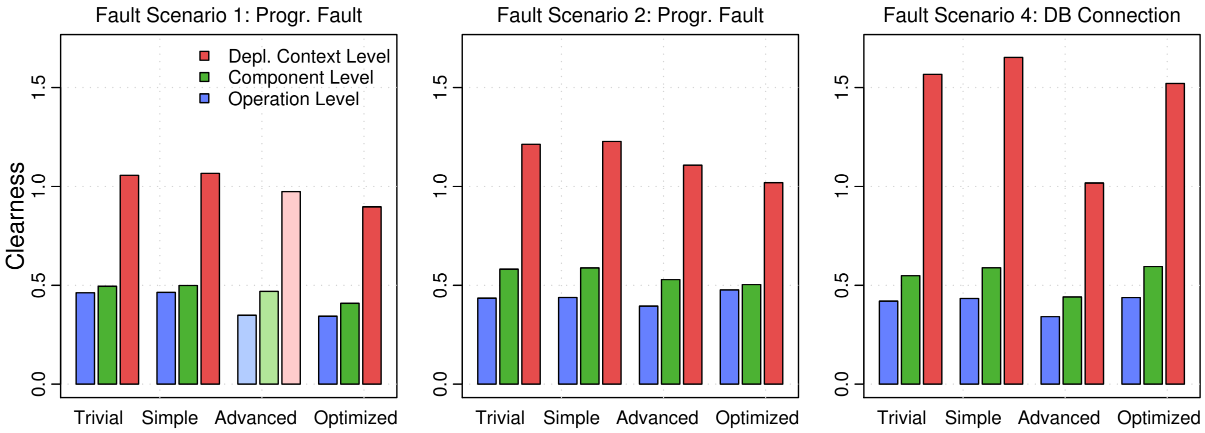


Figure 4.21: Charts of the *clearness* ratings for three other fault injection scenarios, analyzed with four algorithm variants.

algorithm. On the other hand, there is much yellow and orange color for the trivial algorithm, meaning uncertainty regarding the cause of failure.

4.5.10 Cross-Check Appliance to Other Fault Scenarios

To get a clue whether the newly found parameters are only useful with the “trained” scenarios, or whether they work with other scenarios as well, a cross-check examination is performed with the timing data from the experiment scenarios No. 1, 2, and 4. The first two of these were exposed to the programming faults that are discussed in Section 4.3.1 on page 55. The fourth scenario was subject to another database connection slowdown experiment as explained in Section 4.3.2 on page 58. As noted in Table 4.5 on page 66, No. 2 is *the* scenario that showed the best results in the first place, while No. 1 and 4 point to the right direction, giving opportunity for improvements.

Scenario	Injection Variant	Trivial	Simple	Advanced	Optimized	Avg.
No. 1	Programming fault	+	+	- -	o	3.0
No. 2	Programming fault	+	+	o	++	2.0
No. 3	Programming fault	-	-	+	++	2.8
No. 4	DB conn. slowdown	+	++	o	++	1.8
No. 5	DB conn. slowdown	+	++	+	++	1.5
Averages		2.4	2.0	2.8	1.4	2.2

Table 4.14: Overview of the examination results on the five fault injection scenarios on operation and component level. The averages have been calculated as if the rating system would equal the values from 1 (++) to 5 (- -).

For the first scenario, the numbers in Table 4.13 and the plots in Figure 4.21 show that the "optimized" algorithm variant performs significantly worse than the other variants, even than the un-optimized variant – but at least the result is *correct* now ($success = 1.0$).

For the scenarios No. 2 and 4, similar to the observations in the previous section, there is a significant enhancement for the optimized algorithm, on the respective level where the fault has been injected, at the expense of contrast on the other levels.

4.6 Summary

The fault injection experiments on deployment context level are of not much use for further examination, because their results are either all too clear, or they are completely wrong in many cases, because the `Catalog` context on the host `jpet4` ("tier") shows a large number of anomalies in almost *every* situation for unknown reason.

Four of the five faults that have been injected on operation and component level are localized with high precision by the optimized advanced algorithm variant. As can be seen in Table 4.14, the scenarios have different levels of difficulty, and the algorithm variants have different strengths.

Overall, the optimized advanced algorithm performs best in the selected examinations. It provides a correct and clear result not only for the scenarios it has been trained for, but also for two other fault scenarios as well – one programming fault, and one database connection slowdown. The remaining fault scenario is still evaluated correctly ($success = 1.0$), and a significant enhancement is achieved in comparison to the un-optimized variant, but the simpler algorithms provide higher contrast.

The simple algorithm has an advantage over the trivial variant, but the difference is small in most of the cases. When looking at the colored dependency graphs, the contrast produced by the simple algorithm is usually better. Images produced with the trivial algorithm – see e.g. Figure 4.20(a) – contain more yellow and orange shading, reflecting the presence of relative high ratings for a certain percentage of the elements.

Conclusion The advanced algorithm with optimized parameters offers a good chance to point to the right cause, but not without a risk of denoting false positives. This risk is much smaller with the trivial and simple algorithms, although they tend to show more of the *effect*, and less of the *cause*.

Performance

According to Ant, one complete automated run of Tpan with loading 60 MiB of raw timing data, and with correlation of about 41,000 message traces containing 262,000 executions takes round about 60 seconds on an Athlon 1500 MHz with 1 GiB main memory (3.2 GiB/sec peak transfer rate). The Java Virtual Machine uses up to about 400 MiB memory. The small share of time that the Correlator plug-in takes up can be estimated in that one run of Tpan including a series of 20 correlations (including the creation of the dot file and the export to SVG, PS, and PNG) takes little more than 200 seconds on the same machine. Another run with 9 correlations that took a total of 115 seconds suggests that for the current examinations, the pre-processing by Tpan requires about 50 seconds, and a correlation run takes about 7 seconds each.

Chapter 5

Conclusions

As part of an approach for failure diagnosis, a prototype for an anomaly correlator has been developed as a plug-in for *Tpan*, the analysis component of *Kieker* [Rohr et al., 2008b]. In order to improve the fault localization methods that are currently under development by our group, an existing technique for timing behavior anomaly detection [Rohr et al., 2008a] is combined with derived dependency graphs.

The *Correlator* is loaded with pre-processed timing behavior data in the form of *message traces* that allow to construct a calling dependency graph of the application under analysis. In the second step, information about timing anomalies are added in the form of *evaluated executions* that have been analyzed by an anomaly detector.

The idea of anomaly correlation is based on the assumption that timing behavior anomalies *propagate* through the dependency graph, leaving a kind of pattern. In the approach, the propagation is tried to perform backwards, examining small-scale configurations of application elements, and applying simple logical rules on the elements' inner state, also considering their respective environment in the dependency structure.

The plug-in can be extensively configured, and provides methods for automated control using Java `Properties`, an experiment batch instruction file, a parameter override mechanism, and an algorithm class loader. Three levels of aggregation – operations, components, and deployment context – are processed to calculate and assign *rating* values to the application elements that reflect the probability to be the cause of failure.

Along with Tpan, the main interface of the plug-in is the text console. Additional to status and debug messages on screen, a log file is written. The results of the analysis are collected per experiment in a CSV file, and can be fetched through method calls inside of Tpan. Visualization is possible via a highly configurable export to Graphviz dot files that can in turn be converted to various image file formats where the cause rating results are represented by color shades in green to yellow to red. The raw anomaly score distributions as evaluated by the anomaly detector are optionally depicted within the graph elements as histograms in the same color scheme.

The approach has been applied to the distributed JPetStore, our sample application, under the influence of workload generation and fault injection. In a series of 42 experiments, 14 faults out of five categories are applied at various points in the application. Timing behavior information is collected during runtime by Tpmon and written to the file system.

Two experiments have been selected to be used for the examination of the Correlator's abilities, and to optimize its parameters. Three algorithms are tested, and five settings and parameters are varied to each examine their influence on the results. Finally, the optimized parameters are used in combination, and applied to the data of three other experiments.

The following Section 5.1 gives an overview of the positive results of the current work. In Section 5.2, the limits and drawbacks are discussed. Finally, Section 5.3 contains a list of ideas for possible enhancements of the correlation process, and of the evaluation.

5.1 Achievements

The concept works. The Correlator prototype is a highly configurable tool for failure diagnosis based on timing behavior. Although not always pinpointing the correct element of fault injection, at least it declares large parts of the application as *not* containing a fault with high probability, thus reducing the search space for further examinations. The results of the evaluation show that a complex algorithm is able to perform significantly better than a simple aggregation of anomaly scores. The system resource usage of the plug-in during analysis is good compared to other plug-ins that pre-process the data within Tpan.

The Figures 5.1 and 5.2 show examples of the results that can be achieved. Both graphs evolve from the same experiment, where a programming fault has been injected in the operation `CatalogService.getItemListByProduct(String,int,int)`. Due to space restrictions, the deployment contexts for `AccountService` and `OrderService` are omitted from the graphs.

- The graph in Figure 5.1 has been created after applying the *trivial algorithm* that performs aggregation only: The anomaly ratings are accumulated by using a simple unweighted arithmetic mean calculation for each element.

  Although the operation containing the fault is colored in red shade, it is not classified to be the cause of failure. Furthermore, the higher-level elements containing this operation are also classified incorrectly.

- The graph in Figure 5.2 has been created after applying the *advanced algorithm* that tries to use some extra knowledge according to the approach presented in Chapter 3.

  The fault localization is very good: The operation that contains the fault as well as its superior elements are correctly classified and marked as having the highest rating on their respective level. At the same time, all other elements have significantly lower ratings that are reflected in the green to yellow color shades.

For fault injection on component level, an example for a similar progress can be seen in Figure 4.20 on page 79. This examination also confirms the value of the *clearness* rating, introduced in Section 4.1 (pg. 48), that is used for benchmarking the results on the hierarchy levels. As long as the user does not feel confused about the coexistence of three of these ratings, it properly reflects the visual impression of the colored dependency graphs.

5.2 Limitations

In order to get clear results, certain preconditions have to be fulfilled.

- Training data had been collected during a long period of fault-free activity of the application under analysis.
- A failure has occurred, and has been detected in a short period of time.
- The failure is caused by a single fault.
- The fault has an effect on timing behavior.
- The measurements of the timing behavior are correct.
- The anomaly detection works satisfyingly.

During the preparation for the experiments, it turned out that it is hard to construct faults that on the one hand are sufficiently realistic, and on the other hand produce a timing behavior anomaly, and *only* a timing behavior anomaly as far as possible (e.g. no Java exception, no distinct error message, and no system outage). In addition, the effect of the anomaly has to be strong enough to be correctly classified by the anomaly detector, but weak enough to retain a challenge for our studies.

In the examinations performed in Chapter 4, two variants of programming faults have been injected – a method is called twice, and a method call is omitted –, and both are shown to have significant effect on timing behavior, extending or reducing the execution time of the affected operation. On component level, an explicit *anomaly injection* (in contrast to a fault injection) was applied by adding calls to `Thread.sleep(long)` to let the application wait a fixed amount of time. As expected, this fault category is also well detected and classified by our tools.

On the other hand, for the fault injection on deployment context level, our experiments did not yield a fault category that meets the requirements: The effect of the manipulations is non-significant, or completely wrong in most cases as discussed in Section 4.4.2 (pg. 63), so that the correctness for the few remaining cases cannot be ensured. Thus, it stays unclear whether other categories of real-life faults are detected with sufficient quality, and how the analysis performs outside our experiment environment.

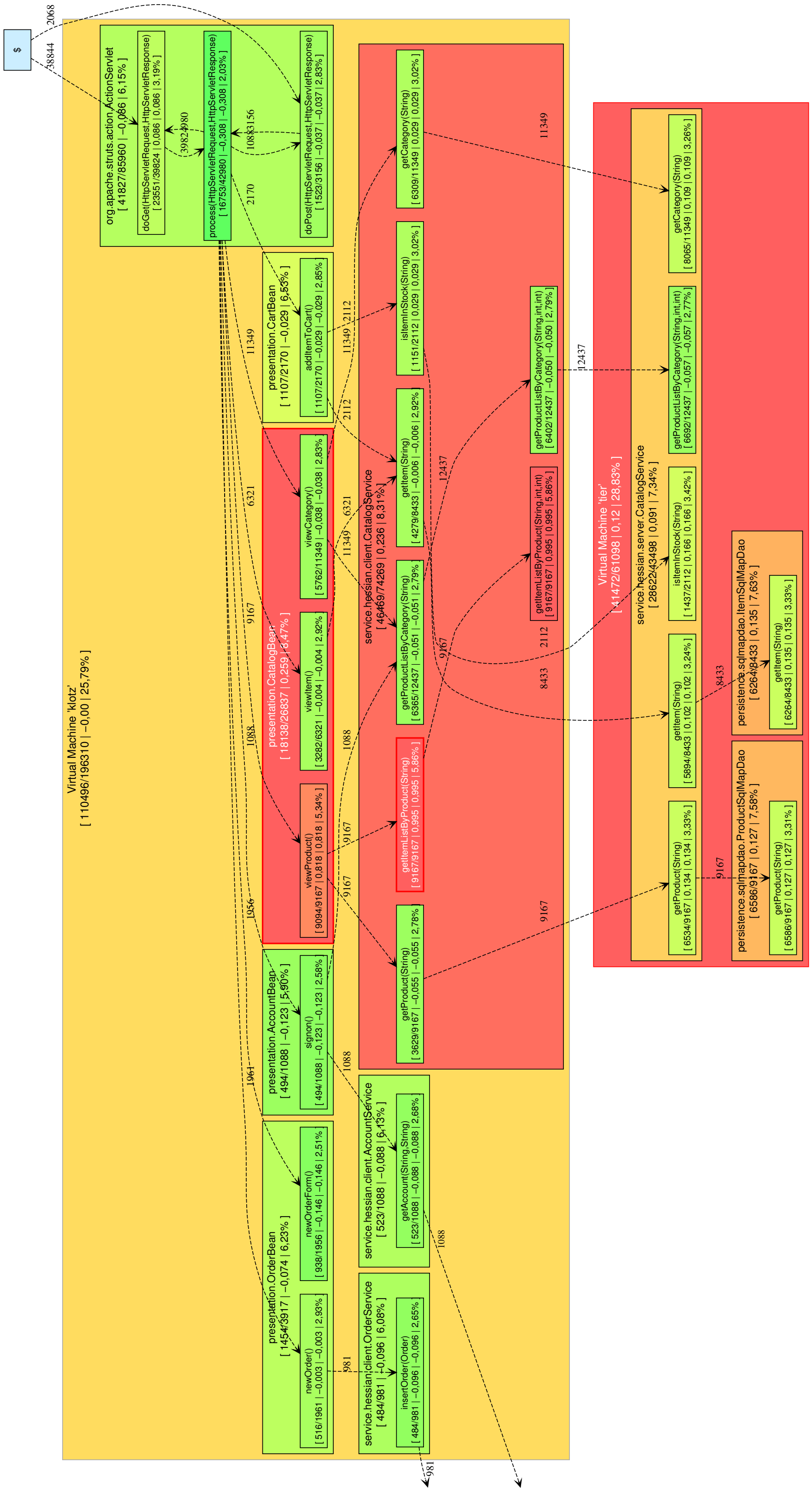


Figure 5.1: Result graph of an analysis using the trivial algorithm. A programming fault has been injected in `getItemListByProduct(String,int,int)`.

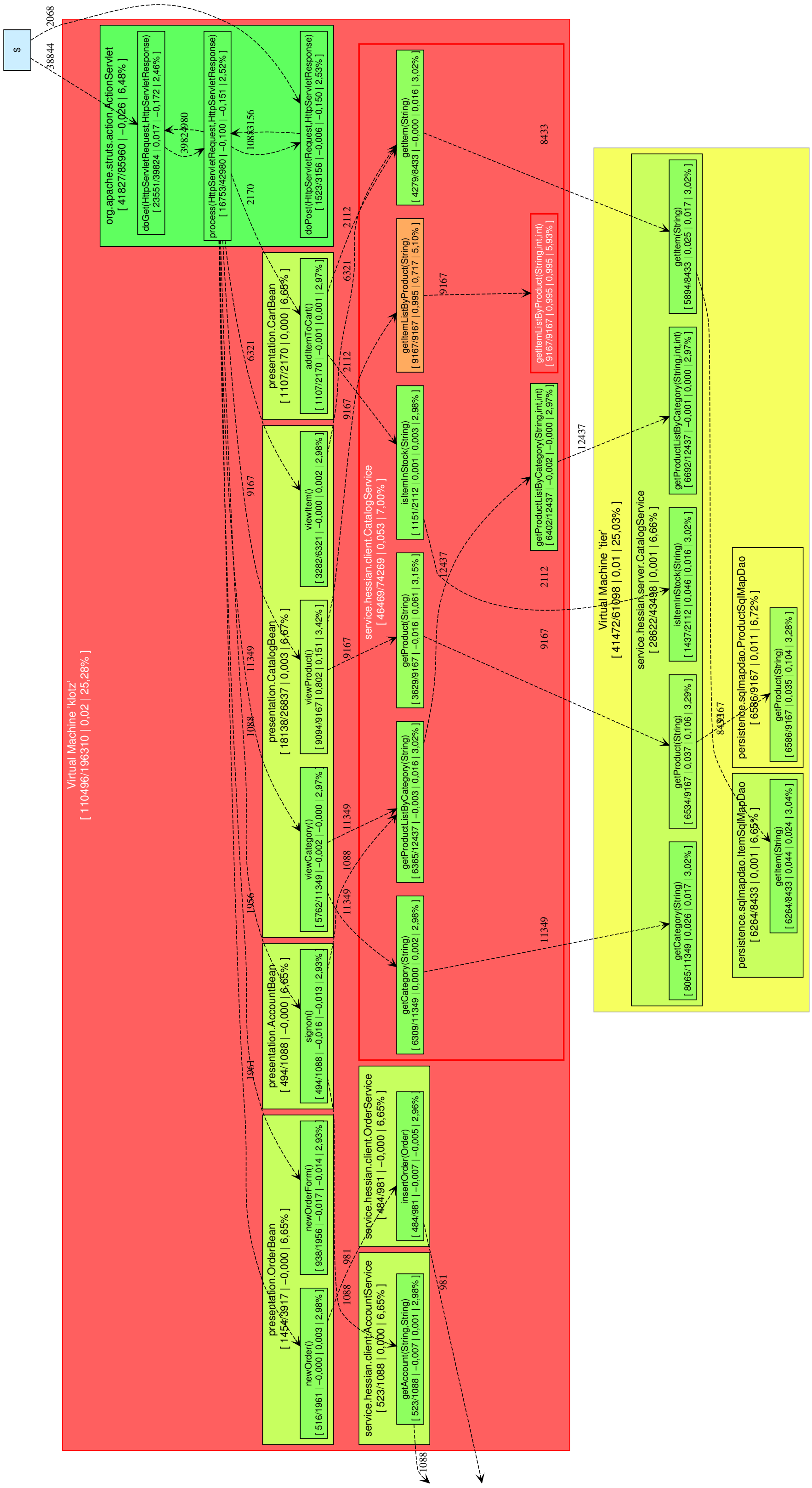


Figure 5.2: Result graph of an analysis using the advanced algorithm. The same experiment data is used as in the previous figure.

Looking at the percentage ratings in general, the results are not as clear as expected. Instead of displaying e.g. “component A contains the cause with 80% probability”, the differences among the elements are usually much smaller, like “7.1% for component X, 6.9% for Y”. This applies to all algorithm variants, and there is no solution yet – if a solution is actually required.

Subject of another observation, especially for our distributed JPetStore and its monitoring instrumentation, is the low cross-linking between the elements: Most of the dependency paths go directly and one-way through the application. Referring to the *advanced algorithm* in particular, during development, the assumption that anomalies propagate fan-shaped towards the top of the structure has been an important part. With the low cross-linking, the related parts of the algorithm are hardly used, because they draw very few conclusions if singular connections up or down an element are detected. Thus, it is assumed that the advanced algorithm cannot show its strengths in the current experiment environment, because not all of its aspects can be thoroughly tested.

Although the Correlator does not rely on *training*, the advanced algorithm seems to take an advantage of that however, due to its various parameters. Depending on the desired precision, e.g. whether the graphs are by a human administrator in any event, or whether the rating functions should be nearly perfect already, an adaption and training of the settings and parameters to a certain environment might improve the results.

It is hard to judge the quality of the anomaly detector. At least for the detection of the programming faults injected in our experiments, it seems to work very well. For the fault injection experiments on deployment context, the results were disappointing. It is supposed that the quality of the Correlator’s results highly depends on the quality of the anomaly detector’s results.

Furthermore, it can be assumed that the examination of timing behavior alone does not suffice for a comprehensive analysis of a failing software system, but it might be a complement to existing diagnosis methods that focus on the observation of other system characteristics.

Due to limited experiment time, the results are restricted to one application and few fault scenarios. It has not been found out whether the “optimized” parameters of the advanced algorithm are suitable for our testing environment only, and how the whole process performs when applied to other architectures.

5.3 Future Work

Besides doing much more and thorough testing and training of the algorithm parameters to special situations, or for universal appliance, a few ideas came up throughout the development of the thesis to enhance the concept.

The following list contains ideas to improve the correlation process.

- A different strategy which have been decided not to implement in the thesis is to *iteratively* apply an attempt for backwards propagation. Even more close to neural networks and cellular automatons, a series of uniform calculations could be repeatedly performed on the structure to step-by-step "reconstruct" the causing situation. The main problem is to decide how often the repetition should run, or to determine a distinct stop condition. Will this lead to a steady state?

- Another perspective within the algorithms would be to not consider the nodes in the calling dependency graphs, but instead the edges – the *messages*.

- In addition to the dependency relations, complex patterns in the time and space of the executions could be analyzed by special *pattern recognition* algorithms, comparable to fingerprinting in intrusion detection systems, or in acoustic scene analysis. For example, with the programming fault experiment No. 3, where an empty list is returned instead of one generated from a method call, it is currently disregarded that in this scenario, a whole operation is *never* executed compared to the training data. This gap should be somehow noticed – maybe under the responsibility of the anomaly detector – to increase the anomaly rating of the operation that otherwise called the missing one.

  Furthermore, a pattern recognition could consider special configurations like "*all* operations within a single deployment context show significant anomalies", or "the number of anomalous connections between two distinct deployment contexts is unusual high".

  The chronology of events, explicitly excluded up to now, could indeed provide evidence for the relevance or irrelevance of reported anomalies. On execution level, perhaps a *burst recognition* could be the basis for a "temporal correlation".

  Instead of having an algorithm developer to learn the meaning and effects of propagation, this might somehow become part of an automated process.

- Pursuing the idea of similarities to neural networks, maybe a good result can be achieved with a feedback mechanism and separately controllable thresholds integrated into our dependency graphs – without complex rules, yet with some training.

- One idea is directed towards a *simulation* approach, as a simpler alternative to pattern recognition. Based on "what if . . . " considerations, a presumed anomaly could be calculated out of the application (not backwards, like in the current prototype, but forwards). Then the number of remaining anomaly appearances is checked and compared with other attempts.

Other ideas tend to improve the evaluation process:

- The problems with the anomalies on deployment context level described in Section 4.4.2 (pg. 63) indicate that the anomaly detector might not work properly. Since this is concentrated on the host `jpet4` ("tier"), some debugging effort might make sense at these two points.
- To check the assumptions about inter-element linkage, a series of experiments with extended monitoring could be set up. Additionally, tests with another application seem to be useful, since the complexity of the JPetStore is not supposed to be a perfect representative for real-life applications.
- Alternative metrics might be developed in order to benchmark the estimation of the Correlator in comparison to the expected result when knowing the real cause of failure. For instance, Gruschke [1998a] uses the length of the path from an event to its cause as a metric for the quality of failure diagnosis.

Up to now, the focus of the efforts is on the inspection of a system *after* a failure has happened. As classified in the introduction, the approach is settled in the category of fault detection and fault removal, but it can also be considered a method of dynamic validation: If used with non-production systems, administrators might use their own workload profiles to perform fault localization in their *testing environment*. Additionally it might be used for regression tests – Myers [2001, p. 14] even claims that "the probability for the existence of further errors in a certain section of a program is proportional to the number of errors already discovered in this section" – where the monitoring would then be concentrated at suspicious components.

Obviously, an extension to a continuous observation and analysis at runtime would be a reasonable advancement.

Finally, an improved user interface would be helpful for evaluation and real-life usage. Instead of exporting the graphs with dot, they could be displayed within a graphical interface along with switches to instantly change the parameters for correlation as well as for presentation. Most of the settings listed in Section B.3 could be implemented as check boxes, radio buttons, or scroll bars.

As a bonus, features like XML import and export would meet the current trends.

Appendix A

Experiment Setup Details

A.1 Experiment Activities

Figure A.1 shows a thorough diagram of all activities around the experiments. The *Environment Setup* at the top of the graph is done only once. The middle section is executed for every *Experiment*, preparing e.g. individual fault injection variants. The centered *Execution* shows the tasks of our main three nested Ant scripts that control one or more experiment runs fully automated. The *Cleanup* phase is done manually again as necessary. Finally, the *Analysis* is done by Tpan, and the activities represent menu items within the console client that can again be controlled through a script.

A.2 Instrumentation of the JPetStore

The following JPetStore methods have been instrumented by Tpmon monitoring probes for the current experiments. There are some other methods that are instrumented, too, but are never called in our setup. The list is ordered alphabetically by the fully qualified class names.

- `com.ibatis.jpetstore.persistence.sqlmapdao.AccountSqlMapDao`
  `getAccount(String username, String password)`
- `com.ibatis.jpetstore.persistence.sqlmapdao.ItemSqlMapDao`
  `getItem(String itemId)`
  `getItemListByProduct(String productId)`
- `com.ibatis.jpetstore.persistence.sqlmapdao.OrderSqlMapDao`
  `insertOrder(Order order)`
- `com.ibatis.jpetstore.persistence.sqlmapdao.ProductSqlMapDao`
  `getProduct(String productId)`
- `com.ibatis.jpetstore.presentation.AccountBean`
  `signon()`
- `com.ibatis.jpetstore.presentation.CartBean`
  `addItemToCart()`

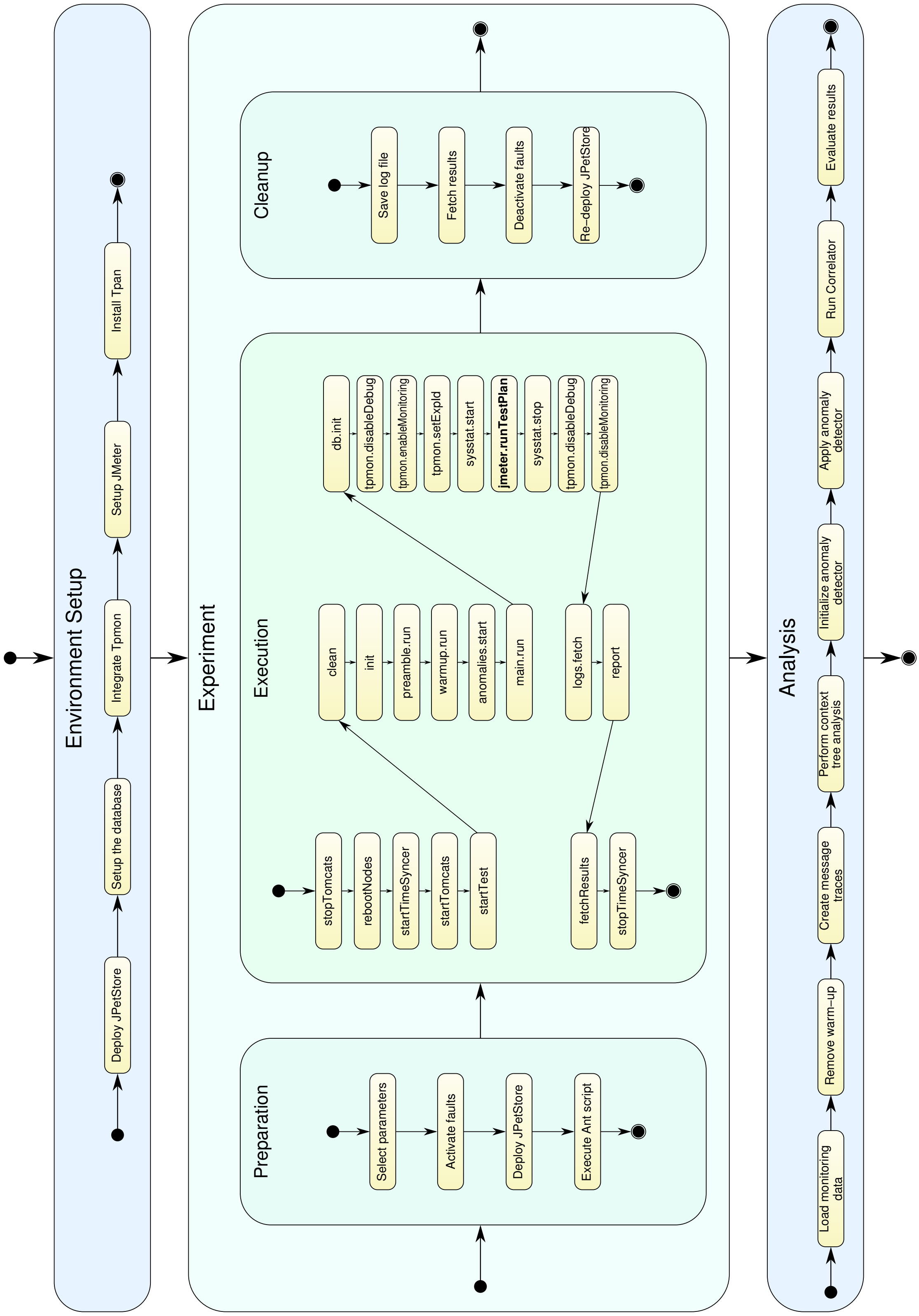


Figure A.1: Activity diagram of the experiments.

- `com.ibatis.jpetstore.presentation.CatalogBean`

  ```
  viewCategory()
  viewItem()
  viewProduct()
  ```

- `com.ibatis.jpetstore.presentation.OrderBean`

  ```
  newOrder()
  newOrderForm()
  ```

- `com.ibatis.jpetstore.service.hessian.client`

  ```
  AccountService.getAccount(String username, String password)
  ```

- `com.ibatis.jpetstore.service.hessian.client.CatalogService`

  ```
  getCategory(String categoryId)
  getItem(String itemId)
  getItemListByProduct(String productId)
  getItemListByProduct(String productId, int skipResults,
           int maxResults)
  getProduct(String productId)
  getProductListByCategory(String categoryId)
  getProductListByCategory(String categoryId, int skipResults,
           int maxResults)
  isItemInStock(String itemId)
  ```

- `com.ibatis.jpetstore.service.hessian.client.OrderService`

  ```
  insertOrder(Order order)
  ```

- `com.ibatis.jpetstore.service.hessian.server.AccountService`

  ```
  getAccount(String username, String password)
  ```

- `com.ibatis.jpetstore.service.hessian.server.CatalogService`

  ```
  getCategory(String categoryId)
  getItem(String itemId)
  getItemListByProduct(String productId, int skipResults,
           int maxResults)
  getProduct(String productId)
  getProductListByCategory(String categoryId, int skipResults,
           int maxResults)
  isItemInStock(String itemId)
  ```

- `com.ibatis.jpetstore.service.hessian.server.OrderService`

  ```
  getNextId(String key)
  insertOrder(Order order)
  ```

- `org.apache.struts.action.ActionServlet`
  `doGet(HttpServletRequest request, HttpServletResponse response)`
  `doPost(HttpServletRequest request, HttpServletResponse response)`
  `process(HttpServletRequest request, HttpServletResponse response)`

A.3 Preparations for Fault Injection

The following list contains the actions that have been done in preparation for the fault injection experiments, matching the activities “Activate faults” and “Deploy JPetStore” in Figure A.1.

- In order to minimize the influence on the experiment environment that is also used for other examinations, the source code of the JPetStore has been copied from “JPetStore-5.0-distributed-Hessian” to “JPetStore-5.0-distributed-Hessian-Injection”.
- Accordingly, in the `build.xml` for deployment, “-injection” is appended to the WAR files to distinguish them from the unmodified files. As a side effect, this allows to run both variants in parallel.
- Likewise, “-injection” is appended to the paths in the `build.properties` so the services can interact with each other.
- After a change in the source code, `ant deploy-all-instrumented` is run in `JPetStore-5.0-distributed-Hessian-Injection` on `jpet1`.

Appendix B

Correlator Plug-in Usage

B.1 Package Content

The package `org.trustsoft.tpmon.logAnalysis.plugins.CorrelatorPlugin` contains the following Java classes in alphabetical order. The descriptions are essentially equal to the Javadoc descriptions.

Algorithm is the abstract base class for all algorithms that are involved in fault localization. Optionally, it can be configured through properties. Algorithm implementations must provide a no-argument constructor to be initialized through `Class.newInstance()`.

AlgorithmAdvanced is an advanced implementation of algorithms to aggregate and correlate information in a dependency graph to estimate the cause of failure. Parameters are controlled through a properties file.

AlgorithmSimple is a simple implementation of functions to aggregate and correlate information in a dependency graph to estimate the cause of failure.

AlgorithmTrivial is the most simple implementation we can imagine: Without correlation, it only performs a simple arithmetic mean calculation on each structure element, independent of any other elements. This seems to be only useful for quick visualization of the "original" state of the application, and not for locating the cause of failure.

Application stores a hierarchically connected structure of all elements (operations, components, deployment contexts) of an application under analysis. The operations itself hold sets of their respective executions with anomalies. Methods are provided to build up the structure, to initiate evaluation, and to present the results.

CorrelatorPlugin is the controlling class of the plug-in, and contains the interface to the environment. There are public methods to (1) build up the dependency graph, (2) integrate anomaly information, (3) evaluate the cause of failure, (4) output the results in text form, and (5) visualize the results as Graphviz dot file.

DistanceAndWeight is a small helper class that stores some attributes for remote anomaly mean calculation to be used in a `Map` which would otherwise require a multi-column table.

DotFactory provides a collection of static methods to compile Graphviz dot elements from `String` attributes and properties. These elements may be compiled to complete dot files externally, or directly exported to various image file formats.

Experiment stores properties and results used for batch runs. The properties are intended to overwrite the default properties.

PresentationDot provides some static methods to present a dependency hierarchy of structure elements in Graphviz Dot form. Parameters are controlled through properties file.

PresentationText provides some static methods to present some structure elements in nice and detailed text form.

PropertiesExtended is an extended version of the standard `Properties`. Additional methods are provided to fetch `Integer` and `Double` objects, and to check them for consistency.

PseudoColor provides methods to manage pseudo colors. Pseudo colors are calculated by mapping the values of data sets to a pre-defined spectrum of colors. For example, a height map could be visualized in grey scale, or a temperature map could be drawn in cold-to-warm colors. In the Correlator, the colors are based on the anomaly cause ratings of the structure elements.

SampleAndWeight is a small helper structure for sample and weight. Two `Double`s are sticked together to allow combined ordering by one of them.

StructureComponent implements unique functionality for components, which work as normal `StructureElement`s otherwise. Components usually have a `StructureDeploymentContext` as parent, they have `StructureOperation`s as children, and they are linked among each other with a *uses* relation.

StructureDeploymentContext implements unique functionality for deployment contexts, which work as normal `StructureElement`s otherwise. Deployment contexts usually have no parent, they have `StructureComponent`s as children, and they are linked among each other with a *uses* relation.

StructureElement serves as a super class for all classes that are part of the dependency structure, and that form the nodes of the dependency graph, respectively. Methods are provided to build up and access the dependency structure. Instances of this class can be automatically sorted by their cause rating.

StructureExecution implements unique functionality for executions, which work as normal `StructureElements` otherwise. Executions usually have a `Structure↩ Operation` as parent, they have no children, and are not linked among each other.

StructureOperation implements unique functionality for operations, which work as normal `StructureElements` otherwise. Operations usually have a `Structure↩ Component` as parent, they have `StructureExecutions` as children, and they are linked among each other with a *uses* relation.

Util provides a bunch of useful static functions for use in different places of the plug-in. Along with various constants, there are methods for debug output, for mathematics, for `String` manipulation, and for file system access.

B.2 Tpan Integration

For communication with the environment – usually Tpan –, the Correlator plug-in has some public methods. Most other methods have package-protected access rights.

B.2.1 Machine Interface

First, the `CorrelatorPlugin` constructor has to be called to initialize the plug-in. It loads the properties file, applies override properties if available, and opens the log file.

In order to load the dependency data of the application under analysis into the plug-in, the methods `processMessageTrace` or `processMessageTraces` can be called in any order. Then `integrateExecutions` is used to load the anomaly information into the structure.

To start the evaluation, execute `evaluateApplication`. Again, override properties are accepted.

It is recommended to run the `check` method afterwards.

To fetch the results, the plug-in provides three methods, one for each hierarchy level, to get `Lists` of the `StructureElements`, sorted in descending order of their cause rating:

```
List<StructureElement> getOperationsOrdered()
List<StructureElement> getComponentsOrdered()
List<StructureElement> getDeploymentContextsOrdered()
```

The method `getResultsCSVLine` can be invoked to get some hard-coded results for appending a line to a CSV file. Usually, it is used in combination with the experiment batch file mechanism.

On finish, `closeLogFile` should be called.

B.2.2 Human Interface

Three groups of methods are focused on providing the results in a human readable form.

1. The results can be printed to an arbitrary `PrintStream` – e.g. `System.out` for printing on command line interface – via `printResultTable` and `printComponents↩ SortedByCauseRating`.

2. The method `writeDotFile` instantly exports the dependency structure to the specified file in Graphviz dot syntax, using the presentation properties from the plug-in.

3. To create image files, `exportGraphicFromDotFile` can be called with a filename and the shortened common file type that is known to dot – e.g. “ps” for PostScript. The export is done by invoking the dot command in a shell.

   If more than one image file type is to be created from the same results, the wrapper method `exportGraphicsFromDotFile` might be useful that takes an arbitrary number of file types in a colon-separated list.

   Finally, for creating large groups image files via an experiment batch file, `writeGraphicsFiles` uses a file prefix (followed e.g. by the experiment identifier) as well as the list of image file types from the plug-in properties.

B.3 Correlator Configuration

The Correlator plug-in can be thoroughly configured through Java `Properties`. These parameters are used as long as no overrides are specified via special experiment instructions.

B.3.1 General

The `correlator.properties` contains some basic parameters for experimentation and debugging. The detail level for the `PrintStream` output, and for the log file can be set, and the file name to append the log messages to. Percentage values can be specified that are used to get a first impression after loading the data – these are cosmetic only and may become deprecated. An important setting is the choice of the correlation algorithm: The name of an existing Java class has to be specified that will be dynamically loaded. For the output, an image file prefix can be set, as well as a colon-separated list of default file types for export.

B.3.2 Presentation

In the `presentation.properties`, a list of visual features for the dependency graphs can be switched on or off. The graph may contain a caption, an explanation text, execution anomaly score histograms, pseudo color shades reflecting the cause ratings, a legend explaining the colors, details of each rating (mainly for debugging purpose), edge weights, deployment context elements, component elements, and operation elements. Most of these can be arbitrarily combined. For instance, a graph may contain deployment contexts and operations, but no components.

For the color shades, there is an option to stretch them to use the full spectrum from green to red to visually rise the contrast. Otherwise it may happen that all elements are yellow, if the anomaly scores predominantly show “normal”.

The edge weights can be switched between *absolute* (number of executions) and *relative* (percentages).

To save space, a prefix can be specified to be removed from each element name. For instance, in our current examinations, all elements start with the package name `com.ibatis.jpetstore`.

For the histograms, the dimensions can be specified in pixels, as well as the file type (as extension, like “png”).

The remaining settings concern the attributes of the elements. There are font families, font sizes, font colors, frame colors, and some special colors for highest-rated elements, and for the “root” element.

B.3.3 Algorithm

Currently, only the advanced algorithm is configured via `Properties`. The other algorithm variants do not need them, although the simple algorithm is prepared to use them. The following five settings from the `algorithm.advanced.properties` equal those that have been examined in Section 4.5 (pg. 67).

1. The input-output-relation factor can be set as a decimal. A value smaller than 1 means that the incoming connections have more influence, and vice versa, while a value of 1 means this feature is practically disabled.

2. The edge weight method for correlation on operation level can be switched between absolute and relative.

3. Three mean methods (power mean, median, and maximum) can be selected independently for aggregation on all three hierarchy levels, and for correlation on operation level.

4. For each of the four power mean possibilities, an independent exponent can be set to emphasize outliers, or to reduce their influence. A value of 1.0 means to practically switch off the power mean, equating it with the arithmetic mean.

5. In order to enhance or weaken the influence of distance in correlation on operation level, the “neighborhood mean distance exponent” is specified that is obviously deactivated by setting it to 1.0 again.

B.4 Experiment Instructions

Before the start of the Correlator plug-in, the surrounding method in Tpan tries to load experiment instructions. An experiment in this context consists of a name at least, and can have a list of override parameters. The name will be used for writing the file names of the dot graphs as well as the results CSV files. The parameters, if the same keys are used as in the properties files, can overwrite almost *any* properties. This is useful to repeat an evaluation with the same data using slightly different parameters, e.g. special algorithm configurations, or different graph options.

In the current implementation, first one experiment is loaded from the properties of Tpan. If that fails, the CSV-like experiment batch file is loaded, that may contain any number of experiments, one per line. Figure 3.18 (pg. 46) shows an example.

If there is more than one line in the batch file, the Correlator plug-in alone is repeatedly run for each experiment without having to reload the data, or to repeat the pre-processing by other plug-ins. Details of the syntax can be found inside of the `correlator-experiments.csv.example` file in the root directory of Tpan.

If no instructions are found, a default experiment without override parameters is created and loaded with the Correlator plug-in.

B.5 New Algorithms

Since the Correlator plug-in is equipped with a class loader, new algorithms could be loaded at runtime.

To create a new algorithm, the abstract `Algorithm` class has to be extended. This way, it has to implement the four evaluations methods – aggregation on all three hierarchy levels, and correlation on operation level. Basic functionality is available to manage the name, and to load and check the optional properties. Additionally, a method is included that performs a simple arithmetic mean calculation of the cause ratings of the specified `StructureElement`'s children. It is recommended to start with a copy of `AlgorithmSimple` though, and to switch its parts as necessary.

Bibliography

Stephen Adler. The Slashdot Effect, an analysis of three internet publications. In *Linux Gazette*, volume 38, March 1999. URL `http://linuxgazette.net/issue38/adler1.html`. Last visited August 10, 2008.

Manoj K. Agarwal, Karen Appleby, Manish Gupta, Gautam Kar, Anindya Neogi, and Anca Sailer. Problem determination using dependency graphs and run-time behavior models. In Akhil Sahai and Felix Wu, editors, *Proc. of 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004)*, volume 3278 of *Lecture Notes in Computer Science (LNCS)*, pages 171–182. Springer, November 2004.

ArchWiki. Cpufrequtils – Arch Linux Wiki. Published on-line, July 2008. URL `http://wiki.archlinux.org/index.php/Cpufrequtils`. Last visited August 10, 2008.

Algirdas Anthony Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, San Diego, California, USA, June 2007. ACM. URL `http://findbugs.cs.umd.edu/papers/FindBugsExperiences07.pdf`. Last visited August 10, 2008.

Richard M. Bailey and Richard C. Soucy. Performance and availability measurement of the IBM information network. *IBM Systems Journal*, 22(4):404–416, 1983.

Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *From Natural to Artificial Swarm Intelligence*. Santa Fe Institute studies in the sciences of complexity. Oxford University Press, New York, NY, USA, September 1999.

Dominik Brodowski and Sebastian Henschel. Linux/ACPI – documentation: The /proc/acpi/processor subdirectory. Published on-line, 2002, 2004. URL `http://acpi.sourceforge.net/documentation/processor.html`. Last visited August 10, 2008.

David W. Cantrell and Eric W. Weisstein. Power mean. Published on-line, December 2003. URL `http://mathworld.wolfram.com/PowerMean.html`. From MathWorld – A Wolfram Web Resource. Last visited August 10, 2008.

Thilo Focke. Performance Monitoring von Middleware-basierten Applikationen. Master's thesis, Carl von Ossietzky Universität Oldenburg, Germany, March 2006.

Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*, January 2006. URL `http://graphviz.org/Documentation/dotguide.pdf`. Last visited August 10, 2008.

Simon Giesecke, Matthias Rohr, and Wilhelm Hasselbring. Software-Betriebs-Leitstände für Unternehmensanwendungslandschaften. In *Proceedings of the Workshop "Software-Leitstände: Integrierte Werkzeuge zur Softwarequalitätssicherung"*, volume P-94 of *Lecture Notes in Informatics*, pages 110–117. Gesellschaft für Informatik, October 2006.

Sebastien Godard. Sysstat. Published on-line, 2008. URL `http://pagesperso-orange.fr/sebastien.godard/`. Last visited August 10, 2008.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, January 1989.

Boris Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98), Newark, DE, USA*, October 1998a.

Boris Gruschke. A new approach for event correlation based on dependency graphs. In *Proceedings of the 5th Workshop of the OpenView University Association*, April 1998b.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1999.

Horst-Joachim Hoffmann. Systemantwortzeiten als Aspekt der Software-Ergonomie und der Wirtschaftsinformatik (book review). *c't Magazin für Computertechnik*, 25:240–240, November 2006.

Intel Corporation. *Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide*, July 2008. URL `http://download.intel.com/design/processor/manuals/253668.pdf`. Last visited August 10, 2008.

Kai S. Juse, Samuel Kounev, and Alejandro P. Buchmann. PetStore-WS: measuring the performance implications of web services. In *Proc. of the 29th International Conference of the Computer Measurement Group (CMG) on Resource Management and Performance Evaluation of Enterprise Computing Systems – CMG2003*, pages 113–123, December 2003.

Emre Kiciman and Armando Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027–1041, September 2005.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, and New York, June 1997. Springer-Verlag, LNCS 1241.

Charles M. Kozierok. The PC guide disk edition. Published on-line, May 2001. URL `http://www.pcguide.com/disk/index.htm`. Site Version: 2.2.0 - Version Date: April 17, 2001.

Heiko Koziolek. The role of experimentation in software engineering. In Wilhelm Hasselbring and Simon Giesecke, editors, *Research Methods in Software Engineering*, volume 1 of *Trustworthy Software Systems*, chapter 2, pages 11–37. GiTO-Verlag, Berlin, July 2005.

Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. IEEE Computer Society Press and McGraw-Hill, 1996.

Daniel Menascé and Virgilio Almeida. *Capacity planning for Web services: metrics, models, and methods*. Prentice Hall, September 2001.

Thomas M. Mitchell. *Machine Learning*. McGraw Hill, 1st edition, March 1997.

Glenford J. Myers. *Methodisches Testen von Programmen*. Oldenbourg Wissenschaftsverlag, 7th edition, 2001.

Matthias Rohr. *Workload-sensitive Timing Behavior Anomaly Detection for Automatic Software Failure Diagnosis*. PhD thesis, Department of Computing Science, University of Oldenburg, Oldenburg, Germany, 2008. *work in progress*.

Matthias Rohr, André van Hoorn, Nina Marwede, Simon Giesecke, Wilhelm Hasselbring, Thilo Focke, and Johannes-Gerhard Schute. Failure diagnosis using timing behavior anomaly analysis under varying workload intensity. *In preparation*, 2008a.

Matthias Rohr, André van Hoorn, Jasminka Matevska, Nils Sommer, Lena Stoever, Simon Giesecke, and Wilhelm Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In Claus Pahl, editor, *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*, pages 80–85, Anaheim, February 2008b. ACTA Press.

Peter Schwenkenberg. Auswirkung von Programmierfehlern auf Softwarezeitverhalten. Master's thesis, Carl von Ossietzky Universität Oldenburg, Germany, August 2007.

Ian Sommerville. *Software Engineering*. Pearson Studium, Pearson Education Deutschland GmbH, München, Germany, 6th edition, 2001.

Lena Stöver. Evaluation dienstbezogener Abhängigkeiten in komponentenbasierten Systemen. Individuelles Projekt, November 2007. Carl von Ossietzky Universität Oldenburg, Germany.

Tom's Hardware Team. What happens when the CPU cooler is removed? Published on-line, 2001. URL `http://video-de.tomshardware.com/video/iLyROoaftlwq.html`. TG Publishing AG, Munich Lab, Germany. Last visited August 10, 2008.

Gabriel Torres. Pentium 4 thermal throttle. Published on-line, March 2005. URL `http://www.hardwaresecrets.com/article.php?id=104`. Last visited August 10, 2008.

André van Hoorn, Matthias Rohr, and Wilhelm Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *Performance Evaluation – Metrics, Models and Benchmarks: Proceedings of the SPEC International Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 124–143, Heidelberg, June 2008. Springer.

Donald Voet, Judith G. Voet, and Charlotte W. Pratt. *Fundamentals of Biochemistry: Life at the Molecular Level*. John Wiley & Sons, New York, USA, 2nd edition, March 2005.

Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002. Last visited August 10, 2008.

Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time Will Tell: Fault localization using time spectra. In *Proceedings of the 13th international conference on Software engineering (ICSE '08)*, pages 81–90. ACM, May 2008.

Acknowledgement

I like to thank the modern medicine, and the brave doctors,
who gave me hope and reason to live on.

I like to thank my advisors for their patience, and the exhaustive feedback
on my word screwings and sentence creativities.

“Uhhm... can you spare a minute? Got another question on bash scripting...”

Thanks to my friends at Shio-Sai for giving opportunities to meet interesting people.

A bow goes to Shihan Lauri Jokinen for providing new insights.

Last not least, a big hug to my family and friends for their support.

Relax.

Declaration

This thesis is my own work and contains no material that has been accepted for the award of any other degree or diploma in any university.

To the best of my knowledge and belief, this thesis contains no material previously published by any other person except where due acknowledgment has been made.

Oldenburg, August 14, 2008

Nina Sophie Marwede