Automatic Diagnosis of Software Functional Faults by Means of Inferred Behavioral Models

PhD Dissertation by:
Fabrizio Pastore

Advisors:
Prof. Mauro Pezzè
Dott. Leonardo Mariani

Supervisor of the Ph.D. Program:
Prof. Stefania Bandini

Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Dottorato di Ricerca in Informatica

XXII edition

to me, my parents, my girlfriend

Contents

List of Figures

List of Tables

Listings

Introduction

Software failures have a relevant impact on today economy. The US National Institute of Standards and Technology (NIST), estimated that software failures cost US economy $59.5 billion annually [95]. Different experiments [82, 48] and catastrophic events [69] indicate that functional and integration faults remain one of the main issues software engineers have to focus on.

The adoption of extensive validation and verification activities during development can improve the quality of the developed software but it cannot guarantee the removal of all the faults in a system. Moreover, validation and verification techniques often require source code or specifications to be applied and thus cannot be used with many software systems, like those that integrate *Off The Shelf* components, which are usually provided without source code or with incomplete specifications.

Since many faults still remain undetected till the software is used in the field, the adoption of techniques that facilitate and reduce the time necessary to diagnose faults can reduce the costs caused by system failures and downtime. In addition, since fault diagnosis is one of the activities with the greatest impact on software development and maintenance costs [128], the adoption of automated fault diagnosis techniques can reduce also the costs of the whole software development process.

Existing automated and semi-automated fault diagnosis techniques present many limitations. *Interactive debugging* techniques require a lot of developers effort, because the whole debugging process is completely manual [21, 6, 81]. *Slicing* [127] and *spectra based* techniques [61, 7] pinpoint the faulty code but do not indicate developers why the code is faulty. These techniques do not help developers in the understanding and correction of the problem, but only in the localization of the fault. *Automated debug-*

ging techniques [130] are typically effective with state-based faults only. *Anomaly detection techniques* [103, 38], instead, help developers in understanding the failure causes by identifying anomalies in the behavior of the system during the failing execution. Unfortunately these techniques force developers to inspect many anomalies, including several ones not related with the fault, before understanding and fix the problem.

This PhD Thesis presents a framework for the diagnosis of functional faults that advances the state of the art by:

- Identifying and correlating different kinds of anomalies: data anomalies, events flow anomalies, and data flow anomalies [37, 88], thus effectively describing faults with heterogeneous causes.
- Automatically filtering false positives [89], thus permitting developers to inspect only anomalies useful to debug the application.
- Presenting behavioral anomalies in a structure that explicitly shows the cause effect relationships between multiple anomalies [89], thus giving developers a map for the inspection of the anomalies which simplify the diagnosis activity.
- Automatically interpreting anomalies [20], thus permitting developers to immediately diagnose the fault.
- Evaluating the solution with multiple third-party and industrial case studies, thus demonstrating the effectiveness of the solution.

The PhD Thesis is structured as follows. Chapter 1 describes the impact of functional faults on modern software systems and presents the existing functional faults diagnosis techniques. Chapter 2 provides an overview of the diagnosis framework defined in this PhD Thesis. Chapters 3, 4, 5 describe in details the diagnosis phases supported by the framework: capture legal behavior, identify runtime misbehavior, and diagnose faults. Finally Chapter 6 presents the results that we obtained by using the framework to diagnose real faults affecting multiple software systems.

Chapter 1

Functional Faults Diagnosis

1.1 Impact of functional faults

Program failures have a relevant impact on today economy: a study released in June 2002 by the US National Institute of Standards and Technology (NIST), states that "Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross national product" [95]. The document reports that almost half of the faults that affect software systems nowadays are identified after they are deployed in field.

Functional and integration faults remain one of the main issues software engineers have to focus on. A survey conducted on 139 bug reports states that half of field failures affecting the Apache Http Server [11], the GNOME desktop environment [54], and the MySql database [117] are due to functional problems. While recent experiments [48] and catastrophic events [69] stress the fact that integration faults are hard to identify at testing time even in well tested systems.

To face the impact of software faults, the European Commission in May 2009 proposed to consider software vendors liable for damages caused as a result of defects in their software products [35]. This news caused concern among software developers, either they were big companies or single programmers [1, 65, 66]. Software developers are afraid of such decision

because they know that despite improved software verification and validation activities could reduce the number and costs of field failures [107], the complete removal of faults is almost impossible and making developers responsible for the the consequences of software faults could impact much on their finances.

The extensive adoption of verification and validation techniques can improve the quality of the developed software but they still cannot guarantee the absence of faults. Many verification and validation techniques require source code or specifications to be successfully applied. Unfortunately these requirements limit the applicability of these techniques, because modern systems often integrate third party Off The Shelf components provided without source code or with incomplete specifications. Furthermore these techniques often do not scale well or present technological limitations (for example Java Path Finder, a well known model checker for Java programs, cannot completely analyze programs that use many Java 6 libraries [93]). The practical applicability of many verification and validation approaches remains limited, thus permitting to identify only a small portion of faults during development.

Despite the effectiveness of the software validation process, faults identified during development, as well as the ones detected after deployment, are often difficult to localize and diagnose [100, 128]. Software developers diagnose faults by reproducing failing executions through a symbolic debugger or by inspecting logs recorded during failing executions looking for unexpected runtime events or anomalous application states that occurred during the failing execution [128]. The effort required in this process is relevant: logs can be huge and hard to inspect, while debuggers often require multiple executions of the applications.

The diagnosis process can be even worse in case of field failures. Privacy reasons could prevent final users from providing the logs, while the replication of the failure could be difficult as well: users seldom provide all the information necessary to replicate the failure, and certain environmental constraints could not be reproduced in the development environment.

Automated and semiautomated solutions that help developers in the diagnosis of faults are thus necessary to reduce debugging effort. Different automated localization techniques have been proposed recently, each one presenting benefits and limitations.

Many of these techniques require source code, need the re-execution of

the failing run in the development environment, or present a relevant impact on performance thus resulting to be not applicable to analyze failures in the field at runtime. Furthemore, several techniques can be effective to address unit faults, but not effective when addressing integration faults.

Following sections analyze the benefits and limitations of the different techniques.

1.2 Automated fault diagnosis techniques

We group existing automated and semi-automated fault diagnosis approaches into six groups according to the information they use to diagnose faults:

- log file analysis techniques compare log files generated by the application with specifications of legal or faulty behaviors to identify suspicious log messages;
- interactive debugging techniques help developers searching for faults by providing tool, namely symbolic debuggers, that permit to manually inspect and alter program executions at runtime;
- automated debugging techniques compare faulty and correct executions to identify the program inputs and variable values that lead a program to failure;
- slicing techniques can be applied to analyze the program source code to identify the instructions that influence the erroneous result of the program;
- spectra based techniques compare the set of program statements executed during correct an failing executions to identify the faulty instructions;
- statistical sampling techniques sample program executions directly in the field to identify faults in case failures occur;
- anomaly detection techniques infer models of the legal application behavior to automatically characterize faults in terms of deviations from the expected behavior.

These techniques produce different benefits and present several requirements. In order to provide an overview of the existing fault localization

Technique	Requirements				
	Specifications	Source code	# passing runs	# failing runs	Replicate failure
Automated log analysis	Yes	No	0	1	No
Interactive debugging	No	Yes	1	0	Yes
Automated debugging	No	Yes	1	1	Yes*
Slicing	No	Yes	1	0	No
Spectra based	No	Yes	1..N	1..N	Yes
Statistical sampling	No	Yes	N	N	No
Anomaly detection	No	No	1	N	Yes*

*Certain techniques run directly in the field and avoid the reproduction of the failure.

Table 1.1: *Requirements of fault diagnosis techniques.*

techniques we show, in Tables 1.1 and 1.2, for each kind of technique: its requirements, and the developers effort saved expressed in terms of the activities developers are not required to do in order to diagnose a fault if that particular technique is adopted.

Almost all the approaches require the existence of one or more legal executions. This requirement does not constitute a real limitation because software engineers often use automated tools to test their systems and thus they are able to replicate test executions with only a limited effort.

Almost all the approaches require that developers reproduce the failure in the development environment. This requirement can preclude the localization and fix of faults occurring on machines not directly accessible by developers. *Statistical sampling* and *automated log file analysis* techniques do not require the replication of the failing execution but they present several limitations that reduce their effectiveness. *Statistical sampling* techniques need multiple faulty executions in order to provide meaningful results, for this reason their effectiveness is limited in case faults manifest only in one or few failing executions. *Automated log analysis* techniques

	Developers' effort saved			
Technique	Identify failure context	Identify misbehavior	Determine expected behavior	Identify the code to fix
Automated log analysis	Yes	Yes	Yes	No
Interactive debugging	No	No	No	No*
Automated debugging	Yes	Yes	No	No**
Slicing	No	No	No	Yes
Spectra based	No	Yes	No	Yes
Statistical sampling	No	Yes	No	Yes
Anomaly detection	Yes	Yes	Yes	No**

* Interactive debugging just provides means that help developers in following the application execution flow to identify the faulty code.

** Anomaly detection and automated debugging identify portions of code that present behaviors that cause the failure, and can be close to the faulty code.

Table 1.2: *Effort reduction in terms of debugging activities each technique save developers.*

instead often require specifications of legal or faulty behaviors to capture suspicious log messages in log files. This requirement makes log analysis approaches not applicable in many contexts because such specifications are seldom available.

Even when the failure can be easily reproduced in the development environment some of the techniques still result of limited applicability: *slicing, spectra based, and statistical sampling* techniques in fact often require source code to be succesfully applied thus precluding the analysis of systems that integrate *Off The Shelf Components* provided without source code.

If we consider the completeness of results, we can identify two groups of techniques: the ones that provide an high level description of the problem by identifying the deviant behavior that lead to a failure, which are *log analysis*, *automated debugging* and *anomaly detection*, and the ones that focus on the identification of the faulty instructions but that do not provide any description of the misbehavior thus not helping developers in the correction of the fault, which are *slicing, spectra based, and statistical sampling*. *Interactive debugging* techniques are the ones that require more developers effort. Interactive debugging techniques just provide means to interact with the application to identify the conditions under which it fails but require developers to drive the whole debugging activity and identify the fix manually. *Automated debugging* techniques automate some of these operations by automatically identifying the minimal set of program values that trigger the failure. Unfortunately knowing the conditions under which a software fails is not sufficient to fix the fault, for this reason even when adopting automated debugging techniques developers still need to figure out what is the expected behavior of the application in order to correct the problem. *Anomaly detection* techniques provide information about the faulty application behavior in terms of violations of models of the expected application behavior. The identification of violations of behavioral models help developers in understanding both the problem and the possible solution. Unfortunately in many complex systems the inspection of the behavioral anomalies can be expensive thus resulting of scarse effectiveness. Anomaly detection techniques in fact often identify spurious anomalies not related with the fault thus forcing developers to inspect many false alarms before being able to identify the real cause of a failure. Moreover developers often need to identify the cause-effect relations between different anoma-

lies in order to completely understand the application misbehaviors and then diagnose the fault. *Automated log analysis* techniques provide useful contextual information too, in fact like anomaly detection techniques they often rely on models that describe the valid behavior of the applications. Unfortunately in case of log analysis models are often design and maintained manually by developers, and the costs of model maintenance often impact on the quality of the models thus reducing the effectiveness of these techniques.

Anomaly detection techniques provide a description of the problems in terms of models deviations but leaves developers the task of identifying the code to fix. Other techniques, *slicing, spectra based, and statistical sampling*, instead automatically identify sets of suspicious instructions. Unfortunately the provided result is often imprecise, and developers need to inspect multiple lines of code before identifying the faulty ones. Furthermore these techniques do not provide any additional information that permits to understand the conditions under which the failure occur. The absence of contextual information make these techniques of limited usefulness for the diagnosis of non trivial faults which are usually difficult to detect through code inspection.

The diagnosis framework described in this PhD Thesis overcomes the limitations of the existing techniques and advances the state of the art by:

- identifying software behavioral anomalies and thus helping developers in understanding both the problem and the possible solution [37, 88];
- automatically filtering false positives thus helping developers to focus on anomalies related with the fault [89];
- automatically correlating anomalies and guiding developers in the inspection of the anomalies thus reducing the time developers require to inspect the anomalies reported [89];
- generating automated interpretations of the anomalies observed to simplify the analysis of the problem [20];
- permitting the analysis of software executions both in the field or in the development environment [89, 88].

Following sections describe the benefits and pitfalls of the different techniques more in detail, while Chapters 2 to 5 present the details of the diagnosis framework presented in this PhD Thesis.

1.3 Automated Log File Analysis

In case software failures occur in the field and the application generates logs of the execution, the first activity performed by developers to diagnose the fault is the inspection of the logs.

The information usually recorded in log files can be very coarse, application often record only information about the main tasks executed like the starting and stopping of a service in a web server, thus resulting of limited usefulness to diagnose faults. However log analysis is still of practical usage in particular cases, for example to identify configuration problems affecting the bootstrap phase (during which many configuration messages are logged), or to identify anomalies in the sequences of tasks executed in long runs (for example the managing of user requests in a a web server).

Log files typically contain data about the program executions: name of the tasks executed and parameters that characterize the specific task activity.

In case runtime data are massively collected from executions, the size of log files can grow fast [91]. This is particularly true when logging collects not only the observed event sequences, but also parameters associated with events. For instance, many e-commerce applications trace IP numbers and URLs of the accessed Web pages to create user profiles and recognize the actions related to the failure [27].

Size and complexity of logs seldom make manual inspection cost-effective. In addition, system administrators and developers usually do not have a complete knowledge of target systems, thus a large amount of time may be necessary to manually isolate suspicious event sequences.

In order to reduce the effort required to analyze log files, automatic techniques have been proposed. Three are the approaches adopted for the analysis of log files: technique that use specifications, techniques that identify known faulty events and techniques that identify anomalies in the logged data. This section focuses on approaches specific for log files analysis, for this reason anomaly detection techniques, which can be adopted to analyze different kinds of data not only to log files, are presented in Section 1.9.

Specification based-techniques match events in log files with formal specifications that describe legal event sequences [10]. The recognized anomalies, i.e., event sequences not accepted by specifications, are presented to testers. These techniques have the important benefit to present only and all the problems that can be detected from logs (if the specification is sound and complete). Unfortunately, complete and consistent specifications are expensive to be produced and maintained. Thus, suitable specifications are seldom available and it is rarely possible to apply such approaches.

Expert systems do not require complete specifications to identify suspicious sequences, but require user-defined catalogs that describe the patterns of events that are commonly related to failures [68]. Since the commonly used log formats (e.g. syslog [86], java simple log format [116], uniform log format [115]) include unstructured data, expert systems for log file analysis require to be tailored with user-defined regular expressions to analyze and recognize interesting event types [75, 121, 59]. The usually large number of possible event types makes the definition and maintenance of such regular expressions an expensive and error prone task [102]. Similarly to expert systems, symptom databases use known patterns of illegal event sequences to detect failure causes [31]. Unfortunately, maintaining symptom databases up-to-date is expensive and their effectiveness is limited to well-known and documented problems, and cannot help in case of undocumented issues.

1.4 Interactive debugging

Debugging is the activity of identifying the program instructions that contain the fault. Traditionally debugging activities are based on the usage of a symbolic debugger, a tool that permits developers to interact with a failing execution by setting breakpoints, inspecting program variables and the stack trace. Symbolic debuggers are useful to support debugging activities but do not automate the diagnosis and developers often have to execute the application multiple times before being able to identify the faulty instructions.

In order to reduce debugging costs, researchers proposed techniques that record the state of the application at certain break points to permit symbolic debuggers to step backward in execution avoiding the restart of

the application and save developers time [21, 6, 81].

Although such approaches reduce debuggers effort they still require that failures can be reproduced in the development environment thus not helping the diagnosis of field failures which still remains a critical activity. *Omniscient Debugging* (ODB) [81] is the only debugging technique that permits to fully record a field execution and reproduce it in the development environment. Unfortunately the complete recording of long runs consumes too much resources to be applicable. To overcome this limitation different captures and replay technique limits the amount of data recorded by recording only certain information. The ADDA technique for example replicates failing execution that depends on the environment [33]. The technique traces a failing execution by recording the order of reads and writes to files and streams and the data exchanged and then uses these data to replicate the same execution in the development environment. Other techniques instead focus on the monitoring and reproduction of behaviors of specific components [73], or on the replication of the failing method call only [17].

Interactive debugging still remains the primary choice developers adopt to identify faults. Unfortunately despite the advances to capture field executions and perform backward runs, interactive debugging relies completely on developers intuition and costs much. Approaches that automate debugging activities are needed to reduce developers effort and costs. Following section describes the advances in automated debugging.

1.5 Automated Debugging

Automated debugging techniques automate the activities commonly performed by developers during debugging providing developers a set of differences between failing and faulty executions that permit to diagnose the fault.

In [129] Zeller introduced the *Delta Debugging* algorithm to identify the smallest set of changes that caused a regression in a program. The technique require the following informations: a test that fails because of a regression, a previous correct version of the program and the set of changes performed by programmers. The Delta Debugging algorithm works in this way: it iteratively applies different subsets of the changes to the original program to create different versions of the program and identify the failing

ones. The minimal set of changes that permit to reproduce the failure is reported to developers.

Delta Debugging has been applied in different contexts. The most effective approach for fault localization is presented in [130]. Given two executions of a program, one failing one not, the algorithm identifies, with the help of a debugger, a set of differences between the memory graph of the program in different execution states (for example after the start, in the middle of the execution and before the failure). The algorithm repeats a legal execution and alters the state of the program by changing the value of program variables with values in the difference set till the minimum set of variables that lead to the failure is found. The approach completely automate the debugging process usually performed by developers manually, and leave developers the task of understanding what is causing these differences.

Other automated debugging approaches present adaptations of the Delta Debugging algorithm to identify program inputs leading to failures [32, 92, 131] and to identify faulty code portions not only in case of regression faults [114].

Experimental evaluations underline the effectiveness of automated debugging in helping developers during debugging. Unfortunately a few limitations still charachterize these approaches. The first limitation is that failing runs must be reproducible as in traditional manual debugging: this prevents the technique to overcome the difficulties that developers have when failures manifest in the field. The second limitation is due to the fact that automated debugging techniques compare a pair of programs that differ for their versions or for the parameters passed: in case the programs are very different the results reported to developers will include a lot of elements (for example a huge list of differing variables or program inputs) thus reducing the effectiveness of the technique.

1.6 Slicing

Program slicing was first introduced by Weiser in 1979 [124] as a decomposition technique that automatically extracts the program statements relevant to a particular computation. A program slice consists of the parts of a program that potentially affect the values computed at some point of interest referred to as a slicing criterion. Typically, a slicing criterion consists

of a pair <p, V>, where p is a program point and V is a subset of program variables. The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C are called the program slice with respect to criterion C.

Program slicing approaches can be grouped according to the nature of data considered to derive the slice: static, dynamic, or both. A broad description of the different slicing approaches is provided in [127].

Static slices are built by analyzing variables reachability through the inspection of abstract representation of the program such as Control Flow Graphs [125], Program Dependence Graphs [99], Value Dependence Graphs [45], System Dependence Graphs [113, 24] or Data Dependencies [97].

Effectiveness of static slicing approaches is often limited by programming language features like indirect access, pointers, function, polymorphism and concurrency. This features make complex pointer analysis or conservative approximations necessary thus increasing both slices size and computation time.

Dynamic slicing approaches have been introduced to reduce the size of the computed slice. Dynamic slicing approaches monitor the system during an execution and trace the program elements covered. Only the statements covered at runtime are considered to build the slices [8, 78, 74]: this lead to slices which are smaller than the ones generated with static approaches. Despite the reduced size of generated slices, dynamic slicing is seldom applied because of the impact on system performance and memory consumption caused by the runtime monitoring.

Combined dynamic-static approaches have been proposed in order to overcome the limitations of static and dynamic approaches. Hybrid slicing for example reduce monitoring costs by focusing only on the statements covered between the breakpoints set in a debugging session [57]. Other approaches reduce the slicing overhead by identifying data dependencies relationships at runtime [120], tracing executed method calls [96] or reduce the amount of monitored data by executing only the statements in the static slice [105].

Other methods to improve program slicing results common to both static and dynamic approaches regard the combination of different slices. In [126] Lyle and Weiser present dicing, which consist in the identification of the difference between a static slice of a correct variable and the slice of an incorrect one. In [101] Pan and Spafford present different heuristics that

use set algebra applied to dynamic slices derived from test cases to identify specific problems. For example missing initializations are identified by selecting statements present in passing tests slices but not in failing ones.

Tables 1.3 and 1.4 summarize results obtained in different empirical evaluations of static and dynamic slicing approaches over medium and large size programs. The average size of a static slice correspond to the 26% of the original program, while the average size a dynamic slice correspond to the 3%. Even if dynamic slices clearly permit to reduce developers effort in locating faulty statements they still report a huge number of lines to be inspected (in large sized programs like the Tomcat web server [15], the 3% of the overall lines of code correspond to more than 300 lines). The huge amount of lines reported makes slicing approaches useful if combined with symbolic debuggers but ineffective if applied as a stand alone technology [79].

	#Programs	LOC			Average Slice Size				
		Avg.	Min.	Max.	SDG-1	SDG-2	DD	Codesurfer	Sprite
S1	43	23421.00	563	149050	30.2	28.1	-	-	-
S2	7	3881.57	551	11320	-	-	52.14	-	-
S3	18	6396.17	842	14554	-	-	-	16	13

Legend	
S1	Suite of programs described in [24]
S2	Suite of programs described in [97]
S3	Suite of programs described in [113]
LOC Avg.	Average size of programs in the suite
LOC Min.	Size of the smallest program in the suite
LOC Max.	Size of the biggest program in the suite
SDG-1	Slicing based on SDG with structure fields collapsed [24]
SDG-2	Slicing based on SDG with structure fields expanded [24]
DDT	Slicing based on Data Dependence Types [97]
Codesurfer	Slicing with the Codesurfer tool [113]
Sprite	Slicing with the Slicer tool [113]

Table 1.3: *Empirical results obtaned with static slicing*

	#Programs	LOC			Average slice size/LOC		
		Avg.	Min.	Max.	DS	FS	RS
Injected faults	5	3747.88	307	12418	3.53	9.70	11.22
Real faults	3	47211.53	716	253832	4.09	7.46	21.53

Legend	
DS	Data slicing approach [132]
FS	Full slicing approach [132]
RS	Relative slicing approach
X/LOC	Size of the slice obtained with the specific approach relative to the program size
X/Coverage	Size of the slice relative to the lines covered during program execution

Table 1.4: *Empirical results obtaned with dynamic slicing*

1.7 Spectra based techniques

Spectra based techniques overcome the limitations of slicing techniques by identifying and comparing the set of instructions executed during correct and failing executions without considering data dependence relations [61]. For example developers collect the set of code blocks executed during correct and failing executions (these sets are known as Block Hit Spectra) and manually compare the blocks contained in the two sets [61, 7]. Code blocks present in the failing sets only are considered suspicious and are manually inspected by developers to look for erroneous instructions.

The manual comparison of spectra belonging to failing and correct runs in order to detect the statements that seem to impact on failing executions is difficult and expensive. For this reason researchers adopted resemblance coefficients [106] to automatically identify the suspicious instructions. Table 1.5 shows the three most effective resemblance coefficients presented in software engineering literature: Jaccard [28], Tarantula [72] and Ochiai [2]. Resemblance coefficients are used in combination with block hit spectra to identify the program statements mostly related with the fault. Each coefficient calculates a score for each program statement that indicates how much related to the fault is. Statements are presented to developers sorted in descending order: statements with the highest score present the greater probability to contain the fault. Spectra coefficients differ for the way the score is calculated, but in general they relates the number of faulty executions in which each statement is executed or not with the numer of correct executions in which the statement is executed or not. In [3] Abreu presents an approach that combines resemblance coefficients with model based debugging in order to filter the results provided by spectra and augment the efficacy of the analysis.

Other approaches extend the usage of resemblance coefficients by applying them not to coverage information but to properties of program variables [30, 83, 70, 133]. For example Liblit et al. identify faulty branch conditions by comparing the number of times a branch condition is true or not in failing and correct executions [83].

Some researchers integrate resemblance coefficients with machine learning techniques to diagnose faults which simultaneously cause multiple test failures [25, 71, 133]. Jones, for example, integrates information about branch coverage and statements suspiciousness to cluster together test

Coefficient	Formula
Jaccard [28]	$S_J(l) = \frac{a_{11}(l)}{a_{11}(l)+a_{01}(l)+a_{10}(l)}$
Tarantula [72]	$S_T(l) = \frac{\frac{a_{11}(l)}{a_{11}(l)+a_{01}(l)}}{\frac{a_{11}(l)}{a_{11}(l)+a_{01}(l)}+\frac{a_{10}(l)}{a_{10}(l)+a_{00}(l)}}$
Ochiai [2]	$S_O(l) = \frac{a_{11}(l)}{\sqrt{(a_{11}(l)+a_{01}(l))*(a_{11}(l)+a_{10}(l))}}$

Legend	
$a_{11}(l)$	Number of failing test cases in which statement l is executed.
$a_{01}(l)$	Number of failing test cases in which statement l is not executed.
$a_{10}(l)$	Number of passing test cases in which statement l is executed.
$a_{00}(l)$	Number of passing test cases in which statement l is not executed.

Table 1.5: *Program spectra coefficients*

cases that present a same failure and identify with Tarantula the faulty statements that cause each failure [71]. *RUBAR* by Briand et al. instead uses information from test case specifications to buid a C4.5 decision tree [104] that clusters together inputs that cause different failures. The faulty statements are identified among the ones executed most frequently when failing inputs are passed [25].

Experimental results obtained by Abreu et al. in [3] indicate that program spectra can be quite effective in localizing faults: in the optimal case program spectra permit to locate the 18% of faulty instructions by inspecting the 1% of the code and over the 70% of faulty instructions by inspecting the 20% (these results are shown in Table 1.6). These results indicate that spectra techniques can be effective in case only small program portions are analyzed during debugging, e.g. test cases that stress unit faults identified during development, but cannot be adopted to analyze complex executions of thousand lines programs. Furthermore these techniques only highlight suspicious code locations but do not provide any additional information to help developers in the final diagnosis of the problem.

Technique	% of faulty lines reched inspecting c% of code		
	c=1%	c=10%	c=20%
Jaccard	<12.00%	<46.00%	<60.00%
Tarantula	12.00%	46.00%	60.00%
Ochiai	14.00%	52.00%	65.00%
ObM	18.00%	60.00%	71.00%

Table 1.6: *Performance analysis of different spectra coefficients*

1.8 Statistical sampling

Program spectra approaches require the complete monitoring of correct and failing executions. The performance overhead caused by the monitoring has a low impact on tests execution but it is often not acceptable to monitor executions in the field.

Techniques based on statistical sampling aim at reducing the overhead of the monitoring by sampling in the field executions [18, 30, 70, 83, 133]. The data relevant for the localization of the fault, for example the coverage of true/false branches [83], are sampled in a manner equivalent to a Bernoulli process: each potential sample is taken or skipped randomly and independently as the program runs, which means that the monitoring framework randomly decide to check if the true or false branch of a particular condition is taken. Since sampled data is automatically retrieved from executions run by the final users of the applications, the statistical relevance of data collected depends on the number of runs and users monitored.

Technique	% of faulty lines reched inspecting *c*% of code		
	c=1%	c=10%	c=20%
Jiang07 [70]	29.23%	56.92%	62.30%
Liblit05 [83]	7.69%	40.00%	63.84%
SOBER [85]	8.64%	52.31%	73.85%

Table 1.7: *Performance comparison of different statistical sampling approaches.*

Sampling have been applied to the monitoring of different kind of information: truthness of branch predicates [83, 85, 70], coverage of execution

paths [30]. Table 1.7 summarizes the results obtained in different empirical evaluations of statistical sampling approaches conducted on the same set of programs ([18], [30], [70]). For each technique we report the percentage of faults covered by inspecting the 1%, 10% and 20% of the program. Sampling techniques effectiveness in locating faults is a bit lower than spectra techniques: by inspecting at most the 20% of the code sampling techniques locate the 50% of faulty instructions while spectra techniques locate the 60%. Even if statistical techniques are carachterized by a lower effectiveness they are more suitable for analyzing in the field failures. A limitation of the statistical approaches is that they require a statistically significant sample, which practically means that in order to identify faults occurring in the field they need a large amount of users experiencing the failure which is not always possible.

1.9 Runtime Anomaly Detection

Runtime anomaly detection techniques use models of the application behavior to identify anomalies in failing executions. To overcome the lack of complete specifications these techniques often use models inferred from data recorded during monitored executions. The analysis of the behavioral deviations guide developers in the identification of the fault.

Anomaly detection techniques vary for the kind of models inferred and the type of data considered at runtime. Diduce [58] and Carrot [103] infer boolean expressions that describe potential data invariants by applying a predefined set of rules to data values monitored during program executions. For example these invariants can detect that the integer value returned by a method is always positive. Other approaches instead do not consider data values but focus on method calls to identify anomalies. In [38] sequences of method invocations are collected to derive for every class of the system all the sequences of methods (up to a given length) that objects of the specific class invoke, and all the sequences of methods belonging to that class invoked from other objects.

Pachika [39] and Behavior Capture and Test [87] instead focus on both data and invocation sequences. Pachika infers automata that model the object usage protocol: states are identified by inferring data invariants over object attributes, transitions correspond to methods that during the execution caused the transition from one state to another. Faults are identified

by comparing models of faulty executions with models of correct executions. Behavior Capture and Test (BCT) instead infer Finite State Automata that generalize the sequences of interactions between components, and data invariants on the parameters exchanged in different invocations. The technique is used to identify integration problems caused by the replacement of components.

Runtime anomaly detection techniques are not only adopted to localize faults but also to identify failures or detect malicious executions. The applicability of anomaly detection techniques for these purposes is strongly limited by the false positives generated by these techniques. The models used to identify anomalies are inferred from data recorded during monitored executions: if models are inferred from data that only partially covers the legal application behavior many false positives can be detected when legal executions not observed before occur. For this reason anomaly detection techniques are mostly used to higlight suspicious executions but they rely on manual inspection to detect if the highlighted executions really correspond to failures or malicious attacks.

Lee et al. in [80] derive association rules that given a sequence of k events (Lee considered system calls) predict the event at position k+1. Violations of the rules identified in traces recorded during failing executions are threated as anomalies. Association rules do not model the order of the observed events for this reason they cannot be applied to detect faults charactherized by a wrong order of operations executed.

Warrander et al. in [123] infer Hidden Markov Models (HMMs) that generalize the sequences of events observed at runtime. Each state of the model represent a different sequence of events of lenght k (k is chosen by developers), while each transition describes the probability to reach a given state from another state. The inferred HMMs are then used at runtime to identify anomalous events that lead to unusual state transitions. HMMs are used in a way which is similar to the way FSA are used in BCT. The only limitation of HMMs is constituted by the time needed to build the model which can take several days for complex executions [123].

Other techniques identify anomalies not as deviations from expected behaviors but by matching the actual execution with models of known faulty behaviors. These models are built through supervised learning algorithms that use information collected during both correct and faulty executions. Chen et al. for example use decision trees to identify failures in large inter-

net sites [27]. The authors derive decision trees that permit to detect if a failure is occurring at runtime by using information about previous executions like the name of the machine in use or the type of request coming to the site. Like all the other supervised learning approaches this technique permits only to identify problems that already occurred and do not help in case of failures never experienced before. Since fault localization usually regard problems not diagnosed in the past the applicability of supervised learning approaches for fault localization is limited.

Anomaly detection techniques are applicable in several context and can monitor different application aspects. The results reported are described in terms of deviations from expected behavors thus helping developers in the identification of the problem and in certain cases permitting an automated identification of the solution [39]. Despite they permit to identify the misbehaviors that lead to the failure they often report many false positives thus resulting not effective in practice expecially when programs are big, complex and a full coverage of the legal application behavior at testing time is not possible.

Chapter 2

Dynamic Diagnosis of Software Misbehaviors

This Chapter presents a case study that shows how existing fault localization techniques can be applied to diagnose a fault affecting a complex software system. The fault is real, it affects the Tomcat web server version 6.0.4 and has been published in the Tomcat bug database with id 40820 [13]. Not all the described techniques have been applied by Tomcat developers but by ourselves. However we identify a set of limitations of the existing techniques that realistically prevent them from providing useful results to developers.

This PhD Thesis describes a framework for fault diagnosis based on anomaly detection that overcomes the limitations of known techniques. The last section of this Chapter introduces the analysis framework and describes our contributions to the state of the art.

2.1 A developer's story

On October 26, 2006, Scott, a J2EE developer, unexpectly realized that one of his web applications was not started by Tomcat 6.0.4 after the server reboot. After manually inspecting logs and running a few debugging sessions Scott discovered that the failure was caused by a `NullPointerException` due to a `null` value returned by the method `JspFactory.getDefaultFactory()`. This information did not help Scott much because it did not

indicate him why a `null` value is returned by method `JspFactory.getDefaultFactory()` and whether the `null` value is originated by a fault in Tomcat (not correctly initializing the JspFactory) or by a fault that he introduced in the application (not correctly managing the initialization of the web application).

Since Tomcat is both extremely complex (more than 100.000 lines of code) and incompletely documented, Scott has been unable to fix the bug and posted the issue to Tomcat Bugzilla [13].

System complexity and lack of specifications increased the time required to debug and fix the problem. In fact Tomcat engineers released a fix only after four months from the discovery of the problem. Even worst, in an intermediate stage of work developers supposed to have fixed the fault only because the were *not able to reproduce the failure*.

Tomcat engineers recognized that the fault was in Tomcat which *does not initialize the default JspFactory before starting the deployed web applications*.

Tomcat engineers likely looked at the static slice of the return value in order to identify the methods supposed to initialize the factory. The default JspFactory is set through method `JspFactory.setDefaultFactory(JspFactory)` which is called from two different code locations: the static initialization block of class `org.apache.jasper.compiler.JspRuntimeContext` and the static initialization block of class `org.apache.jasper.runtime.HttpJspBase`.

Static initialization blocks are invoked by the Java Virtual Machine (JVM) when a class is used for the first time. Class loading occurs at runtime and the order in which classes are loaded depends on execution flow and JVM specifications: the static analysis of the program alone do not allow to determine the class loading sequence, this makes *static slicing of little use to debug Tomcat*.

Java developers rely on static initialization blocks to ensure that certain initialization activities are performed when a class is used for the first time or it is explicitly loaded through the method `Class.forName(String)` (this approach is used to load and configure JDBC drivers for example [44]).

This programming practice has been adopted also by the developers of Tomcat Jasper component: Listing 2.1 shows the implementation of method `JasperListener.lifeCycleEvent` that invokes method `Classloader.loadClass(String)` in order to load class `JspRuntimeContext`

(see lines 6 and 7).

```
58      public void lifecycleEvent(LifecycleEvent event) {
59
60          if (Lifecycle.INIT_EVENT.equals(event.getType())) {
61              try {
62                  // Set JSP factory
63                  this.getClass().getClassLoader().loadClass
64                      ("org.apache.jasper.compiler.JspRuntimeContext");
65              } catch (Throwable t) {
66                  // Should not occur, obviously
67                  log.warn("Couldn't initialize Jasper", t);
68              }
```

Listing 2.1: *JasperListener excerpt*

The programmer who developed Tomcat expected that the execution of method `JasperListener.lifeCycleEvent` forces both the loading of class `JspRuntimeContext` and the execution of its static initializer. However, the programmer who implemented method `JasperListener.lifeCycleEvent` used the wrong API method to load the class: method `Classloader.loadClass(String)` has been used instead of method `Class.forName(String)`. Unfortunately, method `Classloader.loadClass(String)` does not execute the static initialization block (as `Class.forName(String)` does), but postpones its execution to the first usage of the class. During the execution of method `JasperListener.lifeCycleEvent` the class `JspRuntimeContext` is loaded but the static initializer is not executed: The default `JspFactory` is still `null` and web applications cannot gain a reference to a valid `JspFactory`.

Since the signatures of methods `Classloader.loadClass(String)` and `Class.forName(String)` look similar, it is hard to detect the fault by code inspection, if you do not already know the tricky difference they introduce on class loading.

Tomcat engineers can apply other approaches instead of code inspection in order to identify the fault. Unfortunately all the existing automated and semi-automated approaches produce results which are partially useful for the diagnosis of the problem.

The approach that provides the most effective results is *interactive debugging* and is probably the one adopted by Tomcat engineers to debug Tomcat. Tomcat engineers can detect that the class `JspFactory` is not initialized during the invocation of method `Classloader.loadClass(String)` by setting breakpoints in method `JspRuntimeContext.<clinit>()` before and after the execution of `JspFactory.setDefaultFactory(JspFactory)` and in method `StandardContext.start()` before

the invocation of `Classloader.loadClass(String)`. This setup permits evelopers to follow the execution flow of these methods and thus detect that `JspFactory.setDefaultFactory(JspFactory)`, which initializes the factory, is not invoked during the execution of `Classloader.loadClass(String)` but later.

Interactive debugging does not help developers in the identification of the methods to monitor, for this reason they could spend a lot of time before identifying the methods in which breakpoints should be set. Furthermore in this case developers need Scott's web application in order to reproduce te failure and debug the system. Scott could not share his web application for licence issues, thus he had to develop a simpler application that triggers the same failure thus wasting time (the debugging could have been even more difficult if Scott was not a developer himself).

In order to automate the debugging activities the *delta debugging* technique can be applied. Delta debugging works by comparing a successful execution, e.g. the startup of Tomcat without Scott's web application, with a failing one, e.g one in which Scott's web application has been deployed. Delta debugging finds the smallest set of variables values that lead the application to a failure by comparing and altering the state of failing and faulty executions. Unfortunately the failing and correct executions of the Tomcat case study have a similar state, the only difference between them is that in correct ones `JspFactory` was not loaded. Delta Debugging will not identify the loading of class `JspFactory` as a relevant difference because the loading of this class alone does not cause any failure. It is the invocation of method `JspFactory.getDefaultFactory()` what causes the failure but Delta Debugging will not capture it. Method `JspFactory.getDefaultFactory()` is invoked within Scott's web application, which is not deployed in successful runs. The only relevant difference between failing and successful runs that can be identified with Delta Debugging is that the web application is not deployed in successful runs. This information was already known by developers and obviously does not permit to localize the fault.

Another solution for the automated identification of faulty statements is the analysis of program specra. We ran Tarantula to compare spectra recorded during successful Tomcat executions with the spectra recorded during the failing execution of the system. Tarantula detects that the instructions mostly related with the failure are the ones that regard the man-

agment of the `NullPointerException` and the logging of error messages. Unfortunately this information is not useful to diagnose the fault. Tarantula also recognizes the invocation of `Classloader.loadClass(String)` as one of the most suspicious methods, but it also indicates that other 112 statements (the 0.42% of the executed ones) are more suspicious than this one. The manual inspection of 112 statements for the identification of the faulty ones can be tedious, furthermore develpers will not recognize that the invocation of method `Classloader.loadClass(String)` as beeing faulty just by looking at it. More information about the server behavior like the missing initialization of class `JspFactory` is needed to diagnose and fix the fault.

Anomaly detection techniques focus on the detection of behavioral anomalies but still do not ease the diagnosis of the fault. DIDUCE for example detects that the value returned by method `JspFactory.getDefaultFactory()` is returning an anomalous `null` value but it does not provide any information about the cause of this anomalous behavior. PACHIKA instead detects that the invocation of `JspFactory.getDefaultFactory()` is erroneous because the default `JspFactory` has not been initialized. Unfortunately this anomaly can be misleading because it suggests that the problem is in the web application which is calling `JspFactory.getDefaultFactory()` while the problem resides elsewhere, in the class `JasperListener.lifeCycleEvent` which is not properly loading `JspRuntimeContext.<clinit>()`.

In order to help developers in the diagnosis of Tomcat fault automated techniques should indicate developers that:

- the `null` value returned by `JspFactory.getDefaultFactory()` is anomalous thus permitting developers to detect that the problem is not in the caller but in Tomcat itself;
- in the failing execution the class `JspFactory` is used before being initialized thus permitting developers to detect that this is the cause of the `NullPointerException`;
- in successful executions the class `JspFactory` is initialized after web applications deployment thus permitting Tomcat engineers to detect that the initialization comes too late;
- the class `Classloader.loadClass(String)`, which is invoked be-

fore the deploy of web applications, does not force the execution of `JspRuntimeContext.<clinit>()`, which is the method that initializes `JspFactory`, thus permitting engineers to detect that the invocation of method `Classloader.loadClass(String)` is not the corret one and fix the fault.

2.2 Anomaly detection for fault diagnosis

Section 2.1 shows how hard it could be the diagnosis of integration faults (between an application, Tomcat and the JVM in this case). The information required for the diagnosis of Tomcat fault could be identified by an anomaly detection technique that captures information about both variable values and components interactions.

A technique that compares the values exchanged in method calls with models of the value domain can detect that method `JspFactory.getDefaultFactory()` is not supposed to return `null` thus making Scott aware of having used the JspFactory correctly. The comparison of the interactions of Tomcat and JVM components with models that generalize the interactions observed during successful executions permits Tomcat engineers to detect that in the failing execution class `JspFactory` is unexpectedtly loaded for the first time during the deployment of web applications. Furtermore it will permit to detect that the initialization of `JspFactory` happens too late, after the deploy of web applications thus indicating that `Classloader.loadClass(String)`, which is invoked before web applications deployment, is not initializing the `JspFactory`.

This PhD Thesis describes a framework for fault diagnosis based on anomaly detection that permits to diagnose faults experienced in the field. The technique advances current anomaly detection techniques by:

- Capturing the application data flow [37, 88]. We model relationships between runtime events observed and data associated to those events (e.g. parameters exchanged between different components).
- Automatically filtering false positives [89]. We automatically remove anomalies that depend not on the fault but on legal behavior of the system not observed previously.
- Presenting behavioral anomalies in a structured way [89]. We present anomalies in a graph that explicitly shows the cause effect relations

between multiple anomalies, and that identifies different clusters of related anomalies.

- Automatically interpreting anomalous event sequences [20]. We automate the interpratation of anomalous sequences, an activity usually performed by developers manually. For example we tell developers that class `JspFactory` has been used before than expected in the Tomcat case study.

The approach proposed in this PhD Thesis is based on three phases. Figure 2.1 illustrates them.

In the *first phase* we capture the legal application behavior. The first phase takes place at testing time. During tests execution we monitor the application and collect data useful to describe the behavior of the system. We consider two kinds of information: we collect the log files recorded by the application during test execution and we use instrumented monitor that record the sequences of method calls executed by the components of the system and the state of the exchanged parameters.

In order to capture Tomcat legal behavior we can either run functional tests of Tomcat and collect the logs generated at runtime or instrument Tomcat before running the tests and collect information about the invoked methods. A sequence recorded during a successful execution could be the following:

```
< StandardContext.start()Classloader.loadClass(String)...
StandardContext.deployApps()...
JspRuntimeContext.<clinit>()JspFactory.<clinit>()
JspFactory.setDefaultFactory(JspFactory)
JspFactory.getDefaultFactory()... >
```

The sequence describe what happens during a Tomcat successful execution: during the start of the Tomcat Standard Context component the method `Classloader.loadClass(String)` is invoked, web applications are deployed, and then, after the deploy the default `JspFactory` class is set. For each monitored method we also record the state of the exchanged parameters, for example the values of the attributes of the object returned by method `JspFactory.getDefaultFactory()`.

We use the data recorded during tests executions to infer models of the observed behavior. In particular we derive finite state automata that gen-

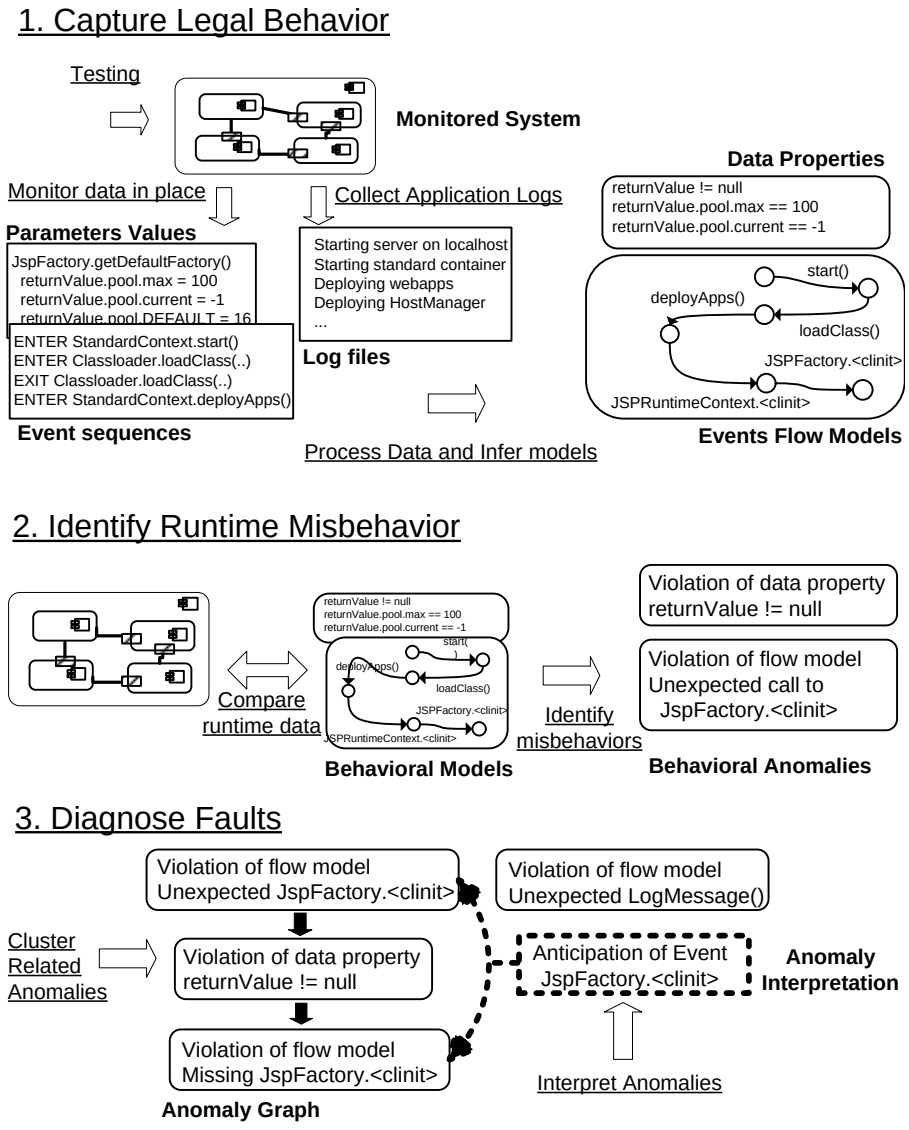


Figure 2.1: *Overview of the Automated Fault Diagnosis Framework.*

eralize the flow of events or data values observed during successful executions and boolean expressions that capture the properties of the observed variables. For example we use Tomcat data to derive a finite state automaton that accepts the sequence of events recorded and boolean expressions that indicates that the value returned by `JspFactory.getDefaultFactory()` is always different from `null`.

In the *second phase* we monitor the application behavior. This phase takes place in the field while the application is running or during the execution of new tests. In this phase we still observe the runtime application behavior by extracting data from the logs or from the instrumented monitors and we compare the data collected with the models inferred in the first phase in order to identify misbehaviors. Violations of the models, for example event sequences not accepted by an automaton or data values that invalidate the inferred boolean properties, correspond to anomalous behaviors of the system. We keep trace of every violation detected.

In order to identify Tomcat misbehaviors we monitor the failing startup of the server. During monitoring we could detect for example that after the execution of `StandardContext.start()` there is an unexpected execution of `JspFactoryImpl.<clinit>()`, which indicates that class `JspFactory` is unexpectedly loaded. Furthermore we can also detect that the `null` value returned by method `JspFactory.getDefaultFactory()` violates the inferred data property that indicates that the return values must be alway not `null`.

In the *third phase* we automatically interpret the anomalies recorded and present them in a way that permits developers to diagnose the fault more easily. This phase takes place after a failing execution has been monitored. At this stage we filter eventual false positives by discarding the anomalies that occur in both failing and correct executions. Then we correlate the remaining anomalies by detecting eventual cause effect relationships, and cluster them according to the likelyhood to be caused by a same cause. Finally we provide an interpretation of flow anomalies by comparing them with a predefined set of patterns that permit to detect for example that the execution is characterized by the anticipation of one or more events. For example in the Tomcat case we detect that the unexpected usage of class `JspFactory` is causing method `JspFactory.getDefaultFactory()` to return an invalid `null` value. Furthermore we indicate developers that the usage of `JspFactory` is anticipated with respect to

sucessful executions. We also detect that the initialization of `JspFactory` is performed by method `JspRuntimeContext.<clinit>()` which occurs after expected, thus indicating that `Classloader.loadClass(String)` is initializing the class.

Following Chapters describe the details of each phase and the preliminary results we obtained. Chapter 3 describes how we collect behavioral data and infer behavioral models. Chapter 4 presents the activities performed at runtime to monitor the application and identify anomalies. Chapter 5 describes the techniques developed to analyze and interpret behavioral anomalies. Finally Chapter 6 discusses the results we obtained with third-party and industrial case studies.

Chapter 3

Capture Legal Behavior

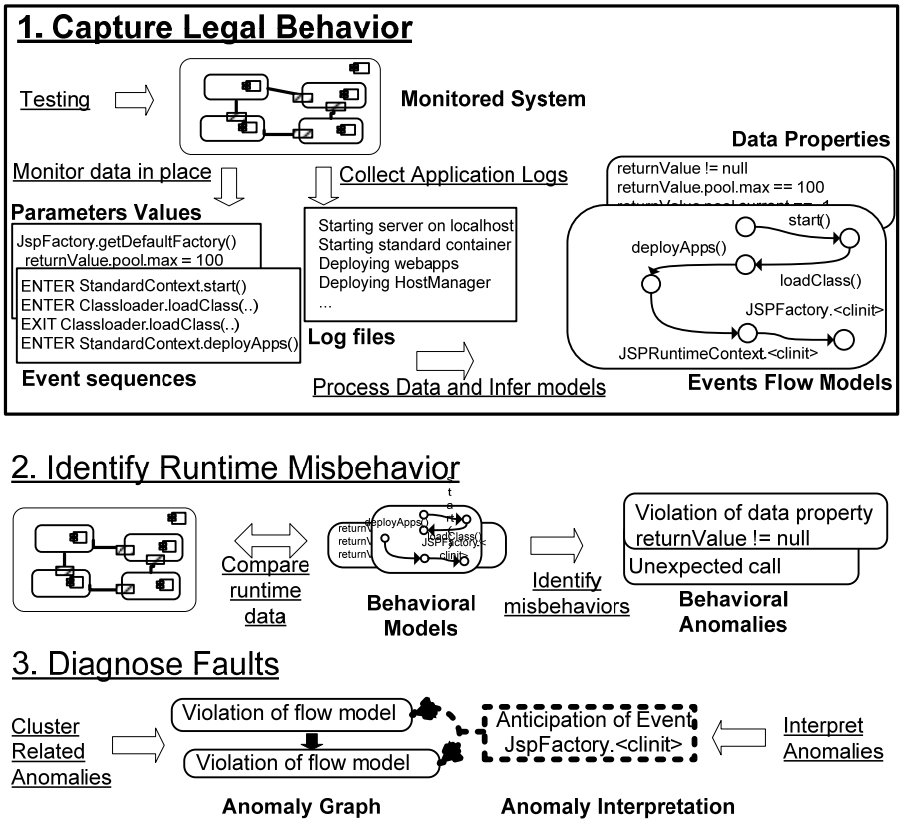


Figure 3.1: *Overview of the Automated Fault Diagnosis Framework. The Capture Legal Behavior phase is highlighted.*

This Chapter describes the activities we perform to capture the legal behavior of a monitored application by inferring models that generalize properties of the data observed during successful executions. Section 3.1 describes which data we consider in order to capture and model the legal behavior of a software system. Section 3.2 presents the techniques that we adopt to mine the behavioral data from concrete executions though the processing of log files and the monitoring the behavior of the software components integrated in the system. Finally Section 3.3 describes the kinds of models we infer and the aspects of the application behavior they capture.

3.1 Identification of the data to be collected

The common way software engineers diagnose faults is either by analyzing the log files generated during the failing execution or by inspecting the state variables of the application using a symbolic debugger, while reproducing the failing execution. In both cases engineers looks for the anomalous sequences of events or the suspicious data values that likely caused the observed failures.

Our approach is based on the idea of automating the activities that developers usually perform manually when they debug a software system. For this reason we first collect data about the behavior of the system during successful executions, and then we compare this data with the ones collected during failing executions in order to identify the misbehaviors that triggered the failure. In order to collect these behavioral data we use either monitoring tool or log files already generated by the software system.

We focus on the functional behavior of software systems. For this reason we focus on the sequences of activities performed by a system, e.g. the sequences of methods the software executes, and the information about the state of the system during the execution, e.g. the values of state variables of the system during its execution. Following sections describe the strategies we adopt to collect these data from concrete executions and then present the techniques we use to process them and build models that capture aspects of the legal behavior of the application useful for the identification of misbehaviors caused by faults.

3.2 Mine behavioral data

This Section presents the techniques we adopt to mine behavioral data from actual executions. Subsection 3.2.1 describes how we process log files to identify useful data, while subsection 3.2.2 presents the techniques that we adopt to extract the data by monitoring the appliction from inside.

3.2.1 Recognize behavioral data in log files

A critical issue that emerges when dealing with log files is that they present unstructured messages in which information on the state of the system and on the activities taking place are mixed. Traditional approaches use ad hoc parsers to extract information from logs [102, 60, 121], but the usually large number of possible event types makes the definition and maintenance of such regular expressions an expensive and error prone task [102]. We want to generate a solution which is flexible and independent from the log format. The only assumption that we make is that the different messages recorded in a log file can be easily distinguished from each other. Then we apply Simple Logfile Clustering Tool (SLCT) [122] to the different log messages in order to extract from the logs the name of the events taking place and the variable values that describe the state of the system.

In order to identify the beginning of a new message we use simple regular expressions defined by users. Depending on the log format the beginning of a new message can be identified by looking for log lines starting with a date for example. The expression that matches this case can be very short and simple; maintaining simple regular expressions is obviously much easier that maintaining a whole parser. Furthermore it is easy to define a predefined set for the most common logging format, thus requiring developers to specify only which is the log format they use.

Altough the intestation of each log message is predefined for all the known formats, the content of the messages is not structured and mixes events and variables. We use SLCT to extract event names and variables. SLCT is a clustering algorithm that identifies substrings that frequently appear in logged events and heuristically generates a set of regular expressions that specify how to separate the constant part of a logged event, i.e., the event name, from the variable part, i.e., its attributes. For instance, if the recorded log files include the events "`User Fabrizio logged in`

`with IP 10.0.1.3"`, `"User Leonardo logged in with IP 10.0.1.7"`, `"User Fabrizio logged in with IP 10.0.1.9"`, `"User Leonardo logged in with IP 10.0.1.5"`, SLCT produces the regular expression `"User * logged in with IP *"` to indicate that the user name and the IP number are the attribute values while the rest is the signature of the event.

SLCT works by identifying the words that frequently appear in each line, grouping lines with common words in clusters, and generating regular expressions that well represent all the elements in a cluster. SLCT discards incidental regular expressions by computing the support associated with each regular expression, and discarding the ones with a support below a given threshold. The support of a regular expression is the number of events in the analyzed log file that are generated by the regular expression.

In some cases, the inferred regular expressions can incidentally include some attribute values as part of the event name. This happens when the same specific attribute values are frequently present in the input log file. For instance, if Fabrizio and Leonardo are the only users who log into the system, SLCT can identify the two regular expressions `"User Fabrizio logged in with IP *"` and `"User Leonardo logged in with IP"`, instead of `"User * logged in with IP *"`. When there are multiple options in the regular expressions that can be generated, SLCT gives priority to the ones with the highest number of constant values, i.e., `"User Fabrizio logged in with IP *"` and `"User Leonardo logged in with IP"` have priority on `"User * logged in with IP *"`.

To avoid imprecise detection of event names, we identify regular expressions in multiple iterations of the SLCT algorithm. This practice has already been successfully applied in [122]. In the first iteration, we identify regular expressions with a support that is greater than 5% of the number of events recorded in the log file. If no regular expression is identified, we reduce the requested support by 25% and we run again the algorithm. If regular expressions are identified, we consider the events not generated by any regular expression and we execute again the algorithm with this set. We continue in this way until no events need to be analyzed or the threshold for the support is less or equal to 1. This process eases the identification of event names because general rules, such as the one containing only the event name, more easily satisfy higher thresholds, than the specific rules

that incidentally include attribute values.

We use the identified expressions to parse the logs and put in a tabular structure event names and the different parameters. The parameters are identified as the words that match the variable part of the expression, while event names are replaced by identified through unique abstract names, each corresponding to a distinct expression.

Table 3.1 shows the regular expressions and corresponding event ids that SLCT identifies for the logs in Figures 3.2 and 3.3. In order to identify relations between portions of paths and URLs we threat the path separator characters like ':' and '/' as spaces, thus permitting SLCT to identify variable sub-paths. In this way we are able to detect that the IP address and the port number associated with `event2` in Figures 3.2 and 3.3 are variables. SLCT correctly identifies the variables associated with an event only if that event occurs with a sufficient number of different parameters, otherwise the values are erroneously incorporated in the event name. If we consider Table 3.1 we see that SLCT, by processing line 2 in Figures 3.2 and 3.3, correctly identifies that the IP-address and the port number associated with Event2 are variable but it does not detect that also the database type, mysql, can vary. Of course SLCT cannot recognize that the database type is a varying element because all the logs present the same value. Tables 3.2 and 3.3 show the event names and parameters identified by applying the rules in Table 3.1 to the logs in Figures 3.2 and 3.3 respectively.

```
1  Starting server on 10.0.0.1
2  Starting connection with DB jdbc:mysql
       ://10.0.0.1:3306/ePortal
3  Connection established
4  Server started
5  Request from IP 10.0.1.3
6  User Fabrizio logs in with IP 10.0.1.3
7  User Fabrizio logged in with IP 10.0.1.3
8  Modify profile for Fabrizio (IP 10.0.1.3)
9  Profile modified for Fabrizio (IP 10.0.1.3)
10 User Fabrizio logged out (IP 10.0.1.3)
11 Request from IP 10.0.1.7
12 User Leonardo logs in with IP 10.0.1.7
13 User Leonardo logged in with IP 10.0.1.7
14 Modify profile for Leonardo (IP 10.0.1.7)
15 Profile modified for Leonardo (IP 10.0.1.7)
16 User Leonardo logged out (IP 10.0.1.7)
17 Server stopped
```

Figure 3.2: *Excerpt of a log recorded when users Fabrizio and Leonardo interact with the web application running at host 10.0.0.1.*

```
1  Starting server on 10.0.0.2
2  Starting connection with DB jdbc:mysql
       ://10.0.0.2:3308/ePortal
3  Connection established
4  Server started
5  Request from IP 10.0.1.5
6  User Leonardo logs in with IP 10.0.1.5
7  User Leonardo logged in with IP 10.0.1.5
8  Modify profile for Leonardo (IP 10.0.1.5)
9  Profile modified for Leonardo (IP 10.0.1.5)
10 User Leonardo logged out (IP 10.0.1.5)
11 Request from IP 10.0.1.9
12 User Mauro logs in with IP 10.0.1.9
13 User Mauro logged in with IP 10.0.1.9
14 Modify profile for Mauro (IP 10.0.1.9)
15 Profile modified for Mauro (IP 10.0.1.9)
16 User Mauro logged out (IP 10.0.1.9)
17 Server stopped
```

Figure 3.3: *Excerpt of a log recorded when users Mauro and Leonardo interact with the web application running at host 10.0.0.2*

Event Id	Corresponding regular expression
Event1	Starting server on *
Event2	Starting connection with DB jdbc mysql * * ePortal
Event3	Server started
Event4	Request form IP *
Event5	User * logs in with IP *
Event6	User * logged in with IP *
Event7	Modify profile for * (IP *)
Event8	Profile modified for * (IP *)
Event9	User * logged out (IP *)

Table 3.1: *Regular expressions derived by SLCT from the logs in Figures 3.2 and 3.3.*

Line #	Event name	Parameters	
		1st	2nd
1	Event1	10.0.0.1	
1	Event2	10.0.0.1	5432
1	Event3		
1	Event4	10.0.1.3	
2	Event5	Fabrizio	10.0.1.3
3	Event6	Fabrizio	10.0.1.3
4	Event7	Fabrizio	10.0.1.3
5	Event8	Fabrizio	10.0.1.3
6	Event9	Fabrizio	10.0.1.3
7	Event4	10.0.1.5	
8	Event5	Leonardo	10.0.1.5
9	Event6	Leonardo	10.0.1.5
10	Event7	Leonardo	10.0.1.5
11	Event8	Leonardo	10.0.1.5
12	Event9	Leonardo	10.0.1.5
12	Event10		

Table 3.2: *Event names and data values extracted from log in Figure 3.3 according to rules in Table 3.1*

Line #	Event	Parameters	
		1st	2nd
1	Event1	10.0.0.3	
1	Event2	10.0.0.3	5432
1	Event3		
1	Event4	10.0.1.5	
2	Event5	Leonardo	10.0.1.5
3	Event6	Leonardo	10.0.1.5
4	Event7	Leonardo	10.0.1.5
5	Event8	Leonardo	10.0.1.5
6	Event9	Leonardo	10.0.1.5
7	Event4	10.0.1.9	
8	Event5	Mauro	10.0.1.9
9	Event6	Mauro	10.0.1.9
10	Event7	Mauro	10.0.1.9
11	Event8	Mauro	10.0.1.9
12	Event9	Mauro	10.0.1.9
12	Event10		

Table 3.3: *Event names and data values extracted from log in Figure 3.3 according to rules in Figure 3.1*

3.2.2 Extract behavioral data through monitors

The common way applications execute tasks and change their state is through methods execution. For this reason many capture and replay techniques (for example [73]) successfully replicate field executions by recording the whole set of invoked methods and the state of the exchanged objects.

Since the complete monitoring of all the methods invoked in a system can affect performance and could be not applicable in the field, we focus only on invocations that cross component boundaries. This permit us to capture both the interactions between the different application components and the results generated by single components. The sequences of cross component method invocations observed during successful executions, indicate how the monitored component correctly interact with others to generate a valid result. The values of the parameters exchanged by method calls instead can be used to identify which are the values that can be generated or accepted by a certain component when a particular method is executed.

We instrument the methods of the components interfaces with monitors that, when a method is entered or returns, record its signature and the state of the exchanged parameters.

Since we focus on functional faults, we record the sequences of methods invoked within each thread separately. This permits us to capture the functional behavior of the application but it does not permit to trace its concurrency. For each monitored method we trace the state of its parameters and return values by applying the object flattening technique [90]. Object flattening traces the state of the objects in this way: for each value which is a primitive type or a string the technique records the string representation of the value, for each object it recursively records the state of its attributes. For every attributes it records the string representation if possible otherwise, if it is a complex object, it proceeds by recursively flattens its state and so on.

The result of the flattening process is a list of <*key,value*> pairs where the key uniquely identify the single parameter or attribute flattened while the value corresponds to the string representation of the attribute value. We construct the key by indicating the position of the parameter and the names of the attributes recursively inspected. Table 3.4 shows the result of the flattening of the value returned by method `JspFactory.get-`

`DefaultFactory()` during a successful execution of Tomcat 6.0.4 in which a method is invoking the factory method `JspFactory.getDefault-Factory()` in order to retrieve the default `JspFactory` object.

Idnetifier	Value
returnValue.log.logger.offValue.intValue()	2147483647
returnValue.USE_POOL.booleanValue()	true
returnValue.pool.lock	!NULL
returnValue.log.logger.anonymous.booleanValue()	0
returnValue.log.logger.kids	null
returnValue.pool.current.intValue()	-1
returnValue.deflt	@returnValue
returnValue.SPEC_VERSION.toString()	2.1
returnValue.pool.DEFAULT_SIZE.intValue()	16
returnValue.pool.max.intValue()	100

Table 3.4: *Result of the flattening of the return value for Tomcat 6.0.4 method* `Jsp-Factory.getDefaultFactory()`.

3.3 Inference of behavioral models

In order to identify faults we look for misbehaviors occurring during a failing execution. We identify misbehaviors by looking for behaviors that differ from the ones observed during successful executions. Since the set of behaviors monitored at testing time is limited, we need to generalize the data observed. For this reason infer models that capture the characteristics of the successful executions.

We derive three different kinds of models: data properties that capture method pre and post conditions [89], events flow models in form of finite state automata that generalize the sequences of observed events [89], and data flow models that capture the flow of values between different events [88]. Following sections describe in details how these models are inferred.

3.3.1 Data properties

Different research reports indicate that faulty data handling procedures are one of the principal cause of unit and integration faults [64, 23]. Illes et al. in [64] indicate that data handling mistakes like boundary values not properly processed, parameters combinations not considered, or invalid return values are often the cause of algorithmic errors affecting the results of single program units. Furthermore in [23] Binder indicates that common causes of integration faults are the violations of methods pre and post-conditions.

Data handling faults not discovered during testing manifest when certain data values never observed before are used or generated by the application components. If we identify the anomalous data we can point the developer directly to the fault. In order to identify anomalous data, we infer data properties that captures the range of values observed in successful executions, and the eventual implications between the different values observed.

Data handling faults not discovered during testing manifest when certain data values never observed before are used or generated by the application components, either because the users input them or the application generates them. If we capture the characteristics of the data observed during successful executions we can identify anomalous data values, either inputs or outputs, that do not respect those properties. Identifying the anomalies helps developers in the localization of the faulty component that generated an anomalous data or in the identification of the conditions that trigger a fault.

In order to capture data properties, we use Daikon, a tool which detects likely program invariants by identifying properties that hold on the data values recorded at testing time [46].

Figure 3.5 shows the data properties associated to method `JspFactory.getDefaultFactory()`. We derived these properties from traces recorded during testing of Tomcat 6.0.4. We executed functional tests for the management panel of Tomcat that we derived by applying the category partition methodology [98]. The first property in Figure 3.5 indicates that the return value is always different from `null`(this means that the `getDefaultFactory` method always returns a valid object). Property 2 indicates that the attribute lock is always different from `null`(in fact this

ID	Boolean expression
1	returnValue != null
2	returnValue.pool.lock != null
3	returnValue.pool.current.intValue() < returnValue.pool.max.intValue()
4	returnValue.log.logger.levelValue.intValue() > returnValue.pool.current.intValue()
5	returnValue.pool.max.intValue() == 100
6	returnValue.log.logger.offValue.intValue() > returnValue.pool.current.intValue()
7	returnValue.pool.DEFAULT_SIZE.intValue() > returnValue.pool.current.intValue()

Table 3.5: *Data properties automatically derived for method* `JspFactory.getDefaultFactory()` *of Tomcat 6.0.4.*

is the monitor used to lock the factory pool and cannot be `null`), while property 3 indicates that the size of the factory pool is always lesser than its max pool size. Since properties are inferred from data recorded during tests executions they can be imprecise, e.g. they infer a domain for the attributes which is a subset of the real domain, or incidental, e.g. they indicates a relation between two attributes that holds only from data collected during testing. Property 4 for example indicates an accidental relation between two integer attributes belonging to the factory pool and to the factory logger respectively.

Property 5 in Table 3.5 is an example of an imprecise property. This property indicates that the max pool attribute is always equal to 100. This property cannot be considered an invariant (the pool size can be changed from configuration files) but it could permit to diagnose eventual faults not detected at testing time: since tests were executed with a pool size of 100 a different configuration could trigger a silent fault. In this case, the identification of the difference between the runtime value and the values observed at testing time for attribute `pool.max` can suggest developers that the problem depends on a new configuration.

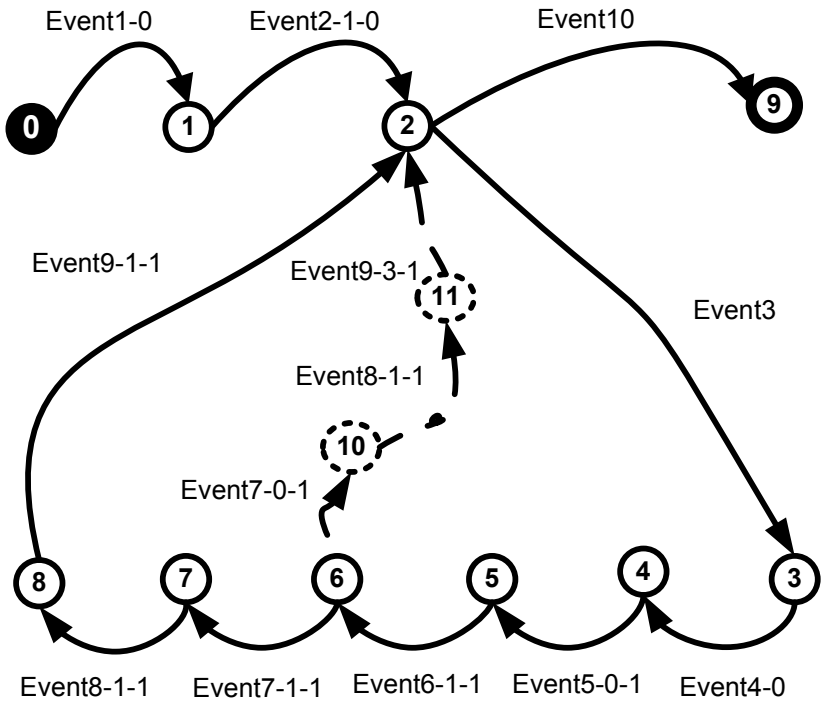
3.3.2 Events flow models

Events recorded at testing time describe the flow of activities during successful executions. Erroneous sequences of operations are common causes of software failures, typical examples are missing objects initializations or violations of interaction protocols [23]. By comparing the sequences of events generated in the field with the sequences recorded at testing we can effectively identify these differences.

The possible sequences of operations performed by a software program are generally infinite, and only a finite set of samples can be observed. For this reason we need to produce a general and compact representation of the possible legal behaviors from the sequences observed during successful executions. To this end we use KBehavior [87] an inference engine capable of deriving Finite State Automata (FSA) that generalize the event sequences that have been recorded. Many inference engines have been proposed in the past, a few examples are the KTail algorithm [22] and the Cook engine [36]. We adopted KBehavior because there are empirical results indicating that it correctly captures the legal application behavior in contexts similar to ours [90] and furthermore it is incremental, and we will use this feature for the identification of anomalies (see Section 1.9).

KBehavior incrementally analyzes the sequences of events generated during successful executions and generates a FSA that both summarizes and generalizes them. At each step, KBehavior reads an event sequence and updates the current FSA according to the content of the sequence. The updated FSA guarantees to generate all the sequences that have been analyzed.

The algorithm used by KBehavior to extend a current FSA given a new event sequence is based on the identification of sub-machines in the current FSA that generate sub-sequences in the event sequence. Once these relations are identified, the portions of the event sequence that do not correspond to any sub-machine are used to create new branches in the current FSA, so that the updated FSA generates both all event sequences generated by the previous FSA and the new event sequence passed as input. For example, Figure 3.4 shows how a FSA can be extended providing a new event sequence to KBehavior. In this simple example, the portion of the event sequence that does not correspond to any sub-machine lead to the addition of a new branch to the current FSA. In the general case, a FSA



FSA extended with input trace: `Event1-0`, `Event2-1-0`, `Event3`, `Event4-0`, `Event5-0-1`, `Event6-1-1`, `Event7-0-1`, `Event8-1-1`, `Event9-3-1`, `Event10`. *The black state is the initial state. The state with the double border is the final state. The dotted arrows and states* `10` *and* `11` *are added by the extension step.*

Figure 3.4: *An example FSA extended with a new trace.*

is extended by gluing the FSAs obtained from the (recursive) execution of KBehavior on the portions of the event sequence that cannot be associated with any sub-machine. Technical details can be found in [87].

3.3.3 Data flow models

Event flow models or data properties alone do not permit to capture all the aspects of an application behavior. Consider for example Figures 3.5 and 3.6. Figure 3.5 shows an excerpt of a legal web server log with a user that logs in, modifies its profile, and then logs out. Figure 3.6 instead shows a faulty behavior that causes the same server to fail: a user logs into the system and changes another user's profile. The application server is not managing authorizations correctly and permits an invalid behavior that leads to a failure.

```
1 Request from IP 10.0.1.3
2 User Fabrizio logs in with IP 10.0.1.3
3 User Fabrizio logged in with IP 10.0.1.3
4 Modify profile for Fabrizio (IP 10.0.1.3)
5 Profile modified for Fabrizio (IP 10.0.1.3)
6 User Fabrizio logged out (IP 10.0.1.3)
```

Figure 3.5: *Excerpt of a legal web application log*

```
1 Request 10.0.1.3
2 User guest logs in (10.0.1.3)
3 User guest logged in 10.0.1.3
4 Modify profile for Leonardo (10.0.1.3)
5 Profile modified for Leonardo (10.0.1.3)
6 User guest logged out (10.0.1.3)
```

Figure 3.6: *Excerpt of a faulty web application. Events in lines 4 and 5 are anomalous: guest user is modifying another user's profile.*

Tables 3.6 and 3.7 show the events names and data values automatically identified from the log messages in Figures 3.5 and 3.6 as described in Section 3.2.1.

If we build a model only considering the sequence of event names and neglecting parameters we would not be able to detect that the messages in Figure 3.6 are anomalous. In fact, the sequences of event names ex-

#	Id	Readable event	Parameters	
			1st	2nd
1	Event4	Request from IP *	10.0.1.3	
2	Event5	User * logs in with IP *	Fabrizio	10.0.1.3
3	Event6	User * logged in with IP *	Fabrizio	10.0.1.3
4	Event7	Modify profile for * (IP *)	Fabrizio	10.0.1.3
5	Event8	Profile modified for * (IP *)	Fabrizio	10.0.1.3
6	Event9	User * logged out (IP *)	Fabrizio	10.0.1.3

Table 3.6: *Events and data values extracted from log in Figure 3.5 using the regular expressions in Table 3.1.*

#	Id	Corresponding expression	Parameters	
			1st	2nd
10	Event4	Request from IP *	10.0.1.8	
11	Event5	User * logs in with IP *	guest	10.0.1.8
12	Event6	User * logged in with IP *	guest	10.0.1.8
13	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.8
14	Event8	Profile modified for * (IP *)	Leonardo	10.0.1.8
15	Event9	User * logged out (IP *)	guest	10.0.1.8

Table 3.7: *Events and data corresponding to log messages in Figure 3.6 using the regular expressions in Table 3.1*

tracted from the legal case, Figure 3.6, and from the faulty case 3.5 coincide: `<Event4, Event5, Event6, Event7, Event8, Event9>`.

In a similar way, data properties alone cannot recognize the anomalous events. Daikon can derive properties like the non-emptiness of the username and the IP number, e.g. `parameter[1].lenght != null`, which do not help us in the identification of the anomalous events.

Data flow relationships are necessary to detect that events 4 and 5 in Figure 3.6 present an anomalous reference to user name. In order to capture both event and data flow relationships in a model, we defined a technique to derive FSA that includes data-flow information in the model. In order to include data-flow information in the automata, it is necessary to abstract from concrete values reported in traces because they are usually too specific to the executions in which they have been observed, and cannot be directly compared with other executions.

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	10.0.0.1	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	10.0.0.1	5432
3	Event3	Server started		
4	Event4	Request from IP *	10.0.1.3	
5	Event5	User * logs in with IP *	Fabrizio	10.0.1.3
6	Event6	User * logged in with IP *	Fabrizio	10.0.1.3
7	Event7	Modify profile for * (IP *)	Fabrizio	10.0.1.3
8	Event8	Profile modified for * (IP *)	Fabrizio	10.0.1.3
9	Event9	User * logged out (IP *)	Fabrizio	10.0.1.3
10	Event4	Request from IP *	10.0.1.5	
11	Event5	User * logs in with IP *	Leonardo	10.0.1.5
12	Event6	User * logged in with IP *	Leonardo	10.0.1.5
13	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.5
14	Event8	Profile modified for * (IP *)	Leonardo	10.0.1.5
15	Event9	User * logged out (IP *)	Leonardo	10.0.1.5
16	Event10	Server stopped		

Table 3.8: *Event names and data values for log in Figure3.2*

Table 3.8 shows a sequence of events and associated data values ex-

#	Readable event	Event	Parameters	
			1st	2nd
1	Event1	Starting server on *	10.0.0.3	
1	Event2	Starting connection with DB jdbc mysql * * ePortal	10.0.0.3	5432
1	Event3	Server started		
1	Event4	Request form IP *	10.0.1.5	
2	Event5	User * logs in with IP *	Leonardo	10.0.1.5
3	Event6	User * logged in with IP *	Leonardo	10.0.1.5
4	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.5
5	Event8	Profile modified for * (IP *)	Leonardo	10.0.1.5
6	Event9	User * logged out (IP *)	Leonardo	10.0.1.5
7	Event4	Request form IP *	10.0.1.9	
8	Event5	User * logs in with IP *	Mauro	10.0.1.9
9	Event6	User * logged in with IP *	Mauro	10.0.1.9
10	Event7	Modify profile for * (IP *)	Mauro	10.0.1.9
11	Event8	Profile modified for * (IP *)	Mauro	10.0.1.9
12	Event9	User * logged out (IP *)	Mauro	10.0.1.9
12	Event10	Server stopped		

Table 3.9: *Event names and data values for log in Figure 3.3*

tracted from the log in Figure 3.2 as presented in Section 3.2.2. Table 3.9 shows a similar sequence with the only difference that the users logging in are different. If we compare the two sequences by comparing both event names and data values of the two sequences, we detect that they do not match because they do not present the same attributes. Since developers cannot test all the possible combinations of values but only a subset of them models that include raw data values cannot be used to efficiently identify anomalies by comparing executions. Their use would lead to many false positives and cause a big waste of developers time for the inspection of false alarms.

We deal with concrete values associated with events by replacing them with symbols that capture data flow information. To this end we defined three rewriting strategies that capture different data flow patterns: *global ordering*, *relative to instantiation* and *relative to access*.

Rewriting strategies are applied to group of homogeneous attributes, i.e., attributes that work on the same data. The application of the rewriting strategies to group of homogeneous attributes reduces the possibility to incidentally correlate unrelated data, thus decreasing the probability to generate false alarms. We name the groups of homogeneous attributes that should be targeted by a same rewriting strategy data-flow clusters.

Each data-flow cluster is automatically identified by comparing values associated with each variable, and heuristically assuming the existence of a correlation between the variables that share a relevant number of values. In order to detect if two variables are part of the same cluster we consider the set of distinct values associated to each. If the 70% of the values in a set appear also in the other set the two variables are part of the same cluster, we call this a two-elements data-flow cluster. A data-flow cluster is composed by all those variables that are part of a two-elements cluster with another variable of the data-flow cluster. We can graphically represent the data-flow clusters as the connected components in an undirected graph where variables are nodes, and edges indicate that the two connected variables are part of a same two-elements cluster.

If we consider Tables 3.8 and 3.9 we identify 4 data clusters: the first cluster includes the first parameter of Event1 and Event2, the second cluster contains only second parameter of Event2, the third cluster is composed by the first parameter of Event5, Event6, Event7, Event8, Event9, while the fourth cluster groups together the first parameter of Event1 and the

second parameter of Event5, Event6, Event7, Event8, Event9. In the following examples we will apply the data transformations to these clusters separately.

Global Ordering

The simplest way to remove the dependency from concrete values is by consistently replacing concrete values with numbers, respecting the order of appearance. Thus, the first concrete value is rewritten as `1`, the second concrete value is rewritten as `2` if never observed before, otherwise the same number is consistently used, and so on using sequential integer values. According to global ordering both the sequences in Tables 3.8 and 3.9 will be rewritten as shown in Table 3.10 thus permitting to be successfully compared. Figure 3.7 shows the automaton inferred from these rewritten sequences which permits to accept both the two sequences.

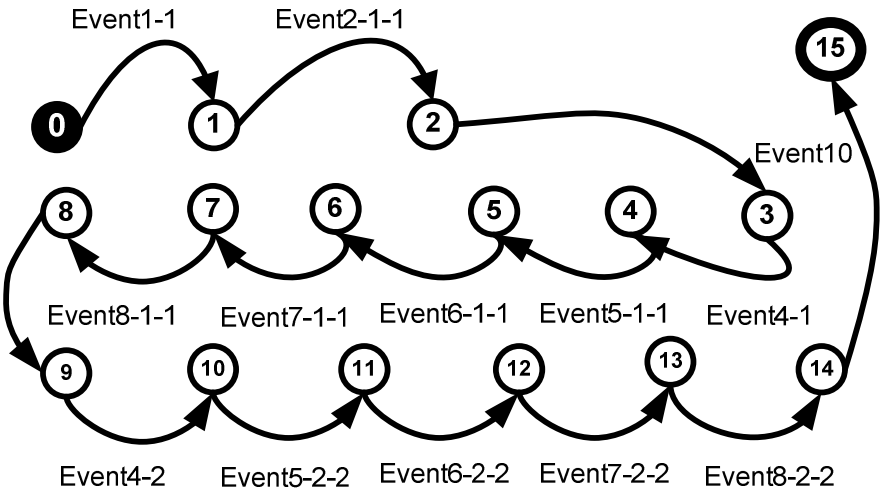


Figure 3.7: *Model inferred from events in Table 3.10.*

Relative to Instantiation

Some interesting patterns that can observed in traces consist of: generate new values, use these values, then introduce further new values, use these values, and so on for many iterations. These patterns cannot be captured by Global Ordering.

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	1	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	1	1
3	Event3	Server started		
4	Event4	Request form IP *	1	
5	Event5	User * logs in with IP *	1	1
6	Event6	User * logged in with IP *	1	1
7	Event7	Modify profile for * (IP *)	1	1
8	Event8	Profile modified for * (IP *)	1	1
9	Event9	User * logged out (IP *)	1	1
10	Event4	Request form IP *	2	
11	Event5	User * logs in with IP *	2	2
12	Event6	User * logged in with IP *	2	2
13	Event7	Modify profile for * (IP *)	2	2
14	Event8	Profile modified for * (IP *)	2	2
15	Event9	User * logged out (IP *)	2	2
16	Event10	Server stopped		

Table 3.10: *Patterns captured by rewriting Tables 3.8 and 3.9 according to the global ordering criterion. Global ordering captures two distinct patterns (highlighted in grey): one in lines 4 to 9, the other in lines 10 to 16.*

Consider the sub-sequences of events at lines 3-9 and 10-19 of Tables 3.8 and 3.9. These two sequences show a repeated pattern indicating that a user logs into the system, changes its profile data, and then logs out.

Consider now Table 3.11, it presents the same behavioral pattern of Tables 3.8 and 3.9. If we rewrite Table 3.11 according to the global ordering criterion we obtain the rewritten sequence in Table 3.12). This table does not match the automaton inferred from data rewritten according to the global ordering criterion, Figure 3.7. This is because Global Ordering does not generalize the creation of new values thus not accepting events from 16 to 21.

To capture such patterns, we rewrite concrete values according to a strategy that focuses on the generation and use of values. In particular, each time a new value is generated, it is rewritten as a 0. If an existing value is detected, it is replaced with a number that indicates the number of new values that have been introduced from its first occurrence plus 1[1]. Thus, the example sequence above would be rewritten as in Table 3.13, which well represents the detected repeated behavior: events 3-9 and 10-19 are rewritten in the same manner.

The model inferred using data rewritten according to relative to instantiation generalize and accept executions in which an arbitrary number of users log-in. Figure 3.8 shows the automaton inferred from the sequence of events in Table 3.13. This model accepts the sequence of events in Table 3.11, rewritten as in Table 3.14: the relative to instantiation criterion permits to accept also sequences that presents more user logging into the system than the ones observed at testing time.

Relative to Access

Relative to instantiation is useful when new concrete values are generated and then used, but does not work well when concrete values are reused multiple times.

Table 3.15 shows the repetition of a same pattern for three times: a user logs into the system, changes its profile data, and then logs out. The pattern is repeated in lines 4 to 9, 11 to 15, and 17 to 21. Table 3.16 shows the events rewritten according to the relative to instantiation criterion. The

[1]We add 1 in order to not put 0 when no new values were introduced between the first occurrence of a value and the others.

#	Readable event	Event	Parameters	
			1st	2nd
1	Event1	Starting server on *	10.0.0.1	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	10.0.0.1	5432
3	Event3	Server started		
4	Event4	Request form IP *	10.0.1.5	
5	Event5	User * logs in with IP *	Leonardo	10.0.1.5
6	Event6	User * logged in with IP *	Leonardo	10.0.1.5
7	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.5
8	Event8	Profile modified for * (IP *)	Leonardo	10.0.1.5
9	Event9	User * logged out (IP *)	Leonardo	10.0.1.5
10	Event4	Request from IP *	10.0.1.8	
11	Event5	User * logs in with IP *	Davide	10.0.1.8
12	Event6	User * logged in with IP *	Davide	10.0.1.8
13	Event7	Modify profile for * (IP *)	Davide	10.0.1.8
14	Event8	Profile modified for * (IP *)	Davide	10.0.1.8
15	Event9	User * logged out (IP *)	Davide	10.0.1.8
16	Event4	Request form IP *	10.0.1.9	
17	Event5	User * logs in with IP *	Mauro	10.0.1.9
18	Event6	User * logged in with IP *	Mauro	10.0.1.9
19	Event7	Modify profile for * (IP *)	Mauro	10.0.1.9
20	Event8	Profile modified for * (IP *)	Mauro	10.0.1.9
21	Event9	User * logged out (IP *)	Mauro	10.0.1.9
22	Event10	Server stopped		

Table 3.11: *Result of the identification of event names and data values for log in Figure 3.3 according to rules in Figure 3.1.*

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	1	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	1	1
3	Event3	Server started		
4	Event4	Request form IP *	1	
5	Event5	User * logs in with IP *	1	1
6	Event6	User * logged in with IP *	1	1
7	Event7	Modify profile for * (IP *)	1	1
8	Event8	Profile modified for * (IP *)	1	1
9	Event9	User * logged out (IP *)	1	1
10	Event4	Request form IP *	2	
11	Event5	User * logs in with IP *	2	2
12	Event6	User * logged in with IP *	2	2
13	Event7	Modify profile for * (IP *)	2	2
14	Event8	Profile modified for * (IP *)	2	2
15	Event9	User * logged out (IP *)	2	2
16	Event4	Request form IP *	3	
17	Event5	User * logs in with IP *	3	3
18	Event6	User * logged in with IP *	3	3
19	Event7	Modify profile for * (IP *)	3	3
20	Event8	Profile modified for * (IP *)	3	3
21	Event9	User * logged out (IP *)	3	3
22	Event10	Server stopped		

Table 3.12: *Table 3.11 rewritten using the global ordering criterion.*

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	0	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	1	0
3	Event3	Server started		
4	Event4	Request form IP *	0	
5	Event5	User * logs in with IP *	0	1
6	Event6	User * logged in with IP *	1	1
7	Event7	Modify profile for * (IP *)	1	1
8	Event8	Profile modified for * (IP *)	1	1
9	Event9	User * logged out (IP *)	1	1
10	Event4	Request form IP *	0	
11	Event5	User * logs in with IP *	0	1
12	Event6	User * logged in with IP *	1	1
13	Event7	Modify profile for * (IP *)	1	1
14	Event8	Profile modified for * (IP *)	1	1
15	Event9	User * logged out (IP *)	1	1
16	Event10	Server stopped		

Table 3.13: *Pattern captured by rewriting Tables 3.8 and 3.9 using the relative to instantiation criterion. The pattern is highlighted in gray and repeated twice: in lines 5 to 9, and in lines 11 to 16.*

#	Readable event	Event	Parameters	
			1st	2nd
1	Event1	Starting server on *	0	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	1	0
3	Event3	Server started		
4	Event4	Request form IP *	0	
5	Event5	User * logs in with IP *	0	1
6	Event6	User * logged in with IP *	1	1
7	Event7	Modify profile for * (IP *)	1	1
8	Event8	Profile modified for * (IP *)	1	1
9	Event9	User * logged out (IP *)	1	1
10	Event4	Request form IP *	0	
11	Event5	User * logs in with IP *	0	1
12	Event6	User * logged in with IP *	1	1
13	Event7	Modify profile for * (IP *)	1	1
14	Event8	Profile modified for * (IP *)	1	1
15	Event9	User * logged out (IP *)	1	1
16	Event4	Request form IP *	0	
17	Event5	User * logs in with IP *	0	1
18	Event6	User * logged in with IP *	1	1
19	Event7	Modify profile for * (IP *)	1	1
20	Event8	Profile modified for * (IP *)	1	1
21	Event9	User * logged out (IP *)	1	1
22	Event10	Server stopped		

Table 3.14: *Pattern captured by rewriting Table 3.11 using the relative to instantiation criterion. The pattern is highlighted in gray and is repeated three times: in lines 4 to 9, in lines 10 to 15, and finally in lines 16 to 21.*

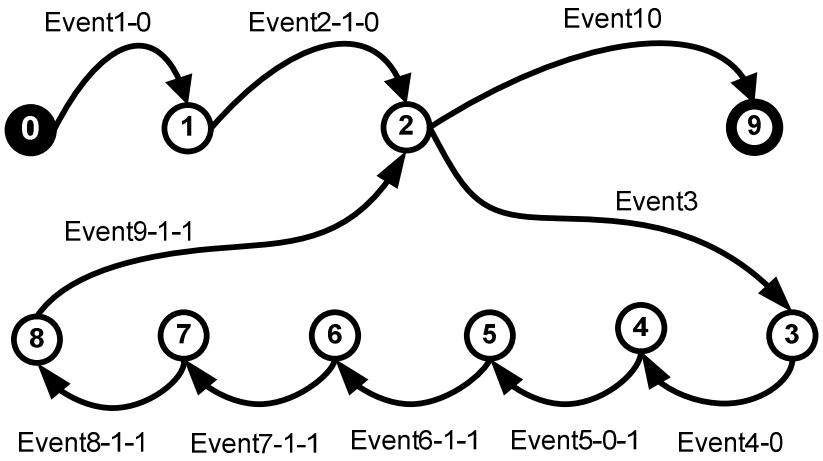


Figure 3.8: *Model inferred from events in Table 3.13.*

rewritten events in lines from 17 to 21 do not match the rewritten events in lines 4 to 9, and 11 to 15. Relative to instantiation in this case does not permit to detect that events 17 to 21 show the same behavioral pattern of the others. This happens because the events from lines 17 to 21 show a user logging in from an IP number seen before (in lines 4 to 9), and this prevents relative to instantiation to capture this pattern. Relative to instantiation in fact rewrites the IP number $10.0.1.3$ with the abstract value 2, thus making events in lines 17 to 21 different from the ones in lines 4 to 9 and 11 to 15.

To capture these patterns, we defined the relative to access rewriting strategy that replaces a concrete value with 0 if it is its first occurrence, otherwise it replaces it with a number that indicates the number of events observed from its last occurrence. The example sequence above would be rewritten as in Table 3.17, that permits to capture that the user behavior follows the same pattern in the three groups of lines: 4 to 9, 10 to 15, and 17 to 21.

Identification of the rewriting strategy

Our approach includes three rewriting strategies. Each strategy focuses on different aspects, and it is hard to say a-priori which strategy can provide the best results for a given set of traces. The rationale for choosing

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	10.0.0.1	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	10.0.0.1	5432
3	Event3	Server started		
4	Event4	Request from IP *	10.0.1.3	
5	Event5	User * logs in with IP *	Fabrizio	10.0.1.3
6	Event6	User * logged in with IP *	Fabrizio	10.0.1.3
7	Event7	Modify profile for * (IP *)	Fabrizio	10.0.1.3
8	Event8	Profile modified for * (IP *)	Fabrizio	10.0.1.3
9	Event9	User * logged out (IP *)	Fabrizio	10.0.1.3
10	Event4	Request from IP *	10.0.1.8	
11	Event5	User * logs in with IP *	Davide	10.0.1.8
12	Event6	User * logged in with IP *	Davide	10.0.1.8
13	Event7	Modify profile for * (IP *)	Davide	10.0.1.8
14	Event8	Profile modified for * (IP *)	Davide	10.0.1.8
15	Event9	User * logged out (IP *)	Davide	10.0.1.8
16	Event4	Request from IP *	10.0.1.3	
17	Event5	User * logs in with IP *	Leonardo	10.0.1.3
18	Event6	User * logged in with IP *	Leonardo	10.0.1.3
19	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.3
20	Event8	Profile modified for * (IP *)	Leonardo	10.0.1.3
21	Event9	User * logged out (IP *)	Leonardo	10.0.1.3
22	Event10	Server stopped		

Table 3.15: *A sequence of events where different users logs in from a same IP number*

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	0	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	1	0
3	Event3	Server started		
4	Event4	Request form IP *	0	
5	Event5	User * logs in with IP *	0	1
6	Event6	User * logged in with IP *	1	1
7	Event7	Modify profile for * (IP *)	1	1
8	Event8	Profile modified for * (IP *)	1	1
9	Event9	User * logged out (IP *)	1	1
10	Event4	Request form IP *	0	
11	Event5	User * logs in with IP *	0	1
12	Event6	User * logged in with IP *	1	1
13	Event7	Modify profile for * (IP *)	1	1
14	Event8	Profile modified for * (IP *)	1	1
15	Event9	User * logged out (IP *)	1	1
16	Event4	Request form IP *	2	
17	Event5	User * logs in with IP *	0	2
18	Event6	User * logged in with IP *	1	2
19	Event7	Modify profile for * (IP *)	1	2
20	Event8	Profile modified for * (IP *)	1	2
21	Event9	User * logged out (IP *)	1	2
22	Event10	Server stopped		

Table 3.16: *Table 3.15 rewritten using the relative to instantiation criterion. Lines 17-21 do not match the pattern of lines 5-9 and 11-15 (in grey): the rewritten values differ from the ones in the pattern.*

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	0	
2	Event2	Starting connection with DB jdbc mysql * * ePortal	1	0
3	Event3	Server started		
4	Event4	Request form IP *	0	
5	Event5	User * logs in with IP *	0	1
6	Event6	User * logged in with IP *	1	1
7	Event7	Modify profile for * (IP *)	1	1
8	Event8	Profile modified for * (IP *)	1	1
9	Event9	User * logged out (IP *)	1	1
10	Event4	Request form IP *	0	
11	Event5	User * logs in with IP *	0	1
12	Event6	User * logged in with IP *	1	1
13	Event7	Modify profile for * (IP *)	1	1
14	Event8	Profile modified for * (IP *)	1	1
15	Event9	User * logged out (IP *)	1	1
16	Event4	Request form IP *	7	
17	Event5	User * logs in with IP *	0	1
18	Event6	User * logged in with IP *	1	1
19	Event7	Modify profile for * (IP *)	1	1
20	Event8	Profile modified for * (IP *)	1	1
21	Event9	User * logged out (IP *)	1	1
22	Event10	Server stopped		

Table 3.17: *Pattern captured by rewriting Table 3.15 using the relative to access criterion. The pattern is highlighted in gray and repeated three times: in lines 5 to 9, in lines 11 to 15, and finally in lines 17 to 21.*

a strategy mainly depends on the nature of the monitored application and the collected data. Since the number of data collected can be extremely high, testers can hardly manually inspect it to choose the proper rewriting strategy. Thus, we developed an automated technique that analyzes the data and identifies the rewriting strategy to be applied to each data-flow cluster.

The rationale for choosing the rewriting strategy is based on the observation that a strategy that applies well to a particular sequence of values would capture its regularity, thus using a small quantity of abstract symbols to rewrite it. Global ordering is effective when a same values set is reused several times, relative to instantiation is effective when new values are created and then used few times, relative to access is effective when values are created and then reused multiple times. To automatically select the rewriting strategy to be used for a given data-flow cluster, we apply the three techniques to the cluster and then we select the one that produces the smallest number of symbols. Since spurious values can be present in data-flow sequences and cause the generation of extra symbols, we select the technique that rewrites more than 50% of concrete values with the less number of symbols, .i.e. the rewriting strategy that can better represent half of the parameter values in a data-flow cluster.

In some cases, data-flow clusters can include attribute values not distributed according to any of our strategies. In that cases, even the best rewriting strategy can provide poor results and cause several false positives in the final analysis. To avoid the selection of an ineffective rewriting strategy (even if it is the best between the three options), we select a rewriting strategy for a data-flow cluster only if attribute values can be rewritten by using at most 10 symbols. If more symbols are necessary for a data-flow cluster, we simply delete attribute values and we work by considering event names only.

Rewriting strategies are very simple and the time required by data analysis is minimal. However, in case a huge amount of data is collected, the time for this analysis can be reduced by only analyzing a randomly selected sample subset of the data.

3.3.4 Models granularity

We infer finite state automata to describe the flow of events and data captured during passing test executions.

A single model that generalizes the legal behavior of the whole application is possible only for small size well tested systems. It is difficult in fact to derive a complete model for a bigger system. For systems of big size providing many functionalities developers cannot test all the combinations of possible inputs or all the possible sequences of activities that the system supports. Consider for example the case of an online store that presents different functionalities: users can register, login, buy a product, etc. The set of all legal sequences obtained by combining these operations is infinite, and testers can only sample the execution space by producing a finite number of test cases. For example, testers can produce test cases that only validate the operations when executed just after the login operations, and not as part of more complex scenarios. Application models derived from data collected during the execution of these tests cannot model all the legal sequences of operations that the application can perform, and can lead to many false positives when used to identify misbehaviors.

In order to infer models that can be effectively used to monitor the application at runtime we can adjust the abstraction level of the model by selecting the events to monitor and by changing the granularity of the model. In the first case by monitoring only a small portion of the events generated by the system we inherently reduce the complexity of the inferred model. In the second case we can derive multiple models that generalize the behavior of the distinct aspects of the system instead of a single model that captures the whole system behavior. By focusing on specific aspects of the system we derive smaller models easier to produce.

The following paragraphs describe the four kinds of models we produce: the system model, and the three finer models, action, component and interface method model.

Application models

This criterion supports the generation of models that capture the behavior of the whole system by describing complex interaction beween multiple components. Application models permit to identify anomalies that span over interaction sequences among different components.

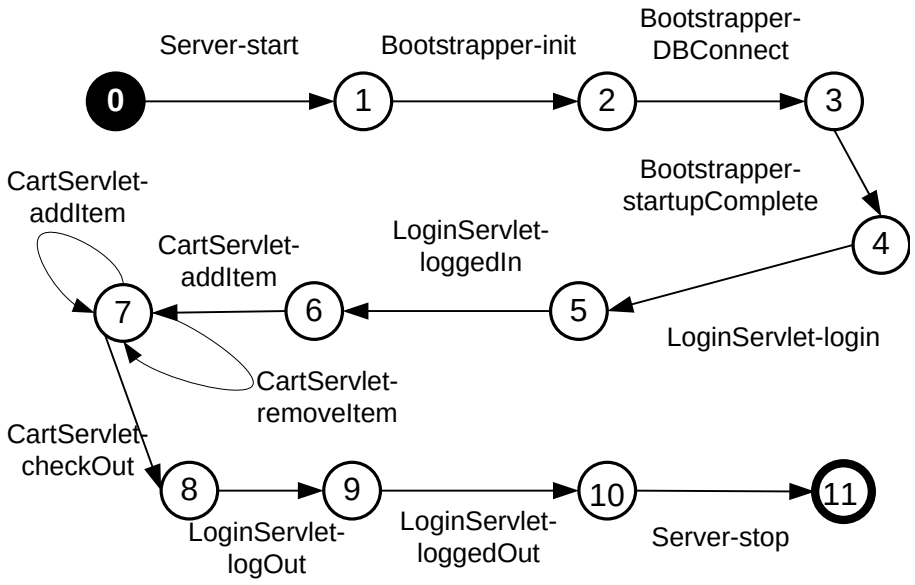


Figure 3.9: *An example application model. Transitions labels indicates the name of a component and the action it is performing (separated by a dash).*

Figure 3.9 shows an example application model that describes the behavior of a web portal. Transitions labels describe both components names and the action they perform. This model describes the legal order of operations performed by the different components. For example it shows that the event "`LoginServlet-login`" must occurr after the event "`Bootstrapper-startupComplete`"; this indicates that a login operation is legal only if performed after the system has been initialized.

Unfortunately it is hard to infer a complete application model. For example to correctly model the fact that after a login another user login can occur, developers should monitor successful executions in which two logins operations are performed in sequence. The same should be done also to capture the fact that a user can make two orders within a same login operation.

In order to simplify the model we omitted data flow information. Including such data, for example the ids of the different items or of different carts used by users concurrently, would increase the complexity of the model and also the difficulty developers have to infer a complete one.

Action models

This criterion supports the generation of models that describe the behavior of the system when a particular user action is performed (e.g. the log-in operation in a web portal).

In order to infer models according to this criterion, we split the sequence of events into different subsequences, each one containing the events generated when a certain user action is performed. To this end we need to be able to identify which user action caused a certain system activity.

If we monitor the system behavior through instrumented monitors we can easily identify the current user action this if the systems has been built according to the Model View Controller design pattern [26]. Systems built according to the MVC design pattern present a control layer that manages user requests; by monitoring the methods of the control layer our framework can automatically identify when a certain user action is performed.

If we capture the application behavior by using information written in log files, we need log messages that mark the beginning and the end of the different actions.

For each kind of action monitored we derive models that generalize the data recorded for that action: finite state automata that accepts the sequences of events associated to that action, boolean properties of the data associated to those events or integrated data and event models that generalize the data and event flow generated for that action.

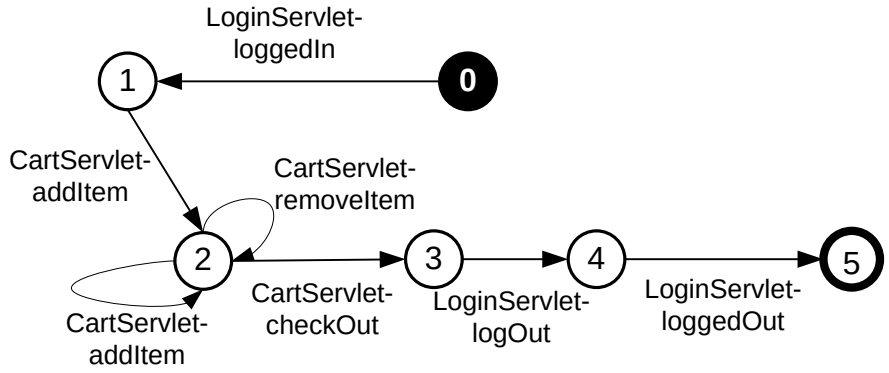


Figure 3.10: *An example action model, the model shows the operations performed when a user buy items from a web portal. Transitions labels indicates the name of a component and the action it is performing (separated by a dash).*

Figure 3.10 shows an example action model that describes the same web portal modeled in Figure 3.9. The action described by this model is the purchasing of items from the web portal. The model correctly indicates that the user must be logged in before being able to purchase items (the first transitions describe successful login operations).

The action model in figure 3.10 has a finer granularity than the applicaion model in Figure 3.9: it does not capture any relations between different login operations, nor between logins and the execution of other components. A consequence of the fine granularity is that the model is more complete with less effort from developer: developers, for example, do not need to run several tests to capture the fact that multiple logins can be executed before the server is stopped. At the same time the model does not permit to capture relations between components involved in different actions, for example it does not indicate that login operations must be performed after the system has been correctly initialized.

Since the action model relates the activities performed by different components during a same user action, it should be necessary to extensively test complex actions before deriving a complete model. For instance the example in Figure 3.10 does not capture the fact that users can perform different purchases without logging out: in order to infer a complete model developers would have to execute different tests in which multiple purchases are performed without a logout operation.

Component models

This criterion supports the analysis of the system by separately considering the behavior of single components. Each model describes the internal behavior of a single monitored component expressed as sequences of operations that can be executed and properties about the data that the component accepts and generates.

To succesfully derive models that capture components behavior, we need to associate each monitored event with the component that generated it. In case we directly monitor the application through instrumented monitors, our monitoring framework automatically adds this information into traces. If we work with log files independently produced by the target application, we require that the log file already includes such information (this requirement is commonly satisfied by many logging systems installed on modern

applications servers like Glassfish or JBoss for example). To simplify the analysis, on the basis of the component information we split the sequences of events into subsequences each containing the events generated by a single component.

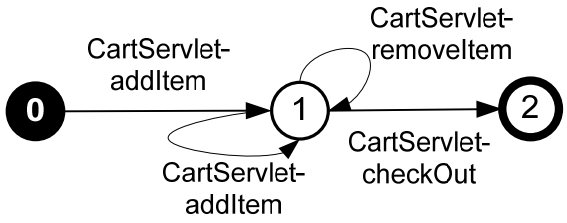


Figure 3.11: *An example component model, the model shows the operations performed by the CartServlet component. Transitions labels indicates the name of a component and the action it is performing (separated by a dash).*

Figure 3.11 shows the model of component CartServlet. The inference of a complete model in this case is less expensive than in the other cases. The model does not derive any constraint over interactions among multiple components, the model in Figure 3.11 for example does not indicate that items can be added to the cart only if a login operation has been performed.

Interface method models

This criterion supports the analysis of the system by separately considering the behavior of single components when a certain method of the interface is invoked. Each model describes the internal behavior of a single monitored component expressed as sequences of operations that it executes (invocations of methods that belong to other components or operations performed by itself).

In order to derive this kind of models it is necessary to collect information about method invocations, in particular we need to know when a method started and when it finished its execution. This information is always available in case we perform monitoring with instrumented monitors, but it is seldom available in case of monitoring of log files. In order to derive these models we split the sequence of monitored events into sub-sequences of events in this way: for each method in the original sequence we generate a sub-sequence that indicates which methods were invoked during its

```
..
ENTER CartServlet.addItem(Item)
ENTER ItemsFacade.getAvailability(Item)
ENTER DB.executeQuery(String)
EXIT DB.executeQuery(String)
EXIT ItemsFacade.getAvailability(Item)
ENTER Cart.addItem(Item,int)
EXIT Cart.addItem(Item,int)
EXIT CartServlet.addItem(Item)
ENTER CartServlet.addItem(Item)
ENTER ItemsFacade.getAvailability(Item)
ENTER DB.executeQuery(String)
EXIT DB.executeQuery(String)
EXIT ItemsFacade.getAvailability(Item)
ENTER Cart.addItem(Item,int)
EXIT Cart.addItem(Item,int)
ENTER ServletException(String)
EXIT ServletException(String)
EXIT CartServlet.addItem(Item)
..
```

Figure 3.12: *A sequence of method invocations monitored with instrumented monitors.*

execution. Then for every monitored method we derive an automata that generalizes all the sequences of operations performed during its execution.

Figure 3.12 shows a sequence of method invocations monitored through instrumented monitors. The sequence has been split in subsequences that represent the methods invoked by each monitored method (see Figure 3.13). During the two different executions of method `CartServlet.addItem(Item)` two different sequences of operations were observed: the first sequence is composed by the invocation of `ItemsFacade.getAvailability(Item)`, and `Cart.addItem(Item,int)` while the second sequence is composed by the invocation of `ItemsFacade.getAvailability(Item)`, and `ServletException(String)`. Method `ItemsFacade.getAvailability(Item)`, in all the two executions monitored, invoked only `DB.executeQuery(String)`. Methods `DB.executeQuery(String)`, `Cart.addItem(Item,int)`, and `ServletException(String)`

Sequences of operations executed by method CartServlet.addItem(Item):
ItemsFacade.getAvailability(Item) Cart.addItem(Item,int)
ItemsFacade.getAvailability(Item) ServletException(String)
Sequences of operations executed by method : ItemsFacade.getAvailability(Item)
DB.executeQuery(String)
DB.executeQuery(String)
Sequences of operations executed by method : DB.executeQuery(String)
Sequences of operations executed by method : Cart.addItem(Item,int)
Sequences of operations executed by method : ServletException(String)

Figure 3.13: *Sequences of operations performed by different methods monitored with instrumented monitors.*

did not execute any monitored method. These sub-sequences are then used to infer the automata in Figure 3.14 that generalize the behavior of the 5 monitored methods.

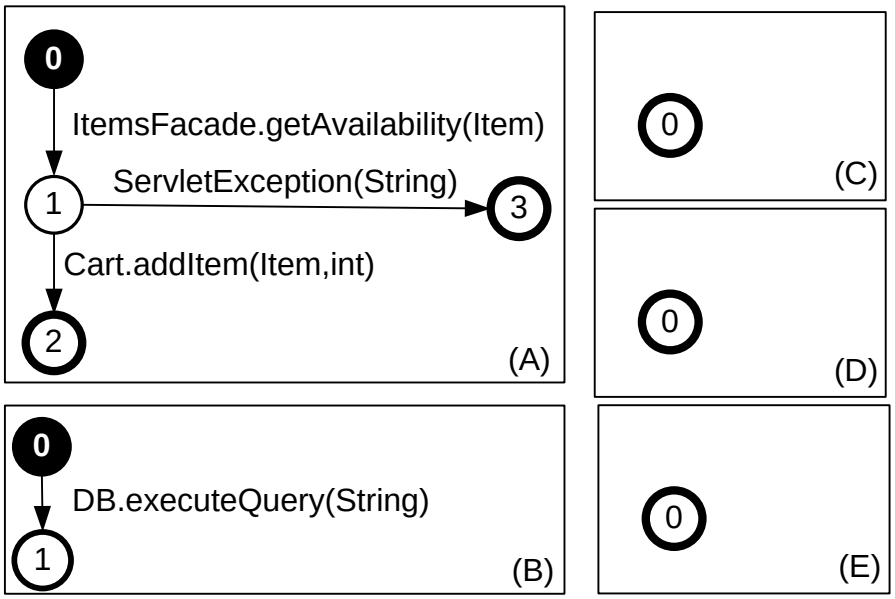


Figure 3.14: *Example interface method models for methods* `CartServlet.addItem(Item)` ***(A)****,* `ItemsFacade.getAvailability(Item)` ***(B)****,* `DB.executeQuery(String)` ***(C)****,* `Cart.addItem(Item,int)` ***(D)****,* `ServletException(String)` ***(E)****.*

Chapter 4

Identify Runtime Misbehavior

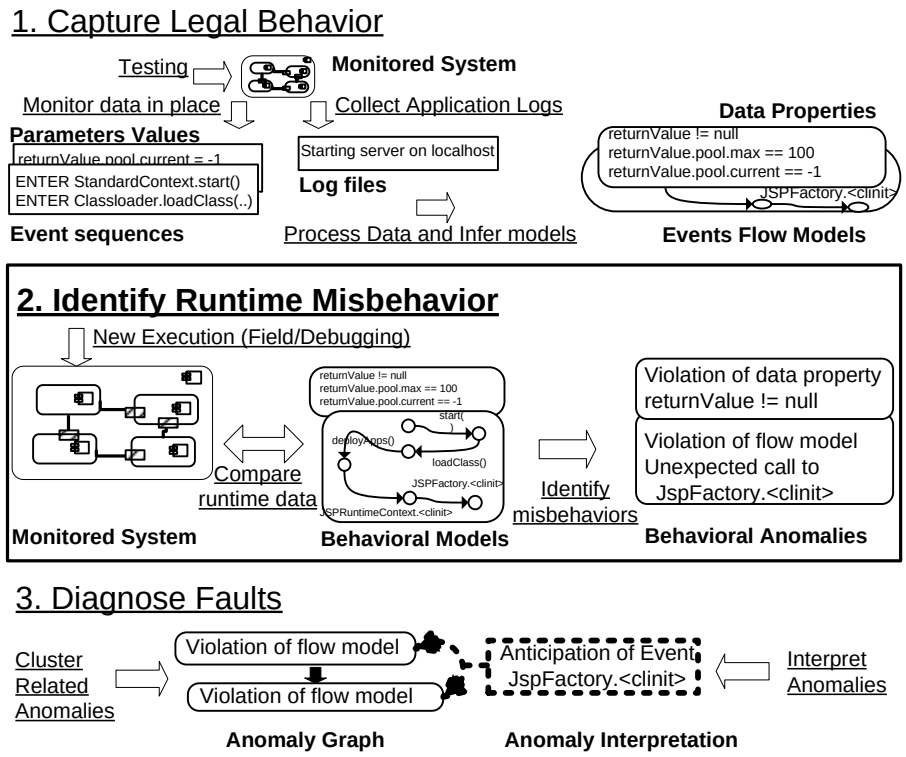


Figure 4.1: *Overview of the Automated Fault Diagnosis Framework. The Identify Runtime Misbehaviors phase is highlighted.*

In field functional failures can often be explained by anomalous data values or sequence of operations that differ from the ones observed during testing. In order to automatically identify such anomalies, we monitor executions and collect information about the anomalous behaviors by comparing the data collected during the failing execution with the models of the expected behavior produced at testing time.

According to the kind of model which is violated by the data observed at runtime we identify three kinds of anomalies: data anomalies, i.e. values that invalidate inferred data properties, anomalous event flows, i.e. event sequences that do not match the corresponding FSA, and anomalous data flow, i.e. sequences of events and data values that do not match a data flow model.

This Chapter describes the techniques adopted to compare actual data, obtained either by monitoring the application or analyzing log files, with the models.

4.1 Identification of data anomalies

In the *behavior capturing* phase, we derive boolean expressions that describe the properties over the attributes that are associated with each monitored event. Each attribute (a variable of the log file or a method parameter) is referenced in the expression by a unique identifier. For example the word `returnValue` in expression `returnValue != null` associated to method `JspFactory.getDefaultFactory()` uniquely identifies the object returned by method `JspFactory.getDefaultFactory()`. In order to discover anomalies in the data observed at runtime, we simply replace the data identifiers with the concrete values and we check if the expression holds. The data values that invalidate an expression are considered anomalous. We keep trace of both the violated expressions and the anomalous data values that violated them. We collect also information that help understanding the context in which the anomaly occurred such as the position of the message in the log or the stack trace of the application when the anomaly is detected. This information permits developers to determine, from the log or directly from the stack, the operations that the system was executing when the anomaly occurred.

The Tomcat case described in Section 2.1 provides an example of a data anomaly useful for the diagnosis of a fault. In the faulty execution of

the Tomcat application server a `null` value is returned by method `JspFactory.getDefaultFactory()`. By monitoring Tomcat, we detect that the `null` value is anomalous because it violates the property `returnValue != null` inferred at testing time (see Section 3.3.1). In addition to the anomalous value, we record the violated property (`JspFactory.getDefaultFactory() returnValue != null`), the attribute causing the property violation (`returnValue`), and the stack trace of the system at the time the anomalous value has been detected (the stack is reported in Figure4.2).

```
1  javax.servlet.jsp.JspFactory.getDefaultFactory:75
2  eltest.ChipsListener.contextInitialized:17
3  org.apache.catalina.core.StandardContext.listenerStart
       :3827
4  org.apache.catalina.core.StandardContext.start:4336
5  org.apache.catalina.core.ContainerBase.
       addChildInternal:760
6  org.apache.catalina.core.ContainerBase.addChild:740
7  org.apache.catalina.core.StandardHost.addChild:525
8  org.apache.catalina.startup.HostConfig.deployWAR:825
9  org.apache.catalina.startup.HostConfig.deployWARs:714
10 org.apache.catalina.startup.HostConfig.deployApps:490
11 org.apache.catalina.startup.HostConfig.start:1138
12 org.apache.catalina.startup.HostConfig.lifecycleEvent
       :311
13 org.apache.catalina.util.LifecycleSupport.
       fireLifecycleEvent:120
14 org.apache.catalina.core.ContainerBase.start:1022
15 org.apache.catalina.core.StandardHost.start:719
16 org.apache.catalina.core.ContainerBase.start:1014
17 org.apache.catalina.core.StandardEngine.start:443
18 org.apache.catalina.core.StandardService.start:451
19 org.apache.catalina.core.StandardServer.start:710
20 org.apache.catalina.startup.Catalina.start:552
21 sun.reflect.NativeMethodAccessorImpl.invoke0:-2
22 sun.reflect.NativeMethodAccessorImpl.invoke:39
23 sun.reflect.DelegatingMethodAccessorImpl.invoke:25
24 java.lang.reflect.Method.invoke:585
25 org.apache.catalina.startup.Bootstrap.start:288
26 org.apache.catalina.startup.Bootstrap.main:413
```

Figure 4.2: *Tomcat stack trace recorded when property* `returnValue != null` *has been violated*

4.2 Identification of anomalous event flows

A straightforward way to identify an anomalous event flow is to check if the finite state automaton that models the legal behaviors accept the sequence of observed events. If the sequence is accepted by the automaton, there are no anomalies. If the sequence is accepted until a given event, for instance the event p, there is an anomaly. The engineer can thus inspect the set of events around p and the sub-machine around the state that has been reached by generating all the symbols of the trace up to event p to understand the reason of the anomaly. We refer to this approach as *coarse grained detection of anomalous flows*.

Coarse grained detection of anomalous flows has a little impact on performance because it only involves a comparison between event names and trasition labels. Unfortunately it has little effectiveness when automata are large, and multiple anomalies or noisy data are present in the analyzed sequence. If a sequence includes multiple anomalies, the first anomaly that is detected hides all successive anomalies. This happens because when an anomaly is detected, the remaining portion of the trace cannot be matched anymore with the automaton and thus the checking is interrupted.

To avoid loss of important information we implemented an algorithm that matches sequences and FSAs on top of the KBehavior extension mechanism. We refer to this approach as *fine grained detection of anomalous flows*. KBehavior is extremely useful in pairing event sequences and sub-machines independently from their positions. For instance, a sub-sequence that is located at the beginning of the trace can be associated with any sub-machine of the FSA. Thus, we use the event sequence to be matched to extend the current model, and we consider all the extensions points as the set of anomalous events that must be inspected by testers. Note that many extensions points of arbitrary length and complexity can be identified, thus making the approach extremely flexible and powerful, even for the identification of multiple anomalies located in different points of the model. In fact, presence of noisy data or multiple anomalies do not hinder effectiveness of the matching process because all anomalous sequences are identified.

Table 4.1 shows a sequence of events observed during a failing execution of an application whose expected behavior is described by the automaton in Figure 4.3. KBheavior extends the FSA according to the sequence

	Event
1	File.open()
2	File.write()
3	File.write()
4	sendFile()
5	File.delete()
6	updateStatus()
7	wait()

Table 4.1: *An anomalous event sequence example*

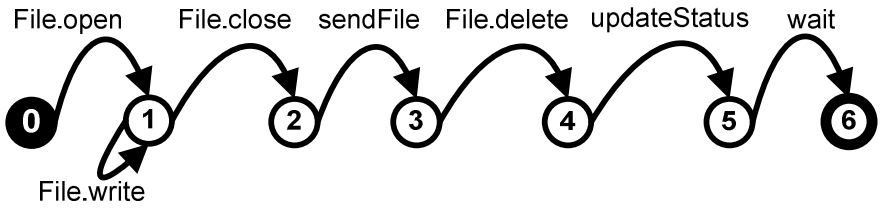


Figure 4.3: *A model of the legal event sequences.*

of events in Table 4.1 as follows. First it detects that events 1 to 3 are accepted by the automaton while event 4, `sendFile()`, is not accepted. After detecting the anomalous event KBehavior looks for an event subsequence, in the remaining portion of the event sequence, that is accepted by the automaton. KBehavior looks for the first matching sub-sequence with a minimal length of K, for this example we consider $K = 2$. The first subsequence accepted by the automaton is made of events `sendFile()` and `File.delete()` and matched the sub-automaton composed by states 3, 4, and 5. After having detected a matching sub-automaton, KBehavior performs the extension of the automaton as shown in Figure 4.4, point *a*, by adding an *epsilon transition* from state 2 to state 3. This extension indicates that no unmatched event is present between the last event matched before the anomaly was detected and the sub-sequence matched in order to extend the automaton. KBehavior then proceeds with the analysis of the remaining portion of the sequence. In state 5 we have an anomaly because we expect a `File.delete()` event but a `File.close()` event is found. KBehavior looks for the next sub-sequence generated by the automaton and finds that the next matching sequence is the one composed by events 6 and 7, `updateStatus()` and `wait()`. After detecting the matching sequence KBehavior performs the extension of the automaton (see Figure 4.4 point *b*): it adds a transition between state 5 and state 6, labeled as `File.close()` because event `File.close()` is not accepted.

We always apply *fine grained detection of anomalous flows* when behavioral data is extracted from log files natively recorded by the application. We have a limited control on the quantity and granularity of information recorded in logs. For this reason we tend to have more noisy data and false positives thus we need to adopt *fine grained detection of anomalous flows* to limit the effects of false positives and noisy data. When we monitor the application through instrumented monitors we can control better the events to monitor and the granularity of the analysis, for this reason in these cases the problem usually does not manifest.

4.3 Identification of anomalous data flows

The approach used to identify anomalous event flows can be successfully applied for the identification of anomalies in the data flow too.

In oder to model the flow of events and data observed during legal ex-

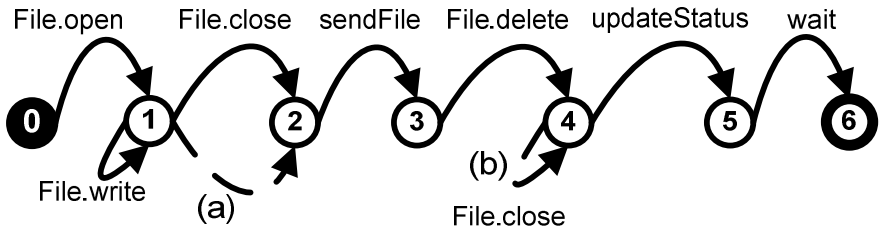


Figure 4.4: *Automaton in Figure 4.3 extended after processing events in Table 4.1. Extensions are drawn as dashed transitions.*

ecutions we build data flow models that include event names and abstract representation of data values in the automata.

Table 4.2 shows a sequence of events and associated parameters observed during a failing execution of a web server. The sequence has been derived by applying the transformation rules in Table 3.1 to the log reported in Figure 4.5. Lines 2, 27, 28 present events not matched by any expression; since these events were never observed during successful executions we replace them with the label `NewEvent`. Line 2 indicates a connection to a `postgresql` database, the connection event does not match the one observed during testing because a different database is used in the field, and the event is not matched by any regular expression. Lines 27 and 28 instead indicates exceptions never observed during testing.

Figure 4.6 shows the automaton describes the legal behaviors observed at testing time. The automaton is inferred from the sequences in Tables 3.8 and 3.9, after applying the relative to access data transformation criterion.

In order to compare the faulty log with the expected model, we need to preprocess the data values collected in the faulty execution with the same transformation rules used to infer the model. Table 4.3 shows the preprocessed data values for the events in Table 4.2.

#	Id	Readable event	Parameters	
			1st	2nd
1	Event1	Starting server on *	10.0.0.1	
2	NewEvent	-	10.0.0.1	3336
3	Event3	Server started		
4	Event4	Request from IP *	10.0.1.3	
5	Event5	User * logs in with IP *	Fabrizio	10.0.1.3
6	Event6	User * logged in with IP *	Fabrizio	10.0.1.3
7	Event7	Modify profile for * (IP *)	Fabrizio	10.0.1.3
8	Event8	Profile modified for * (IP *)	Fabrizio	10.0.1.3
9	Event9	User * logged out (IP *)	Fabrizio	10.0.1.3
10	Event4	Request from IP *	10.0.1.8	
11	Event5	User * logs in with IP *	guest	10.0.1.8
12	Event6	User * logged in with IP *	guest	10.0.1.8
13	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.8
14	Event8	Profile modified for * (IP *)	Leonardo	10.0.1.8
15	Event9	User * logged out (IP *)	guest	10.0.1.8
16	Event4	Request from IP *	10.0.1.7	
17	Event5	User * logs in with IP *	Mauro	10.0.1.7
18	Event6	User * logged in with IP *	Mauro	10.0.1.7
19	Event7	Modify profile for * (IP *)	Mauro	10.0.1.7
20	Event8	Profile modified for * (IP *)	Mauro	10.0.1.7
21	Event9	User * logged out (IP *)	Mauro	10.0.1.7
22	Event4	Request from IP *	10.0.1.9	
23	Event5	User * logs in with IP *	Leonardo	10.0.1.9
24	Event6	User * logged in with IP *	Leonardo	10.0.1.9
25	Event7	Modify profile for * (IP *)	Leonardo	10.0.1.9
26	NewEvent	-		
27	NewEvent	-		

Table 4.2: *Events and data values extracted from log in Figure 4.5 using the regular expressions in Table 3.1.*

```
1  Starting server on 10.0.0.3
2  Starting connection with DB jdbc:postgresql
       ://10.0.0.3:3306/ePortal
3  Server started
4  Request from IP 10.0.1.9
5  User Fabrizio logs in with IP 10.0.1.9
6  User Fabrizio logged in with IP 10.0.1.9
7  Modify profile for Fabrizio (IP 10.0.1.9)
8  Profile modified for Fabrizio (IP 10.0.1.9)
9  User Fabrizio logged out (IP 10.0.1.9)
10 Request from IP 10.0.1.3
11 User guest logs in with IP 10.0.1.3)
12 User guest logged in with IP 10.0.1.3
13 Modify profile for Leonardo (10.0.1.3)
14 Profile modified for Leonardo (10.0.1.3)
15 User guest logged out (10.0.1.3)
16 Request from IP 10.0.1.10
17 User Mauro logs in with IP 10.0.1.10
18 User Mauro logged in with IP 10.0.1.10
19 Modify profile for Mauro (IP 10.0.1.10)
20 Profile modified for Mauro (IP 10.0.1.10)
21 User Mauro logged out (IP 10.0.1.10)
22 Request from IP 10.0.1.7
23 User Leonardo logs in with IP 10.0.1.7
24 User Leonardo logged in with IP 10.0.1.7
25 Modify profile for Leonardo (IP 10.0.1.7)
26 DBException
27 RuntimeError
```

Figure 4.5: *Example of a log collected during a failing execution.*

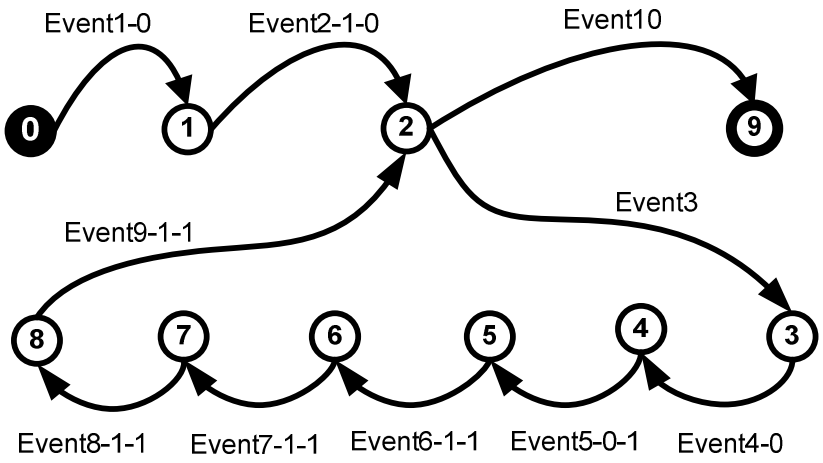


Figure 4.6: *Model of a valid web server behavior.*

Line #	Event	Parameters	
		1st	2nd
1	Event1	0	
2	UnknownEvent		
3	Event3		
4	Event4	0	
5	Event5	0	1
6	Event6	1	1
7	Event7	1	1
8	Event8	1	1
9	Event9	1	1
10	Event4	0	
11	Event5	0	1
12	Event6	1	1
13	Event7	0	1
14	Event8	1	1
15	Event9	3	1
16	Event4	0	
17	Event5	0	1
18	Event6	1	1
19	Event7	1	1
20	Event8	1	1
21	Event9	1	1
22	Event4	0	
23	Event5	2	1
24	Event6	1	1
25	UnknownEvent		
26	UnknownEvent		
27	Event9	1	1

Table 4.3: *Result of the preprocessing of the events in Table 4.2*

As for the identification of the anomalous event flows, we identify the anomalous data flows by extending the automaton that represents legal behaviors with the trace collected in the faulty execution. Figure 4.7 shows the extended automaton.

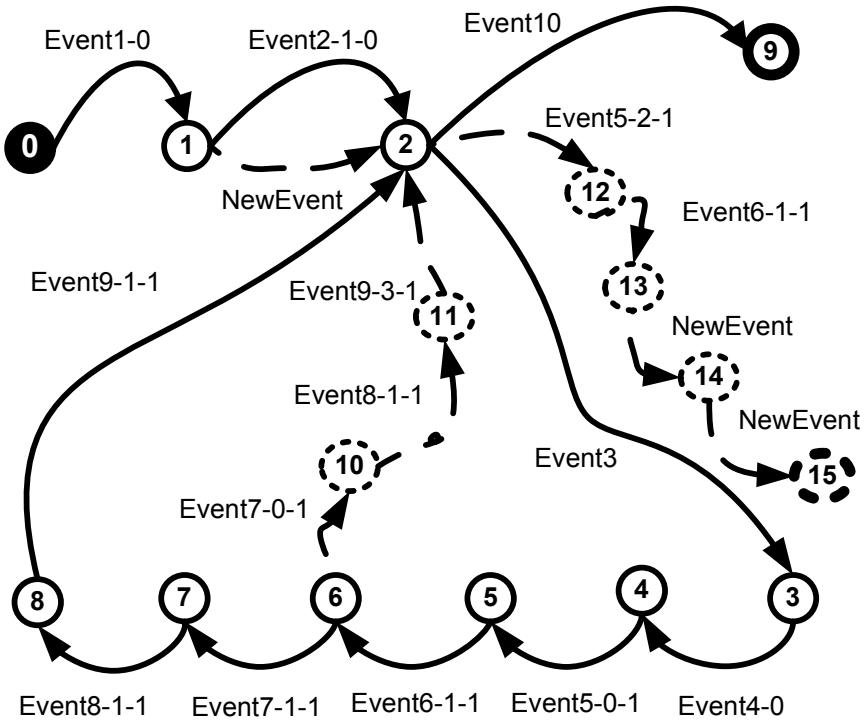


Figure 4.7: *Extension of model 4.6. Dotted lines and states represent extensions.*

The 14th event of the log (see Figure 4.5) presents an anomalous data flow. This event is is a `Modify profile` event that is rewritten as Event7 (see Table 4.3). The event name matches the label expected by the outgoing transition from state 6, but the value assigned to the first parameter does not match expectations.

The integration of data flow information within the model can thus permit to identify an anomalous sequence composed by the 14th, 15th and 16th events which exactly capture the suspicious behavior of a user that modified the profile of another user.

The 23th event is not accepted by the model as well. This happens because the data transformation algorithm detects that the value of the

first parameter has unexpectedly occurred in the past.

The unexpected events resulted in the addition of a tail in the automaton. The tail contains the 26th and 27th event, the exception that caused the crash.

This example showed the usefulness of the adopted approach for the diagnosis of the fault. The first anomaly is not useful for the diagnosis of the problem, it is a false positives due to the difference between the field and the testing environment. The incremental nature of KBehavior permits to also identify the anomalies that follow, thus not beeing limited by the presence of false positives. The second anomaly is directly related to the fault and it represent a user which is not modifying its profile thus permitting to detect that users can modify others' profiles. The final anomaly instead just point developers to understand the failure (it can be useful to determine the components affected by the failure).

Chapter 5

Diagnose Faults

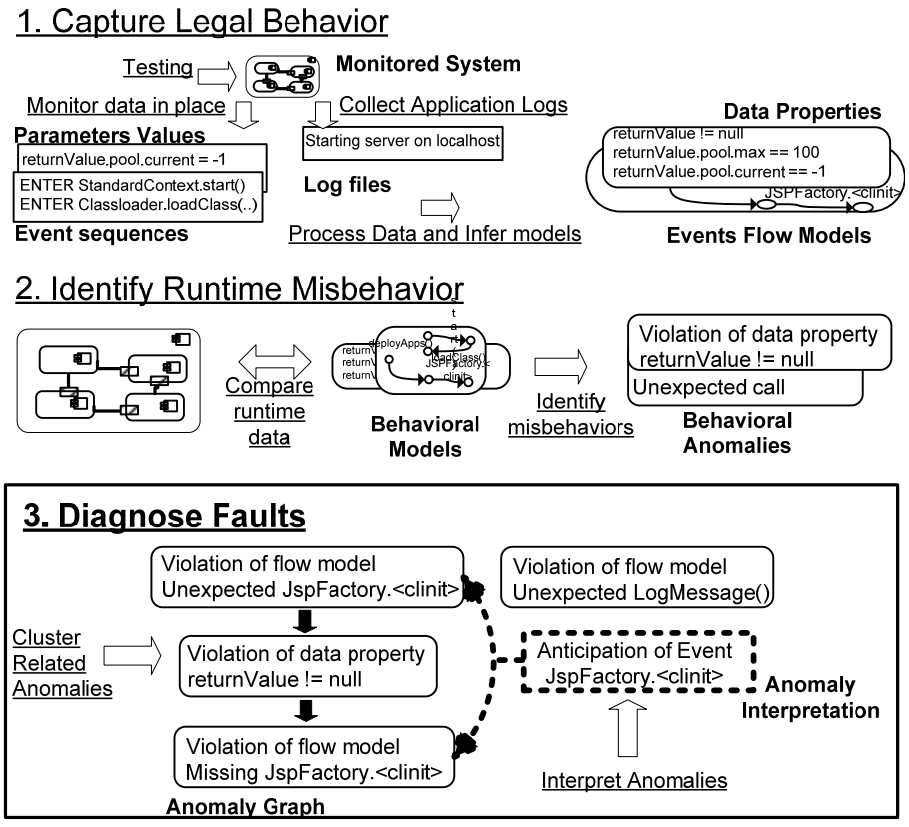


Figure 5.1: *Overview of the Automated Fault Diagnosis Framework. The Diagnose Faults phase is highlighted.*

Existing anomaly detection techniques leave developers the task of interpreting anomalies and understanding eventual causal relations between them to discover the fault location and cause.

Unfortunately anomalies do not always correspond to faulty elements (true positives) but can simply be triggered by correct executions not experienced in the past (false positives). Furthermore many anomalies can be a consequence of few relevant ones. For example unhandled exceptions can cause unexpected termination of method invocations in multiple points.

Manually inspecting all the identified anomalies to distinguish true from false positives can be extremely time-consuming and can significantly reduce the effectiveness of the approach [108, 134]. We overcome this problem with some simple but effective heuristics that eliminate most anomalies that correspond to false positives.

Inspecting a list of anomalies is tedious and error prone. Single anomalies permit to have only a partial vision of the problem; in order to understand what is the fault it is often necessary to inspect all the anomalies identified and identify the causal relations among them. We address this problem by automatically aggregating related anomalies to provide a comprehensive view of the failure under investigation and by automatically providing interpretations that reduce the time required to developers to understand the problem.

This Chapter presents the techniques that we defined to filter false positives, aggregate related anomalies and provide an interpretation of them.

5.1 Filtering False Positives

We automatically distinguish violations that are likely related to faults from false positives with a simple but effective heuristic: *Model violations that occur both during successful and failing executions are likely related to new software behaviors, while model violations that occur only during failing executions are most likely anomalies*. We identify model violations that occur both in successful and failing execution by analyzing logs collected in both cases or by continuously collecting the anomalies observed by the instrumented monitors. We automatically discard model violations that occur during both successful and failing executions, and we analyze violations that occur only during failing ones.

The heuristics is inspired by techniques based on the frequency of violations in faulty and correct executions to filter false positives that have been proposed by Liu and Han [84] and Liblit et al. [83].

5.2 Discovering Relations Between Anomalies

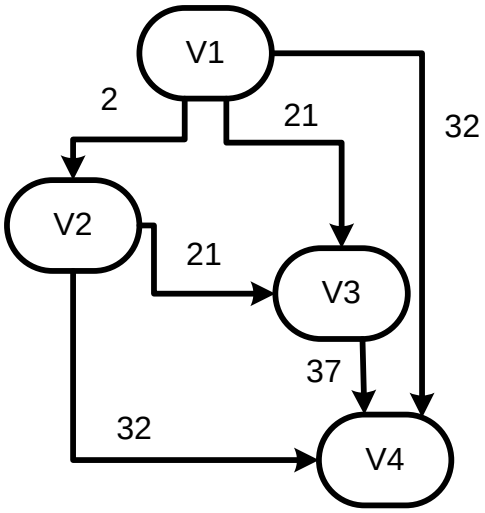
Usually behavioral anomalies do not occur in isolation, but a single problem can be related to several ones. For instance, an erroneous value can violate a data property and then generate an exception that violates event flows or data models, until the program recovers from the exception or fails completely. In addition, single anomalies often provide only partial information about the fault, while sets of related anomalies can better indicate the nature and the localization of the fault. It is thus important to cluster together sets of related anomalies.

The approach we adopt to cluster related anomalies is based on the heuristic observation that *anomalies related to the same fault are usually detected in methods executed from a (close) common ancestor method in the hierarchy of dynamic method invocations*. This happens because when a method executes an operation that fails, usually the failure is followed by a sequence of anomalous actions that violate other models while executing the remaining operations in the method.

To identify clusters of anomalies, we build an *anomaly graph* that represents the cause-effect relation between anomalous events, which are events that violate data properties or event flow models. Nodes of the anomaly graph represent anomalous events, and edges represent likely cause-effect relations between events. Edges have an associated weight that indicates the distance between two events. Anomaly graphs are composed of one or more connected components. A connected component is an acyclic graph that models a set of anomalous events related to the same runtime problem. Acyclic connected components have always at least one root node. The root node of the connected component of an anomaly graph represents the model violation that has likely originated the violations represented by the nodes in the component. Components with more than one root indicate that we identify several potential causes of the runtime problem.

Figure 5.2 shows a simple anomaly graph that we generated while analyzing Tomcat fault #80420 [13]. The anomaly graph includes a single connected component that represents the cause effect relation between the

two experienced model violations.



Description of the anomaly nodes:

V1: Violation of event flow model in state 0 for method
org.apache.catalina.core.StandardContext.start()
unexpected call to javax.servlet.jsp.JspFactory<clinit>()

V2: Violation of data property for method
javax.servlet.jsp.JspFactory.getDefaultFactory()
exit_returnValue != null does not hold

V3: Violation of event flow model in state 24 for method
org.apache.jasper.compiler.JspRuntimeContext<clinit>()
unexpected call to org.apache.jasper.runtime.JspFactoryImpl.<init>()
call to org.apache.jasper.runtime.JspFactoryImpl.<clinit>() was
expected

V4: Violation of event flow model in state 8 for method
org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run()
unexpected call to org.apache.jasper.runtime.JspFactoryImpl.<init>()
call to org.apache.jasper.runtime.JspFactoryImpl.<clinit>() was
expected

Figure 5.2: *The anomaly graph corresponding to the problem experienced on Tomcat 6.0.4.*

Anomaly graphs can address failures that depend both on single as well as multiple faults. Failures that depend on single anomalous events generate anomaly graphs with a single connected component. Failures that depend on multiple faults, for instance many failures that derive either from state-based faults or from the rare combination of multiple events, generate anomaly graphs that can be heuristically refined into multiple connected components, each corresponding to the cause of a failure.

The weights associated to each graph edge indicate the distance between two anomalies. We calculate the distance on the basis of the dynamic call tree of the application, derived from the stack traces recorded when anomalous event occur. Common log file formats do not provide any information about the stack, thus not permitting to perform correlation between anomalies. A possible solution to overcome this limitation is to configure the application logging component to record the application stack every time a log message is logged.

We build an anomaly graph from the dynamic call tree of the failing execution and from the model violations revealed during the failing execution. We first build an initial anomaly graph. When the anomaly graph does not provide sufficient information to isolate the faults, we refine the initial graph by removing likely useless edges to highlight the different faults that affected the program execution.

5.2.1 Dynamic Call Tree

Here we quickly recall the definition of dynamic call tree (DCT) [9]. A dynamic call tree is a tree that represents a program execution, in our case, a failing execution. Nodes represent methods, directed edges represent method invocations. A node n_i is connected to a node n_j if method n_i has invoked method n_j in the considered execution. We define the distance between two nodes n_i and n_j as the minimum number of edges to be traversed to move from n_i to n_j ignoring the direction of the edges. Hereafter we will refer to the distance between two nodes of the dynamic call tree as *DCT distance*. In oder to model different threads of execution within a same DCT we consider all the branches of the three deriving from a same root.

The leafs of the tree are the methods in which model violations occur: methods receiving or generating anomalous data values in case of data properties violations or methods unexpectedly invoked or terminated in

case of anomalous event flows.

For example, Figure 5.3 shows the dynamic call tree corresponding to a known failure of Tomcat 6.0.4. Node 0 is the common root and represent the application itself, while the branches starting at node 1 and node 40 represent two different execution threads. The DCT distance between nodes 26 and 39 is 21 because we need to traverse at least 21 edges to move from node 26 to node 39.

5.2.2 Initial Anomaly Graph

To analyze a failing execution and identify the faults, we first build an initial anomaly graph. The anomaly graph indicates the relation between anomalies that have been revealed during the failing execution and that have not been identified as false positives by the heuristics discussed in the former paragraph. The nodes of the initial graph represent anomalies. The edges indicate the order of occurrence of anomalies. An edge $\langle A, B \rangle$ connects an anomaly A that has occurred before B in the execution. Edges are annotated with the DCT distance between the DCT nodes that corresponds to the anomalies.

Figure 5.2 shows the anomaly graph derived from the Tomcat dynamic call tree of Figure 5.3 that contains four anoamlies at nodes 26, 27, 39, and 47.

As anticipated in Section 2.1 the fault in the Tomcat application server is a delayed variable initialization, which causes the erroneous propagation of a `null` value within the system when method `JspFactory.getDefaultFactory()` is invoked during server startup.

Although easy to describe and fix, this fault is hard to diagnose. In fact, it persisted in few releases before being diagnosed and removed. The initial anomaly graph of Figure 5.2 indicates the problem and its cause. Following paragraph describe how developers can diagnose the fault and identify the fix by inspecting the generated anomaly graph.

Node $V1$ of the anomaly graph corresponds to the violation of the expected event flow model for method `StandardContext.start()`. In state 24 it was observed an unexpected call to `JspFactory.<clinit>()`, which is the static initializer block of class `JspFactory`. In Java the static block initializer of a class invoked only once per execution the first time the class is used. This means that violation $V1$ indicates that class `JspFactory`

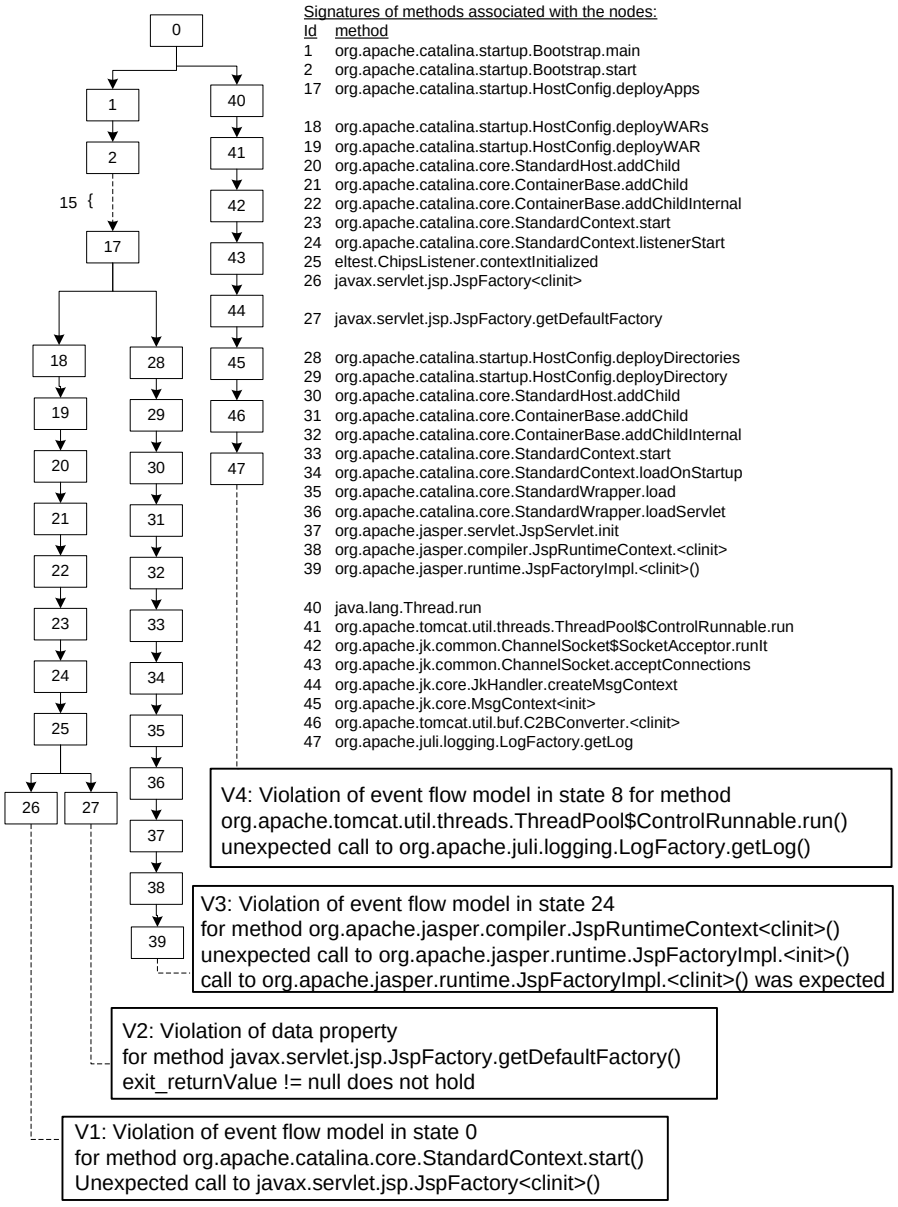


Figure 5.3: *The dynamic call tree corresponding to a failure in Tomcat 6.0.4 (nodes from* 3 *to* 16 *are not displayed for brevity).*

was used before than expected. Node $V2$ corresponds to the violation of the data property `returnValue != null` that is associated with method `JspFactory.getDefaultFactory()`. This violation indicates that Tomcat is unexpectedly returning a `null` value. The edge connecting $V1$ to $V2$ stresses a relation between the anticipated usage of class `JspFactory` and the consequent anomalous `null` value.

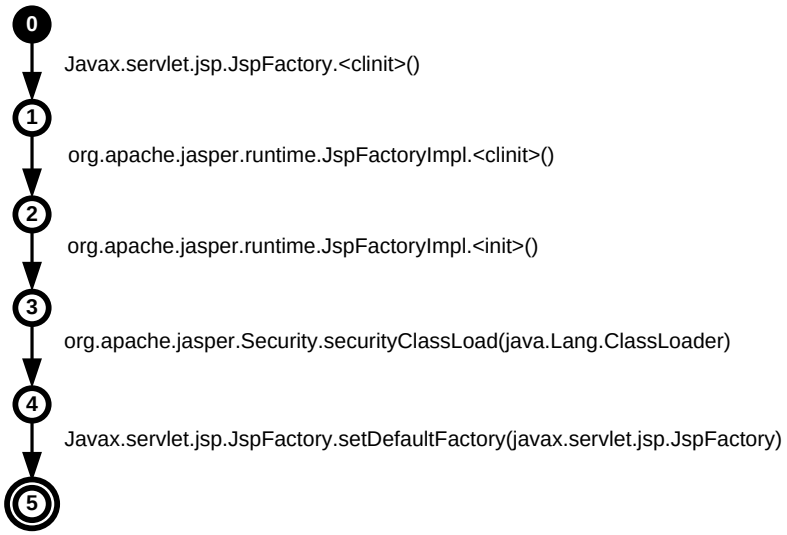


Figure 5.4: *Tomcat event flow model violated in $V3$.*

Node $V3$ corresponds to the violation of the model shown in Figure 5.4 during the execution of method `JspRuntimeContext.<clinit>()`. In state 0 an unexpected call to the `JspFactoryImpl` static class initializer, `JspFactoryImpl.<clinit>()`, is detected while a call to the static initializer of class `JspFactory`, `JspFactory.<clinit>()`, was expected. If we compare the anomalous sequence of events generated by `JspRuntimeContext.<clinit>()` (Table 5.1) with the model associated to method `JspRuntimeContext.<clinit>()` (Figure 5.4) we detect that the whole sequence is accepted by the automaton from State 1, which means that the anomaly depends on the absence of a call to method `JspFactory.<clinit>()`. The identification of the missing event could be faster if the *fine grained anomaly detection approach* is adopted. In this case no manual comparison is required because KBehavior extends the automaton by introducing an epsilon transition that clearly indicates that the whole

sequence is accepted from State 1 (Figure 5.5).

#	Event
1	org.apache.jasper.runtime.JspFactoryImpl.<clinit>()
2	org.apache.jasper.runtime.JspFactoryImpl.<init>()
3	org.apache.jasper.security.SecurityClassLoad.securityClassLoad(ClassLoader)
4	javax.servlet.jsp.JspFactory.setDefaultFactory(javax.servlet.jsp.JspFactory)

Table 5.1: *Sequence of events generated by* `JspRuntimeContext.<clinit>()` *during the faulty execution of Tomcat 6.0.4*

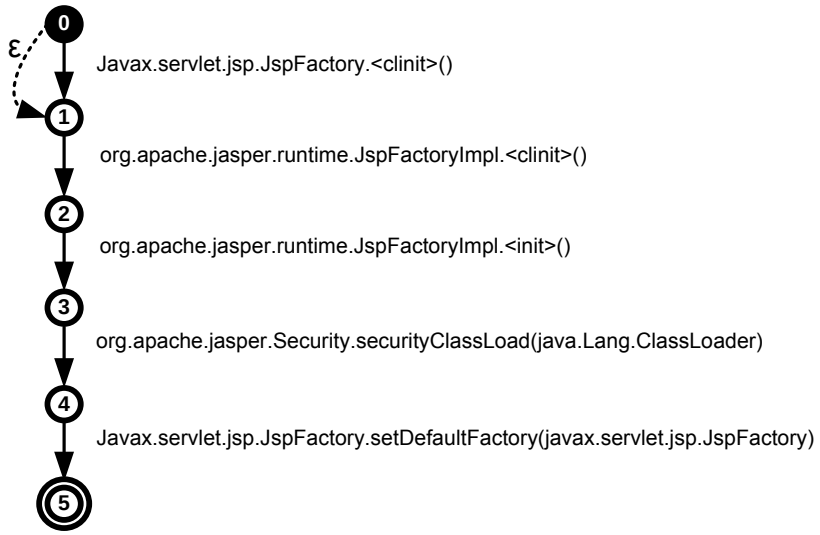


Figure 5.5: *The extended automaton obtained by applying fine analysis to $V3$ data.*

The anomaly graph helps both the diagnosis of the problem and the identification of the solution. The anomaly graph shows that an anticipated usage of `JspFactory`is triggering the failure. Anomalies $V1$ and $V3$ clearly indicate that class `JspFactory` is used for the first time during the execution of method `StandardContext.start()`(as indicated by anomaly $V1$) instead of during the execution of method `JspRuntime-Context.<clinit>()` (as indicated by anomaly $V3$). Anomaly $V2$ instead indicates that the `null` value returned by method `JspFactory.get-DefaultFactory()` is anomalous and is a consequence of the anticipated usage of class `JspFactory` (violation $V1$).

The inspection of the anomaly graph would have helped Tomcat Engineers in fixing the fault. Tomcat engineers evidently expect that method `JspRuntimeContext.<clinit>()` is executed before `StandardContext.start()` but this does not happen. In fact violation $V3$, triggered during the invocation of `JspRuntimeContext.<clinit>()`, occurs after violation $V1$, triggered during the execution of `StandardContext.start()`. The stack trace associated to anomaly $V3$ shows that `JspRuntimeContext.<clinit>()` is not executed by method `Classloader.loadClass(String)` as implementors expect thus guiding them to the resolution of the problem: the invocation of `Classloader.loadClass(String)` is not correct, invocation of method `Class.forName(String)` is required instead.

A last node, $V4$, is present in the anomaly graph, this node indicates an unexpected call to `LogFactory.getLog()`, which probably is a consequence of the failure: Tomcat caught the `NullPointerException` and logged an error message.

Anomaly $V4$, does not help the diagnosis of the problem and may confuse software engineers. Automatically removing anomaly $V4$ from the cluster of the three most important ones would help developers to focus on violations $V1$, $V2$, $V3$ and save time.

The identification of the most important anomalies and the isolation of the less important ones becomes necessary when a large number of anomalies must be analyzed, like in the case of graph in Figure 5.6. This graph corresponds to a known failure of Eclipse 3.3, and need to be refined in order to provide useful information to developers (this problem has been investigated as part of our case studies and it is indicated under the name `E2` in Table 6.2 in Section 6.2). Section 5.2.3 presents the algorithms used to automatically refine the initial anomaly graphs when they are not clear.

5.2.3 Refined Anomaly Graph

We refine initial anomaly graphs by removing the edges that connect distant anomalies, which should not correspond to semantically relevant relations, according to our heuristic.

We identify a weight $BestWeight$ that indicates a threshold above which violations are too distant and thus unrelated, and we remove all edges with weight greater than $BestWeight$. The distance $BestWeight$ that distin-

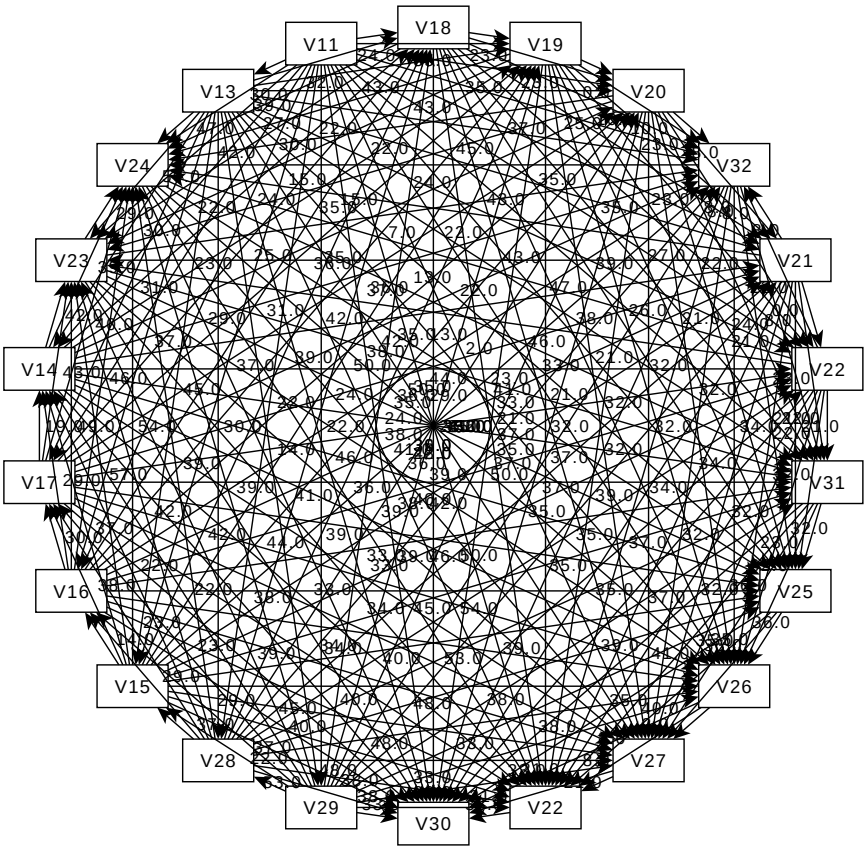


Figure 5.6: *The initial anomaly graph corresponding to the problem experienced with Eclipse 3.3.*

guishes semantically relevant from irrelevant relations strongly depends on the graph topology. Inappropriate values for $BestWeight$ can greatly reduce the effectiveness of the approach. Values too small can fragment the graph, wasting important information, values too large do not significantly reduce the graph that remains hard to interpret.

We use an incremental clustering strategy to identify suitable values for $BestWeight$ [55]. Intuitively, our incremental clustering strategy works as follows. We consider the sequence of the graphs obtained from the initial graph by incrementally removing edges of decreasing weight, starting from the edges with highest weight, till the edges with minimum weight, and considering only the graphs with a different number of weakly connected components.[1] For instance, if removing the edges with weight greater or equal M generates a graph g and removing the edges with weight greater or equal $M - 1$ generates a graph g' with the same number of connected components than g, the graph g' does not belong to the sequence. We ignore graphs with the same number of connected components because we are interested in identifying clusters of violations and not just in removing edges.

We then measure the average *coupling* of the connected components in each anomaly graph. Intuitively, the coupling indicates the degree of focus of the connected components on single problems. Isolated nodes are associated to the best coupling ($coupling = 0$). The coupling increases while adding nodes and edges. Thus, the average coupling of connected components strictly decreases while removing edges in decreasing order of weight. When considering the graphs generated by incrementally removing edges with highest weight, we notice that initially the coupling decreases fast, but at a given point the decrease rate becomes small.

We experimentally noticed that the different decreasing of the coupling is related to the relevance of the cause-effect relations between nodes. When the coupling decreases fast, we remove "heavy" edges that are likely to connect unrelated anomalies. When the decreasing rate of the coupling becomes small, by removing further edges we are likely to disconnect related components thus missing useful information. We compute the value of $BestWeight$ referring to the maximum variation between the coupling of

[1] A weakly connected component is a maximum subgraph where all pairs of nodes are weakly connected, that is connected on the associated undirected graph [29]. Hereafter we will refer to weakly connected components simply as connected components.

consecutive graphs.

This rational is shared with popular clustering algorithms based on *within clustering dispersion* [55].

In the following, we define precisely the clustering strategy. We start by defining anomaly graphs and weakly connected components.
DEF An anomaly graph is a weighted directed graph $G = (N, E, W, m_W)$, where

- $N = \{n_1, \dots n_K\}$ is a finite non-empty set of nodes,
- $E \subseteq N \times N$ is a set of weighted directed edges,
- $W = \mathbb{N}_0$ is the set of weights,
- $m_W : E \to W$ is a mapping that associates weights to edges.
  When the weigh function is clear from the context, we indicate the value of $m_W(e)$ with e_w.

□
DEF An anomaly graph $G = (N, E, W, m_W)$ is weakly connected if for any pair of nodes $n_a, n_b \in N$, there exists an undirected path from n_a to n_b, that is a sequence
$\langle \{n_a, n_1\}, \{n_1, n_2\}, \dots, \{n_{K-1}, n_K\}, \{n_K, n_b\} \rangle$ such that:

- $(n_a, n_1) \in E \vee (n_1, n_a) \in E$,
- $(n_i, n_{i+1}) \in E \vee (n_{i+1}, n_i) \in E, \forall i = 1 \dots K-1$,
- $(n_K, n_b) \in E \vee (n_b, n_K) \in E$,

□
Our initial anomaly graph is weakly connected by construction.

During the refinement process we remove edges, and we disconnect the graph. In a disconnected graph, we can identify a finite set of weakly connected components that we introduce formally to define the concept of coupling of anomaly graphs.
DEF A weakly connected component CC of G is a subgraph of G such that:

- CC is a weakly connected graph,
- It is not possible to add a node in G to CC and preserve weak connectivity for CC.

□

We can now define the concept of coupling of anomaly graphs starting from the definition of coupling of weakly connected components.

DEF The coupling of a weakly connected component $CC = (N, E, W, m_W)$ is the average weight of its edges:

$$coupling(CC) = \begin{cases} \frac{\sum_{e_i \in E} e_{iw}}{|E|} & for |E| > 0 \\ 0 & for |E| = 0 \end{cases}$$

□

DEF The coupling of an anomaly graph G composed of the set of weakly connected components $\{CC_1, \ldots, CC_m\}$ is the average coupling of the connected components:

$$coupling(G) = \frac{\sum_{i=1\ldots m} coupling(CC_i)}{m}$$

□

We can now define the refinement sequence of anomaly graphs as the sequence of graphs obtained from an initial weighted graph by incrementally removing the edges of highest weight. We use the refinement sequence of anomaly graphs to compute *BestWeight*, which is essential for our clustering strategy.

DEF Let $\mathbb{G}raph$ be the set of weighted directed graphs. Given an anomaly graph $G = (N, E, W, m_W) \in \mathbb{G}raph$ and a function $maxW : \mathbb{G}raph \to \mathbb{N}_0$ that returns the maximum of the weights associated with the edges of a graph $G \in \mathbb{G}raph$ (0 for graphs with no edges), we define a function $next : \mathbb{G}raph \to \mathbb{G}raph$ that refines anomaly graphs as follow:

$next(G) = G'$, where $G' = (N, E', W, m'_W)$, with $E' = \{e \in E | m_W(e) < maxW(G)\}$ and m'_W is the restriction of m_W to E'

□

Given an anomaly graph G, we can obtain a sequence of graphs *SG* as follow $G_0 = G$, $G_1 = next(G_0)$, $G_2 = next(G_1)$, ..., $G_k = next(G_{k-1})$. The sequence *SG* terminates with a graph with no edges and $maxW(G_k) = 0$.

We now refine the sequence by removing the graphs that do not include more weakly connected components than their predecessors, since these graphs do not provide additional information to identify runtime problems with respect to the graphs in the refined sequence.

DEF Let $countCC : G \to \mathbb{N}_0$ be the function that counts the number of weakly connected components in a graph. The sequence of graphs $SG =$

$G_0, G_1, \ldots, G_k$ can be refined into the longest sequence $RG = RG_0, \ldots, RG_p$ that satisfies the following properties

- $\forall i = 1, \ldots p \ \exists$ one and only one j, such that $RG_i = G_j$ (we write $corr(RG_i) = j$) (RG is a subset of SG)
- if $i < j, corr(RG_i) < corr(RG_j)$ (RG preserves the order of SG)
- $\forall i = 1, \ldots p-1$, $countCC(RG_i) < countCC(RG_{i+1})$ (graphs in RG contains a strictly increasing number of connected components)

□

Finally, to define $BestWeight$ we introduce the functions $gain$ and $trend$ that measure the change of coupling within a refined sequence RG of graphs. The coupling of the graphs that belong to a refined sequence RG is strictly decreasing, and is zero for the last element of the sequence (a graph with no edges.) The graphs that belong to the head of the refined sequence aggregate sets of nodes that correspond to different problems into single connected components. The graphs that belong to the tail of the refined sequence break sets of nodes that correspond to the same problems into distinct components. We noticed experimentally that the graph that provides the better information about the problems under investigation corresponds to a sudden variation in the change of coupling of the graphs in the sequence. The functions $gain$ and $trend$ compute the changes of coupling, and allow us to identify the sharpest variation that correspond to $BestWeight$, and identifies the anomaly graph of best usage for the software engineers.

Function $gain$ is the ratio between the differences of coupling and the differences of weights of consecutive graphs in a refined sequence of graphs RG.

DEF Let $RG = RG_0, \ldots, RG_n$ be a refined sequence of graphs. For all pairs of consecutive graphs RG_i with $MaxW = w_i$, and RG_{i+1} with $MaxW = w_{i+1}$ the gain from RG_i to RG_{i+1} is

$gain(RG_i, RG_{i+1}) = \frac{cupling(RG_i) - coupling(RG_{i+1})}{w_i - w_{i+1}}$.

We denote $gain(RG_i, RG_{i+1})$ as $gain_{i,i+1}$.

□

Given a sequence of gains obtained from a refined sequence of graphs, the $trend$ in gain at step i is the difference of gains between the considered and the previous step in the sequence.

DEF Let $gain_{0,1}, gain_{1,2}, gain_{2,3}, \ldots gain_{n-1,n}$ be the sequence of gains computed for a refined sequence of graphs $RG_0, RG_1, \ldots, RG_n$,
$trend_i = |gain_{i,i+1} - gain_{i-1,i}|$, for $i \in 2, \ldots n-1$.
□

$BestWeight$ is the weight that corresponds to the highest trend.
DEF Let $trend_2, trend_3, \ldots trend_{n-1}$ be the sequence of trends computed for a refined sequence of graphs $RG_1, RG_2, \ldots, RG_n$,
$BestTrend = k | trend_k \geq trend_i \forall i = 2 \ldots n-1$.
$BestWeight = w_{BestTrend}$
□

The resulting graph obtained by removing the edges with a weight that exceeds $BestWeight$ correspond to $RG_{BestWeight}$, because, by definition, $RG_{BestWeight}$ is the graph whose max weight is $w_{BestWeight}$.

Example of Refined Anomaly Graphs

We illustrate our strategy by considering the case of Tomcat 6.0.4 and Eclipse 3.3 presented in previous Sections.

Tomcat case study

Table 5.2 shows the porperties of the graphs derived by iteratively applying the function *next(G)* to the initial anomaly graph plotted in Figure 5.2. The initial anomaly graph corresponds to graph *G0* in Table 5.2. For each graph we report the number of connected components, and the max weight in the graph. Futhermore we indicate the corresponding *Refined Graph* if any.

Graph	# Connected Components	Max Weight	Coupling	Refined Graph
G_0	1	37	$\frac{\frac{2+21+21+32+32+37}{4}}{1} = 36,25$	RG_0
G_1	1	32	$\frac{\frac{2+21+21+32+32}{4}}{1} = 27$	-
G_2	2	21	$\frac{\frac{2+21+21}{3}+0}{2} = 7,333333333$	RG_1
G_3	3	2	$\frac{\frac{2}{2}+0+0}{3} = 0,333333333$	RG_2
G_4	4	0	$\frac{\frac{2}{2}+0+0}{3} = 0$	RG_3

Table 5.2: *Sequence of Graphs identified for the Tomcat case study. For each graph we report the number of connected components, max weight, the coupling and we indicate if it is a refined graph.*

Table 5.3 presents the four *refined graphs* derived from graphs G_0 .. G_4 (this data obviously match data in Table 5.2).

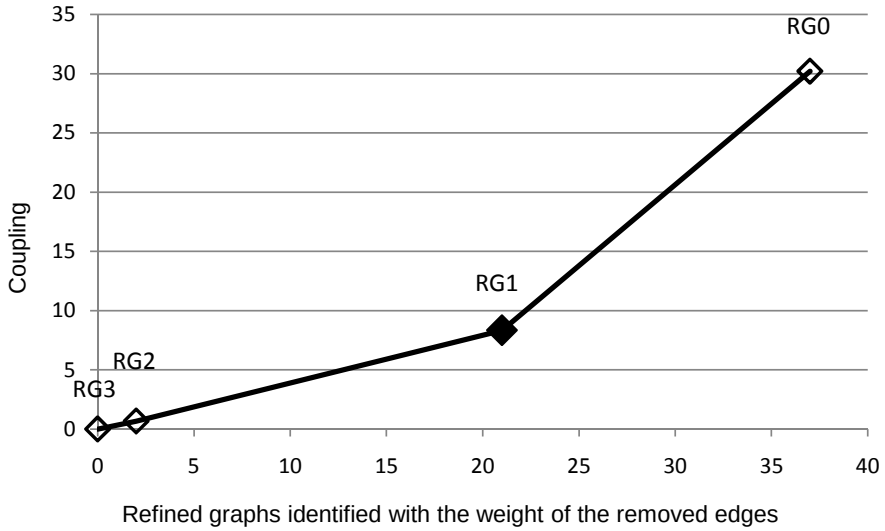


Figure 5.7: *The refined graph for Tomcat case study.*

Refined Graph	Max Weight	Coupling
RG_0	37	30,2
RG_1	21	8,333333333
RG_2	2	0,67
RG_3	0	0

Table 5.3: *Refined Graph identified for the Tomcat case study. For each graph we report the max weight and the coupling.*

Gain Id	Gain value
$gain_{0,1}$	$\frac{30,2-8,333333}{37-21} = 1,366666667$
$gain_{1,2}$	$\frac{8,333333-0,67}{21-2} = 0,403508772$
$gain_{2,3}$	$\frac{0,67-0}{2-0} = 0,333333333$

Table 5.4: *Gains calculated considering the Refined Graphs in Table 5.3.*

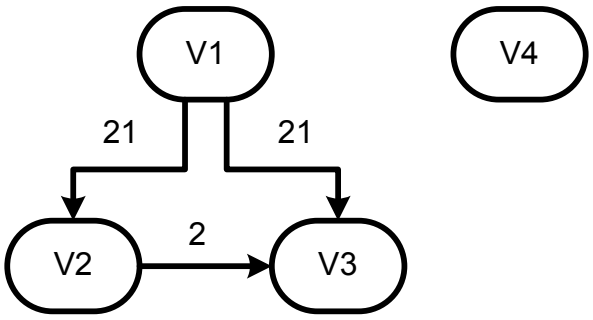
Figure 5.7 plots in a chart the coupling and max weight associated to the four *refined graphs* of Table 5.3. The slope of the lines that connect

Trend Id	Trend Value
$trend_1$	$\|0,403508772 - 1,366666667\| = 0,963157895$
$trend_2$	$\|0,333333333 - 0,403508772\| = 0,070175439$

Table 5.5: *Trends calculated considering the gains in Table 5.4.*

the points correspond to the gain. The best trend $BestTrend$ is the point of highest difference in the gains, and corresponds to the sudden change from a big to a small gain indicated by the black filled point that identify RG_1. The same result is obtained with the formula described before. Tables 5.4 and 5.5 show the gains and trends calculated for the four *refined graphs*. $Trend_1$ has the greatest value, which means that the $BestWeight$ is the one associated to RG_1. For this reason the refined graph which couples anomalies better is RG_1.

Figure 5.8 shows the refined anomaly graph RG_1. This graph is composed by two weakly connected components which clearly separate the relevant anomalies identified during the failing execution of Tomcat 6.0.4 from the anomaly due to logging.



Description of the anomaly nodes:

V1: Violation of event flow model for method
org.apache.catalina.core.StandardContext.start()
unexpected call to javax.servlet.jsp.JspFactory<clinit>()

V2: Violation of data property for method
javax.servlet.jsp.JspFactory.getDefaultFactory()
exit_returnValue != null does not hold

V3: Violation of event flow model for method
org.apache.jasper.compiler.JspRuntimeContext<clinit>()
unexpected call to org.apache.jasper.runtime.JspFactoryImpl.<init>()
call to org.apache.jasper.runtime.JspFactoryImpl.<clinit>() was
expected

V4: Violation of event flow model for method
org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run()
unexpected call to org.apache.jasper.runtime.JspFactoryImpl.<init>()
call to org.apache.jasper.runtime.JspFactoryImpl.<clinit>() was
expected

Figure 5.8: *The refined graph for Tomcat case study.*

Eclipse case study

Figure 5.9 shows the trends obtained for a case study that involves more anomalies, the Eclipse 3.3 case study that we discuss in more details in Section 6.2. In this case the configuration associated with $BestWeight$ consists of 11 weakly connected components that represent the 11 sets of related violations. Figure 5.6 shows the initial (connected) anomaly graph, and Figure 5.10 shows the anomaly graph identified by $BestWeight$ composed of 11 disconnected components.

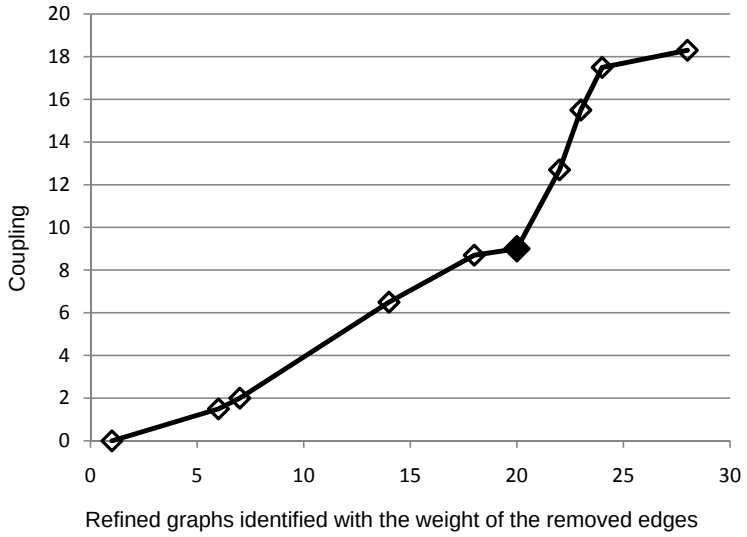


Figure 5.9: *The trend for the Eclipse 3.3. case study.*

Since actual problems correspond to sets of strictly related anomalies, while false positives often correspond to isolated anomalies, we examine the weakly connected components in the final anomaly graph sorted by size, from the biggest to the smallest. In the case of the Eclipse 3.3 case study the relevant connected components that describe the problem are the two largest ones. The remaining 9 weakly connected components include several false positives, and can be ignored by the test designers, who can understand the problem by looking at the first two components only.

In particular, the problem of the Eclipse 3.3 case study is related to a fault that prevents the execution of an undo operation. The undo operation cannot be completed because the EMF plug-in does not record the state of some specific objects, and when the GMF plug-in interacts with the EMF

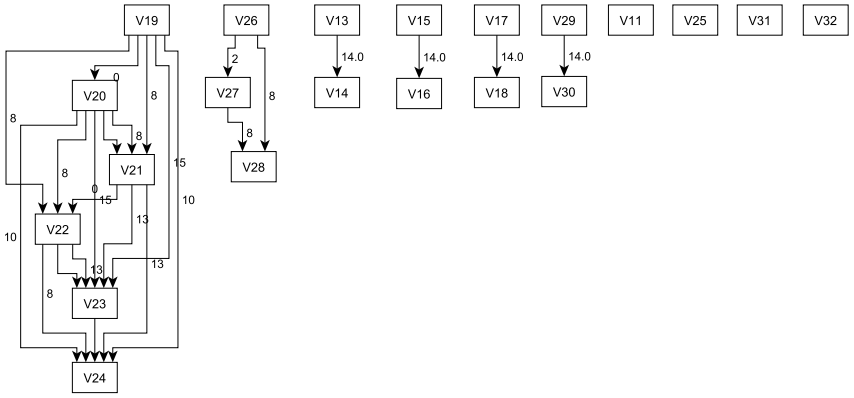


Figure 5.10: *The anomaly graph that corresponds to the problem experienced with Eclipse 3.3. and has been identified by* $BestWeight$.

plug-in to execute the undo, the state of those objects is `null`, while it should hold a value.

The first component of the refined anomaly graph indicates an anomalous `null` value returned by method `NotificationImpl.getNewValue()` at node $V20$, while the second component indicates an anomalous `null` value returned by method `NotificationImpl.getOldValue()` at node $V26$. These are the two methods that are executed by the application to complete an undo. The clustering strategy correctly divided the initial anomaly graph into components and correctly rank them to understand the problem: The first component indicates the call that returns an unexpectedly `null` value, rather than the old value required by the undo operation, and the second component indicates the call that returns an unexpectedly `null` value, rather than the new value required by the undo operation. The two clusters with highest rank carry enough information to understand the cause of the failure and diagnose the faults, while all lower ranked components represent spurious violations that carry information that we did not need to understand the failing execution. We do not provide the details of all the violations represented by the other nodes in the clusters to avoid a useless and tedious discussion of low level implementation details. The nodes in the two top rank clusters represent violations that are consequences of the violations associated with nodes $V20$ and $V26$, and indicate that Eclipse attempts to manage the `null` values (unexpected interactions mostly related to method `NotificationImpl.getNewValue()`).

5.3 Automated Interpretation of Flow Anomalies

Section 5.2.2 showed how developers can diagnose Tomcat fault #40820 by inspecting the clusters of anomalies identified by the proposed approach. The analysis of the anomalous event sequences has been extremely important for the diagnosis of Tomcat fault. In order to diagnose the fault we compared the sequence of events in the neighborhood of violations $V1$ and $V3$ with the corresponding automata and detected that violation $V3$ was caused by the absence of the invocation of method `JspFactory.<clinit>()` which was unexpectedtly anticipated during the execution of `StandardContext.start()` as highlighted by violation $V1$.

The interpretation of flow anomalies is in general the key activity for the diagnosis of erroneous sequences of operations or wrong data flows. Unfortunately the manual comparison of expected and observed sequences can be quite expensive especially if automata are huge and many different event sequences are accepted by the automata in the anomaly neighborhood.

In order to reduce diagnosis effort we automatically interpret model violations by comparing the differences between expected and observed event sequences in the anomaly neighborhood with a predefined set of anomaly patterns. The approach is based on three phases: identify basic interpretations, identify composite interpretations, and prioritize interpretations. We named the approach Automata Violations Analysis, AVA [20].

AVA requires that *fine grained anomaly detection* is performed because it does not only identify the starting point of anomalies but detects whole sequences of anomalous events thus easing the identification of the anomaly neighborhood. Following paragraphs summarize the phases of the approach, while next sections describe each phase in depth.

Identify basic interpretations consists of analyzing both anomalies and inferred models used to identify such anomalies to generate a more relevant, even if simple, interpretation about the unexpected events. For example the detection that violation $V3$ of the Tomcat case study is due to the absence of event `JspFactory.<clinit>()`.

Identify composite interpretations consists of analyzing and combining the available set of basic interpretations to identify richer interpretations of failing executions. In the Tomcat case study violations $V1$ and $V3$ corre-

spond to the addition of an event and its absence in the future. An anticipation of an event represents a more precise interpretations of the failing execution.

Prioritize interpretations consists of ordering basic and composite interpretations according to their likelihood to effectively explain differences between correct and failing executions. Ordering interpretations is useful to start the investigation of failure causes from good interpretations, and avoid the investigation of interpretations with little relevance.

5.3.1 Basic Interpretations

The first operation executed by AVA consists of automatically identifying basic interpretations, i.e., interpretations of the anomalies detected in a failing execution specified according to a catalog of user-understandable basic anomaly patterns. Formally, an interpretation is a triple <*anomaly pattern*, *confidence value*, *extra info*>, where *anomaly pattern* is the name of an anomaly pattern; *confidence value* is a confidence value in the range 0 to 1 that specifies how much the anomaly pattern well describes anomalies observed in the failing execution; and *extra info* specifies additional information that ease the understanding of the interpretation. Basic interpretations are derived by locally analyzing single anomalies (the way we implemented locality is presented in Section 5.3.1).

An example basic interpretation is

<*delete*, 1, *event* `JspFactory.<clinit>()` *generated from state* 0 *has been skipped in the failing execution closest expected sequence: "JspFactory.<clinit>() JspFactoryImpl.<clinit>() JspFactoryImpl.<init>() SecurityClassLoader.securityClassLoad(ClassLoader) JspFactory.setDefaultFactory(JspFactory)"; observed sequence " JspFactoryImpl.<clinit>() JspFactoryImpl.<init>() SecurityClassLoader.securityClassLoad(ClassLoader) JspFactory.setDefaultFactory(JspFactory)";* >.

The first item *delete* specifies that the interpretation refers to the *delete* anomaly pattern. The confidence value 1 indicates perfect confidence on the interpretation, thus the only difference between the expected behavior and the observed behavior consists of deleted events. The last element of the triple specifies that only the event `JspFactory.<clinit>()` has been deleted, and presents both the observed event sequence and the expected correct sequence that is closest to the observed one. Section 5.3.1 describes how we compute confidence values.

The identification of basic interpretations is based on two steps: (1) define scope and expected behaviors, and (2) align and score basic anomaly patterns. *Define scope and identify expected behaviors* consists of identifying the legal behaviors accepted by the inferred FSA that are close to the anomaly under analysis. *Align and score basic patterns* consists of executing a customized string alignment algorithm to compare possible behaviors with the failing execution and measure how well each basic anomaly pat-

terns describes the observed anomaly.

Define scope and expected behaviors

AVA analyzes each anomaly by comparing the expected event sequences and the observed event sequence. To reduce the size of the problem and concentrate the analysis on the target anomaly, the comparison is restricted to the expected and observed events that are close to the place where the anomaly has been detected. The exact scope of the analysis depends on the kind of analyzed anomaly. Since we identify anomalies as extensions of FSAs, the kind of anomalies to be analyzed can be classified according to the type of extension introduced by KBehavior. In particular, KBehavior can extend a model in three possible ways: adding a branch, adding a tail, and adding a final state.

Branch extension A branch extension consists of extending a FSA with the addition of a new branch, as shown in Figure 5.11. A new branch has an *extension start point* and an *extension end point*. The extension start point is the state of the FSA augmented with a new outgoing transition. The extension end point is the state of the FSA augmented with a new incoming transition.

We can have two kinds of branch extensions: branches pointing to the future and branches pointing to the past. We have a branch pointing to the future when the extension end point is reachable from the extension start point. This extension indicates that a new behavior has been observed instead of an expected behavior. We have a branch pointing to the past when the extension end point is not reachable from the extension start point in the original FSA. This extension indicates that an expected behavior has been unexpectedly observed multiple times.

We define the scope of the local analysis depending on the kind of branch that has been added. In the case of a *branch pointing to the future*, we consider behaviors immediately preceding the anomaly, i.e., immediately preceding the extension start point, alternative to the anomaly, i.e., between the extension start and end points, and immediately following the anomaly, i.e., immediately following the extension end point. In particular, we consider the event sequences that can be obtained by concatenating the sequences of maximum length k_1 that can be generated before the exten-

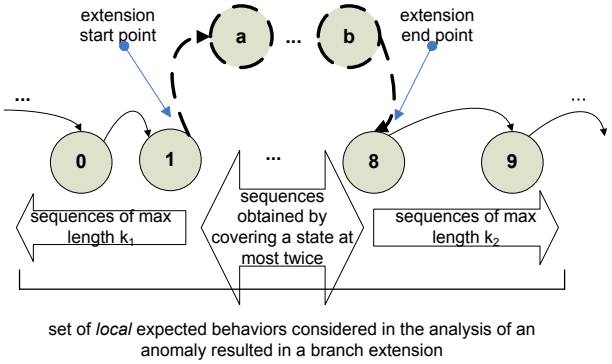


Figure 5.11: *A branch extension pointing to the future*

sion start point, the sequences of events that can be generated between the extension start point and the extension end point by traversing each state at most twice, and the sequences of events of maximum length k_2 that can be generated after the extension end point. Parameters k_1 and k_2 are integer values defined by testers to tune the scope of the local analysis. We extend the boundaries of the local analysis outside extension start and end points because some interpretations can depend on events located before or after the extension points, e.g., postponed events and added events. Figure 5.11 visually shows the set of events that are considered in the local analysis of a branch pointing to the future.

The subsequence of the events observed in the failing execution to be compared with the expected behaviors is selected with an analogous strategy, i.e., we consider the k_1 events before the first event in the branch pointing to the future, the events in the branch pointing to the future, and the k_2 events after the last event in the branch pointing to the future. Figure 5.12 shows the expected and observed sequences that are considered for the analysis with an example FSA and $k_1 = k_2 = 1$.

In the case of a *branch pointing to the past*, we consider behaviors immediately preceding the anomaly, i.e., immediately preceding the extension start point, and immediately following the anomaly, i.e., immediately following the extension start point. In particular, we consider the event sequences that can be obtained by concatenating the sequences of maximum length k_1 that can be generated before the extension start point and the sequences of events of maximum length $k_2 + l$ that can be generated

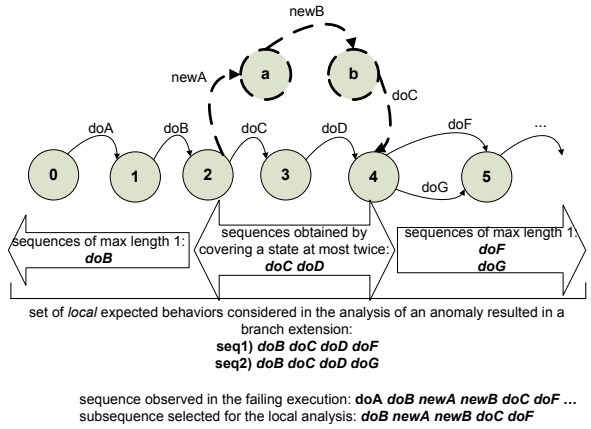


Figure 5.12: *An example set of behaviors considered in the analysis of a branch extension pointing to the future*

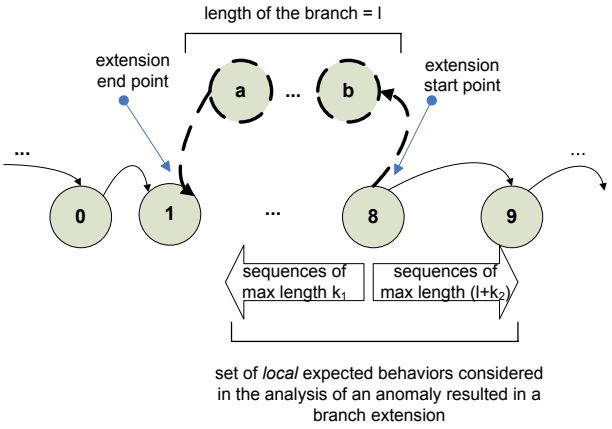


Figure 5.13: *A branch extension pointing to the past*

after the extension start point, where l is the length of the branch pointing to the past. We take into account the length l of the branch generated by the anomaly to consider expected sequences with the same length than the subsequence considered for the failing execution. Figure 5.13 visually shows the set of events that are considered in the local analysis of a branch pointing to the past.

We adopt an analogous strategy to select the subsequence of the events observed in the failing execution to be compared with the expected behaviors. In this case we consider the k_1 events before the first event in the

branch pointing to the past, the l events in the branch and the k_2 events after the last event in the branch. Figure 5.14 shows the expected and observed sequences that are considered for the analysis with an example FSA and $k_1 = k_2 = 2$.

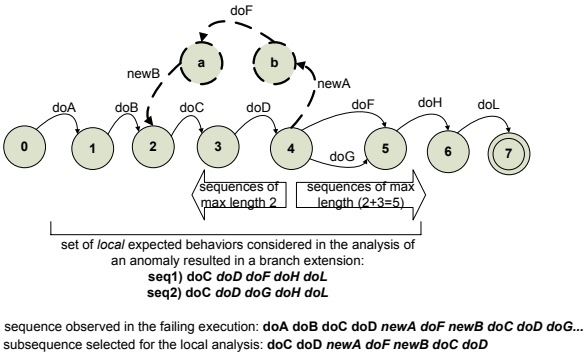


Figure 5.14: *An example set of behaviors considered in the analysis of a branch extension pointing to the past*

Tail extension

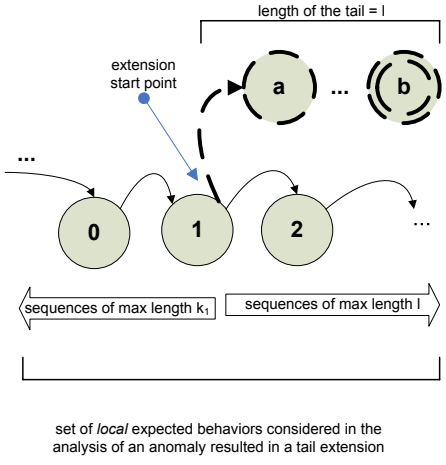


Figure 5.15: *Tail extension*

A tail extension consists of extending a FSA with the addition of a new tail, as shown in Figure 5.15. A new tail has an *extension start point* and

a final state at the end of the tail. The extension start point is the state of the FSA which is augmented with a new outgoing transition.

We define the scope of the analysis as the set of behaviors immediately preceding the anomaly, i.e., immediately preceding the extension start point, and immediately following the anomaly, i.e., immediately following the extension start point. In particular, we consider the event sequences that can be obtained by concatenating the sequences of maximum length k_1 that can be generated before the extension start point with the sequences of events of maximum length l that can be generated after the extension start point, where l is the length of the tail (this choice guarantees the analysis of expected and failing sequences of the same length). Figure 5.15 visually shows the set of events that are considered in the local analysis of a tail.

The subsequence of the observed behavior to be compared with expected behaviors is selected with an analogous strategy, i.e., we consider the k_1 events before the first event in the tail and the l events in the tail. Figure 5.16 shows the expected and observed sequences that are considered for the analysis with an example FSA and $k_1 = 2$.

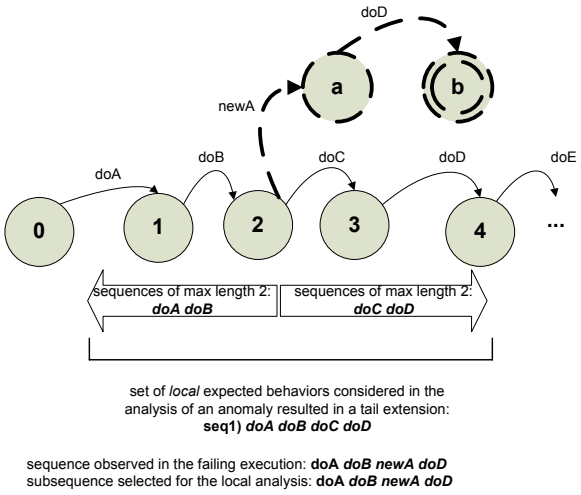


Figure 5.16: *An example set of behaviors considered in the analysis of a tail extension*

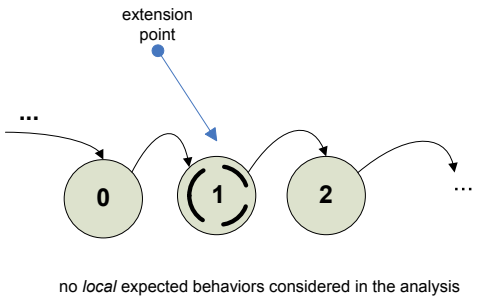


Figure 5.17: *Final state extension*

Final state extension

A final state extension consists of extending a FSA by replacing a regular state with a final state, as shown in Figure 5.17. A final state extension has an *extension point* which is the replaced state.

This kind of extension is a special case because there are no unexpected events: the execution simply terminated at an anticipated time. In this case, local analysis is not needed because the anticipated interruption of the execution is identified with perfect confidence.

Align and score basic patterns

In the previous step, we selected the expected behaviors to be considered for local analysis. In this step, we compare each expected behavior with the sequence observed in the failing execution to interpret differences. These differences will be presented to testers as likely failure causes.

To evaluate the relevance of a given interpretation, we compute its confidence values as the distance between each pair $<$*expected behavior*, *anomalous behavior*$>$. A distance is a real value in the range $0..1$ that indicates how well a given interpretation fits the case under analysis.

Event sequences observed during failing executions often include noise, i.e., events that are not accepted by the model and are not directly related to the investigated failure. Since we identify anomalies by extending a FSA with a new trace by matching subsequences with sub-automata, and adding the extra states and transitions necessary to accept the whole trace, presence of noisy events can modify the sequences in the added branches or even cause the addition of new branches. To be applicable to large and

complex cases, the strategy to recognize anomaly patterns must also work when extensions include noise.

To this end, we use a string alignment algorithm that compares the local expected behavior with the local anomalous behavior and finds a proper fitting of events, despite presence of noisy data. We run our string alignment algorithm with different configurations corresponding to the kind of basic interpretation that is investigated. Each possible alignment of the two analyzed event sequences is associated with a score that indicates how much the two strings are close according to the selected configuration. For example, an observed and an expected sequence that only differs for deleted events will be extremely close according to the delete interpretation.

The string alignment algorithm that we use is a modified version of the Needleman-Wunsch global alignment algorithm [94]. Our modified version can be configured to evaluate the differences between two aligned strings according to different weights. Depending on the alignment, we can have four results when comparing two aligned symbols: match, mismatch, deleteGap, and addGap. Figure 5.18 shows these cases.

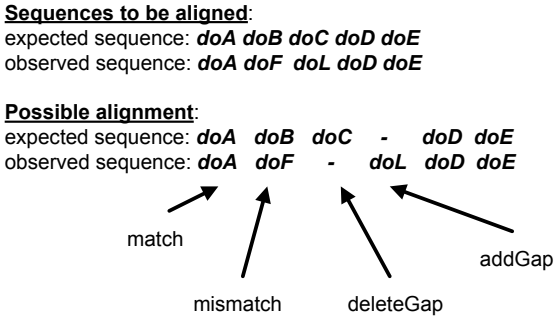


Figure 5.18: *Example alignment.*

We have a match when there is the same symbol at the same position. We have a mismatch when there are two different symbols at the same position. We have a deleteGap when there is a symbol in the expected sequence and a gap in the observed sequence. We have an addGap when there is a gap in the expected sequence and a symbol in the observed sequence. Gaps are automatically introduced by the alignment algorithm to find the best correspondence between the two strings under analysis. We modified the alignment algorithm to support a different evaluation for the

deleteGap and addGap. The original version of the algorithm cannot distinguish between these gaps. We modified the algorithm because gaps have not the same semantics in our domain, e.g., a deleteGap is a strongly favorable indication of deleted events, while an addGap is against this same interpretation.

The possible alignments for two sequences are multiple, the string alignment algorithm finds the solution that maximizes the sum of weights associated to each symbol. For example if we assign a weight of 1 to matches and mismatches, and a weight if 0 to add and delete gaps the algorithm aligns sequences $< \quad T0 \;\; T1 \;\; T2 \;\; T2 \;\; T3 \;\; T4 \;\; T5 \;\; T6 \quad >$ and $< \; T0\, T1\, T7\, T8\, T2\, T3\, T4\, T5\, T6 \; >$ as follows:

$$< - \, T0\, T1\, T2\, T2\, T3\, T4\, T5\, T6 >$$
$$< T0\, T1\, T7\, T8\, T2\, T3\, T4\, T5\, T6 >$$

While if we assign a weight of 1 to matches and add gaps, and a weight if 0 to mismatches and delete gaps the algorithm generates the following aligned sequences:

$$< T0\, T1 \;\; - \;\; - \, T2\, T2\, T3\, T4\, T5\, T6 >$$
$$< T0\, T1\, T7\, T8\, T2 \; - \; T3\, T4\, T5\, T6 >$$

We identify configurations, i.e., sets of weights assigned to the possible result of symbol comparison, that positively evaluate matchings and differences that are consistent with the investigated interpretation, and negatively evaluate differences that are not consistent with the investigated interpretation. We defined four basic interpretations: delete, insert, replace and final state.

Finally we assign a score to each possible alignment by summing the weights associated with each symbol in the aligned sequence and then normalizing in the range $0..1$ by dividing for the number of places (a place can be a symbol or a gap) in the aligned sequences. This score indicates the confidence value of the interpretation.

In the following, we present the configuration of each basic interpretation, and we show with an example how each configuration is evaluated.

It is worth to mention that interpretations are decorated with extra information that specifies the expected sequence that is closest to the anomalous sequence and the start state of the anomalous behavior.

Delete

A delete interpretation is associated with the following weights:

$match = +1$
$mismatch = -1$
$deleteGap = +1$
$addGap = -1$

Since we are analyzing an anomaly, it is not possible that all events match, thus the maximum evaluation is obtained when we have matches and some deleted events. Eventual noise represented by the presence of *addGap* and *mismatch* decreases the value of the interpretation. If we consider violation $V3$ of Tomcat case study, the best alignment according to the configuration of a delete interpretation is shown in Figure 5.19. The example shows the case of a pure deleted event and the normalized value of the interpretation is in effect $\frac{3}{3} = 1$.

Best alignment according to delete for violation $V3$

Expected sequence:	$T6$	$T8$	$T9$
Observed sequence:	$-$	$T8$	$T9$
	delete gap	match	match
	+1	+1	+1

resulting sum = 3
score= $\frac{3}{3} = 1.0$

Figure 5.19: *Example of delete interpretation for violation $V3$ of Tomcat case study. Method names have been replaced with symbols according to Table 5.6.*

Id	Signature
T0	org.apache.catalina.core.ApplicationContext.getFacade()
T1	javax.servlet.ServletContextEvent.<init>(javax.servlet.ServletContext)
T2	javax.servlet.ServletContextEvent.getServletContext()
T3	org.apache.catalina.Wrapper.load()
T4	org.apache.naming.ContextBindings.unbindThread(Object,Object)
T5	org.apache.naming.resources.DirContextURLStreamHandler.unbind()
T6	javax.servlet.jsp.JspFactory.<clinit>()
T7	javax.servlet.jsp.JspFactory.getDefaultFactory()
T8	org.apache.jasper.runtime.JspFactoryImpl.<init>()
T9	org.apache.jasper.security.SecurityClassLoad.securityClassLoad(ClassLoader)

Table 5.6: *Ids used to replace method signatures in Figures 5.19, 5.22, 5.20, 5.23, and in Tables 5.9,and 5.10.*

Insert

An insert interpretation is associated with the following weights:

$match = +1$
$mismatch = -1$
$deleteGap = -1$
$addGap = +1$

Since we are analyzing an anomaly, it is not possible that all events match, thus the maximum evaluation is obtained when we have matches and some added events. Eventual noise represented by the presence of *deleteGap* and *mismatch* decreases the value of the interpretation. Figure 5.20 shows an insert interpretation for violation $V1$ of the Tomcat case study. The example shows that this observed sequence is not a pure insertion because it does not present only added events but also some missing event. In fact, the normalized value of the interpretation is $\frac{6}{8} = 0.75$.

Best alignment according to insert for violation $V1$

Expected:	T0	T1	-	-	-	T3	T4	T5
Observed:	T0	T1	T6	T7	T2	-	T4	T5
	match	match	insert gap	insert gap	insert gap	delete gap	match	match
	+1	+1	+1	+1	+1	+1	-1	+1

resulting sum = 6
score= $\frac{6}{8} = 0.75$

Figure 5.20: *Example of insert interpretation for violation $V1$ of Tomcat case study. Method names have been replaced with symbols according to Table 5.6.*

If we apply the delete interpretation configuration to the same sequence of events (Figure 5.21) we obtain a score of 0.4. The scoreis not zero because we have both matching symbols and a deleted one. The normalized score that we assign to each interpretations permits to compare anomalies and determine the one that fits better in each case: in case of violation $V1$ we can automatically detect that the insertion of three events is more relevant than the deletion of a single one.

Best alignment according to delete for violation $V1$

Expected:	T0	T1	-	-	-	T3	T4	T5
Observed:	T0	T1	T6	T7	T2	-	T4	T5
	match	match	insert	insert	insert	delete	match	match
			gap	gap	gap	gap		
	+1	+1	-1	-1	-1	+1	+1	+1

resulting sum = 4

score= $\frac{4}{8} = 0.5$

Figure 5.21: *Example of delete interpretation for violation* $V1$ *of Tomcat case study. Method names have been replaced with symbols according to Table 5.6.*

Replace

A replace interpretation is associated with the following weights:

$match = +1$
$mismatch = +1$
$deleteGap = -1$
$addGap = -1$

Since we are analyzing an anomaly, it is not possible that all events match, thus the maximum evaluation is obtained when we have matches and some mismatching events. Eventual noise represented by the presence of *deleteGap* and *addGap* decreases the value of the interpretation. Figure 5.22 shows the interpretation of Tomcat violation $V3$ as replace. Violations $V3$ do not have only replaced events, so the resulting score is 0.67. The score for replace interpretation of violation $V3$ is lower than delete interpetation for the same violation (which is 1.0, see Figure 5.20), this means that violation $V3$ is caused more likely by the absence of events rather than by their replacement.

Best alignment according to replace for violation $V3$

Expected sequence:	$T6$	$T8$	$T9$
Observed sequence:	$T8$	$T9$	$-$
	mismatch	mismatch	delete gap
	+1	+1	-1

resulting sum = 2
score= $\frac{2}{3} = 0.66$

Figure 5.22: *Example of replacement interpretation for violation $V3$ of Tomcat case study. Method names have been replaced with symbols according to Table 5.6.*

Final state

A final state interpretation does not require fine grained analysis to be discovered. In fact, every time a final state extension is detected, a final state interpretation with perfect confidence is generated. This interpretation indicates that the application terminated earlier than expected.

Since with high probability a failing execution terminates differently than a regular execution, AVA frequently generates this interpretation. However, it exists at most one occurrence of this interpretation per analyzed failure. Thus, this interpretation does not change the readability of the output, while providing the useful information about the point where the application terminated its execution.

5.3.2 Composite Interpretations

Basic interpretations can be useful in many cases, however there exist further interesting interpretations that can be discovered by combining basic interpretations. We define a composite interpretation as an interpretation that is function of basic or composite interpretations. In the context of this work, we defined three composite interpretations: anticipation, postponements and swap. In the following, we describe composite interpretations and show a few examples.

Anticipation

An anticipation is the case of an event that occurs earlier than expected. We analyze anomalies to discover if an anticipation occurred in all the cases where we detect two basic interpretations I, D, where I is an insert or replacement interpretation, D is a delete or replacement interpretation, and I occurs before D in the sequence of events recorded during the failing execution.

To discover if an anticipation occurred, we define $I.newEvents$ as the sequence of all the events in the observed sequence that are classified as added or mismatched, and $D.removedEvents$ as the sequence that includes all the events in the expected sequence that are classified as deleted or mismatched. To discover if there are common symbols with same order between these two sequences (thus suggesting an anticipation), we align $I.newEvents$ with $D.removedEvents$ by using the following configuration:

$match = +1$
$mismatch = 0$
$deleteGap = 0$
$addGap = 0$

The sequence $alig$ of the symbols that match between $I.newEvents$ and $D.removedEvents$ after string alignment represents the events that have been anticipated. The sequence $notAlig$, which consists of the events that do not match after string alignment, includes events that are only added in the first interpretation, only deleted in the second interpretation or appear with the wrong order. If the number of aligned events

is 0, we cannot have an anticipation.

If we have at least 1 aligned event, we have a behavior that can be interpreted as an anticipation, even if it may be associated with a low confidence value. We compute the confidence value of the anticipation by positively counting the events that are anticipated and the ones that match, and negatively counting the differences that do not contribute to the anticipation. In particular, given that the interpretation I has $I.m$ matches, $I.n$ mismatches, $I.a$ addGaps and $I.d$ deleteGaps, and the interpretation D has $D.m$ matches, $D.n$ mismatches, $D.a$ addGaps and $D.d$ deleteGaps, and the sequence $notAlig$ is further split into $notAlig.g$ gaps and $notAlig.n$ mismatches, we calculate the number of events matching and not matching the pattern and finally compute the confidence value:

$$matching = 2 * \sharp alig + \sharp I.m + \sharp D.m$$

$$mismatching = 2 * \sharp notAlig.n + \sharp notAlig.g + \sharp I.d + \sharp D.a$$

$$confidence = \frac{matching - mismatching}{matching + mismatching}$$

where $\sharp$ indicates the number of events in the sequence that follows.

Note that $I.m$, $I.a$, $D.m$ and $D.d$ do not appear explicitly in the formula for computing confidence value because events in these sequences have been used to generate the sequences $alig$ and $notAlig$. Moreover, events in $alig$ and $notAlig.n$ are multiplied by a factor of 2 because two symbols (1 symbol per analyzed string) contribute to generate the case.

Figure 5.23 shows an example application of the anticipation interpretation. The anticipation regards anomalies $V1$ and $V3$ of Tomcat case study. Anomaly $V1$ is a replace of events $T2$ $T2$ $T3$ with events $T6$ $T7$ $T2$ while anomaly $V3$ corresponds to a delete of event $T6$. AVA detects that event $T6$ has been anticipated ($T6$ is the only event in the *alig* set). Events $T7$ and $T2$ do not match anticipation pattern instead, for this reason the confidence value is lower than 0, 0.6.

Replace interpretation for violation $V1$
Expected sequence: $T0\ T1\ T2\ T2\ T3\ T4\ T5$
Observed sequence: $T0\ T1\ T6\ T7\ T2\ T4\ T5$
Delete interpretation for violation $V3$
Expected sequence: $T6\ T8\ T9$
Observed sequence: $-\ T8\ T9$
Interpretation:
I.newEvents = $\{T6\ T7\ T2\}$
D.removedEvents = $\{T6\}$
I.newEvents alignment = $\{T6\ T7\ T2\}$
D.removedEvents alignment = $\{T6\ -\ -\}$

I.m = 4
I.d = 0
D.m = 2
D.a = 0

alig = $\{T6\}$
notAlig = $\{T7\ T2\}$
notAlig.g = $\{T7\ T2\}$
notAlig.n = $\{\}$

matching = $I.m + D.m + 2 * alig = 4 + 2 + 2 * 1 = 8$
mismatching = $I.d + D.a + 2 * notAlig.n + notAlig.g = 0 + 0 + 2 * 0 + 2 = 2$
confidence= $\frac{match - mismatch}{match + mismatch} = \frac{6}{10} = 0.6$

Figure 5.23: *J*

Example of anticipation interpretation for violations $V1$ and $V3$ of the Tomcat case study.

Postponement

The case of the postponement is symmetric to the anticipation.

A postponement consists of events that occur later than expected. We analyze anomalies to discover if a postponement occurred in all the cases where we detect two basic interpretations D, I, where D is a delete or replacement interpretation, I is an insert or replacement interpretation, and D occurs before I in the trace recorded during the failing execution.

To discover if a postponement occurred, we define $D.removedEvents$ as the sequence that includes all the events in the expected sequence that are classified as deleted or mismatched, and $I.newEvents$ as the sequence of all the events in the observed sequence that are classified as added or mismatched. To discover if there are common symbols with proper ordering between these two sequences (thus suggesting a postponement), we align $D.removedEvents$ and $I.newEvents$ with the same configuration used for the anticipation.

The sequence $alig$ of the symbols that match after string alignment represents the events that have been postponed. The sequence $notAlig$ indicates events that are only deleted in the first interpretation, only added in the second interpretation or appear with the wrong order. If the number of aligned events is 0, we cannot have a postponement.

If we have at least 1 aligned event, we have a behavior that can be interpret as a postponement, even if it may be associated with a low confidence value. We compute the confidence value of the postponement by positively counting the events that are postponement and the ones that match, and negatively counting the differences that do not contribute to the postponement, resulting in the same formula used for the anticipation.

Swap

A swap is the case of a sequence of events that are anticipated by replacing others that are postponed. This interpretation can be easily discovered by combining the anticipation with the postponement. In particular, if both such interpretations have been discovered for an observed trace, we also generate a swap interpretation with a confidence value given by the average value of the confidence values for the anticipation and the postponement.

5.3.3 Prioritize Interpretations

Discovery of basic and composite interpretations ends with a list of possible interpretations that have a confidence value greater than 0. The list of candidate interpretations can be large: depending from the structure of the inferred FSA each anomaly can be compared with several possible expected behaviors, and each pair $<$ *anomalous behavior*, *expected behavior* $>$ can have multiple explanations.

Let us consider the Tomcat case study again. Figures 5.24 and 5.25 show the extensions performed by applying fine grained anomaly detection to the automata associated to methods `StandardContext.start()` and `JspRuntimeContext.<clinit>()` respectively. Tables 5.7 and 5.8 show the sequences of events that violated the two models. By comparing the neighborhood of the anomalies with the violated models, AVA identifies three pairs of <exepected - observed> event sequences. Considering that we have defined 4 possible types of basic interpretations, and 3 types of composites interpretations, the technique may produce tens of interpretations with confidence values greater than 0 (Table 5.9 lists all the possible interpretations for the Tomcat case study which are more that 60 for two anomalies only).

#	Event
58	org.apache.AnnotationProcessor.processAnnotations(Object)
59	org.apache.AnnotationProcessor.postConstruct(Object)
60	org.apache.catalina.core.ApplicationContext.getFacade()
61	javax.servlet.ServletContextEvent<init>(ServletContext)
62	**javax.servlet.jsp.JspFactory.<clinit>()**
63	**javax.servlet.jsp.JspFactory.getDefaultFactory()**
64	**javax.servlet.ServletContextEvent.getServletContext()**
65	org.apache.naming.ContextBindings.unbindThread(Object,Object)
66	org.apache.naming.resources.DirContextURLStreamHandler.unbind()

Table 5.7: *Excerpt of the sequence of events observed during the anomalous execution of method* `StandardContext.start()`*. We highlighted in bold the anomalous events.*

To avoid having testers inspecting data with little interest, we rework the output produced by AVA in two steps: data aggregation and prioritization of the results.

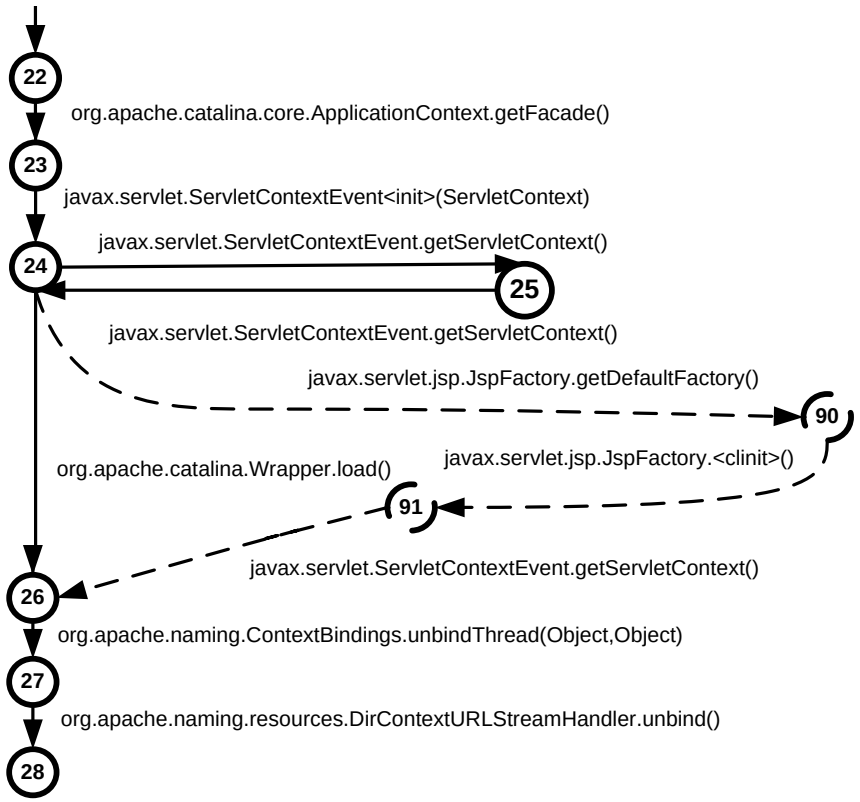


Figure 5.24: *Fine grained analysis of violation $V1$ of the Tomcat case study. Dotted arrows and states represent the anomalous branches inserted. For space reasons we reported only the modified portion of* `StandardContext.start()` *FSA.*

#	Event
1	org.apache.jasper.runtime.JspFactoryImpl.<clinit>()
2	org.apache.jasper.runtime.JspFactoryImpl.<init>()
3	org.apache.jasper.security.SecurityClassLoad.securityClassLoad(ClassLoader)
4	javax.servlet.jsp.JspFactory.setDefaultFactory(javax.servlet.jsp.JspFactory)

Table 5.8: *Excerpt of the sequence of events observed during the anomalous execution of method* `JspRuntimeContext.<clinit>()`.

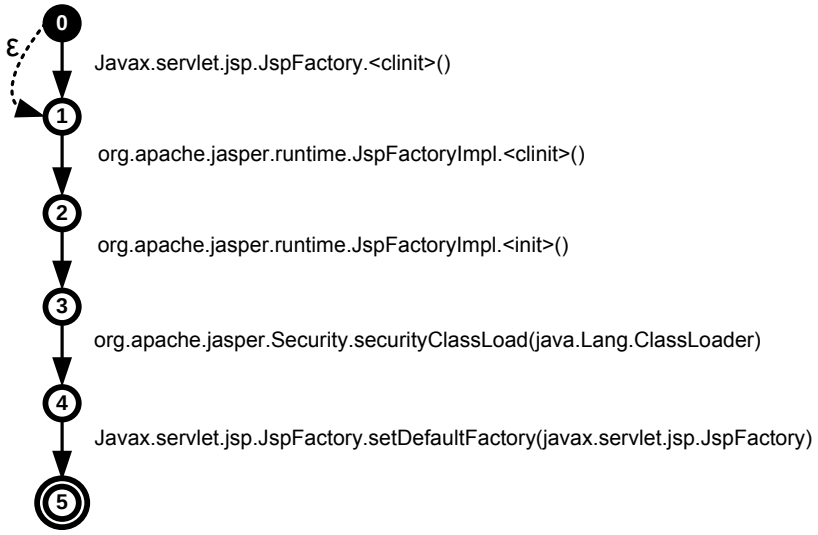


Figure 5.25: *Fine grained analysis of violation $V3$ of the Tomcat case study. The dotted arrow represents the anomalous branch inserted.*

Data aggregation consists of building a unique representation of equivalent interpretations. The same interpretations for a same anomaly are represented as a single interpretation associated with the best confidence value. For instance, violation $V1$ for Tomcat case study has two $insert$ interpretations with confidence values 0.75, and 0.56; in this case we display $insert$ only once with confidence value 0.75. If necessary, testers can expand this information and see the other equivalent interpretations that have been hidden. This reduction is applied to both basic and composite interpretations.

Finally, the overall set of resulting interpretations are globally ordered according to their confidence values. Since testers use AVA to look for explanations of observed failures, we present first explanations with high confidence values, thus clearly explaining differences between the expected behavior and the observed behavior, then the ones with low confidence value, which can be harder to read.

Table 5.10 shows the results produced by AVA for Tomcat case study. Violation $V3$ is correctly identified as a pure insertion of events. Violation $V1$ instead is identified as a replacement (line 2): this interpretation does not help developers to pinpoint the source of the problem, but just under-

lines the fact that the system is executing unexpected tasks instead of the expected ones. In line 3 AVA identifies anomaly $V1$ as an anticipation even if not "pure" (three methods are unexpectedly invoked but another one is not executed because the failure alters the expected execution flow).

The identification of the problem is provided in line 4: AVA correctly detects that $V1$ and $V3$ correspond to the anticipation of an event. The score of the interpretation is lower than 1.0, which indicates that the anticipation is not pure. In fact the anticipated execution of `JspFactory.<clinit>()` is caused by an unexpected invocation of method `JspFactory.getDefaultFactory()` that causes the execution of methods `JspFactory.<clinit>()` and return the null value which causes the failure.

Id	Violation	Interpretation	Score	Extra information
I1	V1	Delete	0.25	Expected:T0 T1 - - - T3 T4 T5. Observed:T0 T1 T6 T7 T2 - T4 T5
I2	V1	Insert	0.75	Expected:T0 T1 - - - T3 T4 T5. Observed: T0 T1 T6 T7 T2 - T4 T5
I3	V1	Replace	0.43	Expected: - - T0 T1 T3 T4 T5. Observed: T0 T1 T6 T7 T2 T4 T5.
I4	V1	Delete	0.56	Expected:T0 T1 - - T2 T2 T3 T4 T5. Observed:T0 T1 T6 T7 T2 - - T4 T5.
I5	V1	Insert	0.56	Expected:T0 T1 - - T2 T2 T3 T4 T5. Observed:T0 T1 T6 T7 T2 - - T4 T5
I6	V1	Replace	1	Expected:T0 T1 T2 T2 T3 T4 T5. Observed:T0 T1 T6 T7 T2 T4 T5.
I7	V3	Delete	1	Expected: T6 T8 T9. Observed: - T8 T9.
I8	V3	Insert	0.33	Expected: T6 T8 T9. Observed: - T8 T9.
I9	V3	Replace	0.33	Expected: T6 T8 T9. Observed: T8 T9 - .
I10	V1-V3	Swap	0.27	Composed by I1 I7
I11	V1-V3	Postponement	0.9	Composed by I1 I8
I12	V1-V3	Swap	0.27	Composed by I1 I8
I13	V1-V3	Anticipation	0.45	Composed by I2 I7
I14	V1-V3	Swap	0.27	Composed by I2 I7
I15	V1-V3	Swap	0.27	Composed by I2 I8
I16	V1-V3	Anticipation	0.9	Composed by I2 I9
I17	V1-V3	Anticipation	0.2	Composed by I3 I7
I18	V1-V3	Swap	0.33	Composed by I4 I7
I19	V1-V3	Postponement	0.17	Composed by I4 I8
I20	V1-V3	Swap	0.33	Composed by I4 I8
I21	V1-V3	Anticipation	0.5	Composed by I5 I7
I22	V1-V3	Swap	0.33	Composed by I5 I7
I23	V1-V3	Swap	0.33	Composed by I5 I8
I24	V1-V3	Anticipation	0.17	Composed by I5 I9
I25	V1-V3	Anticipation	0.6	Composed by I6 I7
I26	V1-V3	Swap	0.4	Composed by I6 I7
I27	V1-V3	Postponement	0.2	Composed by I6 I8
I28	V1-V3	Swap	0.4	Composed by I6 I8
I29	V1-V3	Anticipation	0.2	Composed by I6 I9

Table 5.9: *Interpretations automatically identified with AVA for Tomcat case study. Method names have been replaced with ids according to Table 5.6. To save space we do not report the alignment results for composite interpretations.*

Id	Violation	Interpretation	Score	Extra information
I7	V3	Delete	1	Deleted event: T6. Expected sequence: T6 T8 T9. Observed: - T8 T9 .
I6	V1	Replace	1	Events T2 T2 T3 replaced with events: T6 T7 T2. Expected sequence: T0 T1 T2 T2 T3 T4 T5. Observed: T0 T1 T6 T7 T2 T4 T5.
I2	V1	Insert	0.75	Expected sequence: T0 T1 - - - T3 T4 T5. Observed: T0 T1 T6 T7 T2 - T4 T5.
I25	V1-V3	Anticipation	0.6	Anticipated event: T6. Noisy events: T7 T2 (inserted). Composed by I6 I7
I4	V1	Delete	0.56	Deleted events: T2 T3. Noisy events: T6 T7 (inserted). Expected sequence: T0 T1 - - T2 T2 T3 T4 T5. Observed: T0 T1 T6 T7 T2 - - T4 T5.
I18	V1-V3	Swap	0.4	Composed by I6 I7
I8	V3	Insert	0.33	Noisy events; T6 (deleted). Expected sequence: T6 T8 T9. Observed: - T8 T9.
I9	V3	Replace	0.33	Events T6 T8 replaced with events T8 T9. Noisy events: T9 (deleted). Expected sequence: T6 T8 T9. Observed: T8 T9 - .
I27	V1-V3	Postponement	0.2	Composed by I6 I8

Table 5.10: *AVA interpretations for the Tomcat case study. Method names have been replaced with ids according to Table 5.6.*

Chapter 6

Empirical Validation

This Chapter presents the results we obtained by applying the techniques described in previous chapters to diagnose real faults affecting medium and large size programs developed by both industries and open source foundations.

The analysis framework described in this PhD Thesis has been implemented by two Java tools: BCT and KLFA. BCT implements the functionalities for the diagnosis of faults through instrumented monitors, while KLFA implements the functionalities for the diagnosis of faults through the analysis of log files. We implemented two separate tools because they present distinct inputs and requirements: BCT works on Java applications and requires that developers instrument the application code, while KLFA can monitor systems implemented using different languages but requires that the monitored system generates log files. We implemented the techniques for the inference of models and diagnosis of faults, e.g. AVA, as standalone libraries, which are shared by the two tools. Section 6.1 describes the two tools in details.

Sections 6.2 to 6.4 present the results of the empirical evaluation. Section 6.2 presents the evaluation of the effectiveness of the strategies to filter false positives and correlate anomalies. Section 6.3 highlights the effectiveness of data flow models for the identification of anomalous behaviors, and compares the anomaly detection capabilities of models inferred with different levels of granularity. Section 6.4 presents the evaluation of the quality of the interpretations provided by AVA.

6.1 Tools

We implemented the conceptual framework described in previous chapters by two analysis tools implemented in JAVA: BCT [89] and KLFA [88]. Table 6.1 shows the functionalities implemented by each tool. The two tools extract behavioral information from distinct sources, BCT uses instrumented monitors while KLFA uses log files. For this reason each tool implements the functionalities necessary to diagnose faults using the specific source of behavioral information used. Following paragraphs provide an overview of the two tools.

	Feature	BCT	KLFA
Data Identification	Log file data extraction	No	Yes
	Extraction thrugh monitors	Yes	No
Models Types	Data properties	Yes	No
	Event flow models	Yes	Yes
	Data flow models	No	Yes
Models Granularity	Application	No	Yes
	Action	No	Yes
	Component	No	Yes
	Interface Method	Yes	No
Anomaly Detection	Coarse grained	Yes	Yes
	Fine grained	Yes	Yes
Fault Diagnosis	False positives filtering	Yes	No
	Anomaly graphs	Yes	No
	Automated Interpretations (AVA)	Yes	Yes

Table 6.1: *Features implemented by BCT and KLFA*

6.1.1 BCT

BCT is a tool for the dignosis of faults in Java systems. BCT has been implemented as a library and as an Eclipse plugin. The library implements the monitoring, inference and fault diagnosis functionalities. The Eclipse plug-in provides the graphical interfaces to drive the execution of BCT more easily. BCT diagnoses faults by performing the three activities described in this PhD Thesis: it captures the legal behavior of components

by monitoring successful test executions, it identifies misbehaviors during failing runs, and finally it diagnoses faults by analyzing the misbehaviors. Figure 6.1 shows the activities supported by BCT.

Capture Legal Behavior

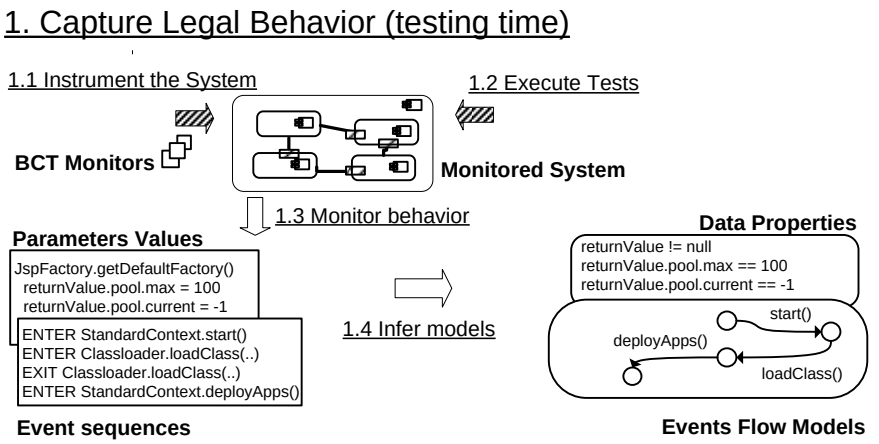
In the first stage developers instrument the target application with monitors that record information about the behavior of its components. Developers must identify the components they intend to monitor by specifying which classes are part of each component. They perform this activity by defining for each component a set of regular expressions that match the name of the classes which are part of a component. BCT performs the instrumentation with either Aspectwerkz [19], which is an aspect-oriented framework [77], or the TPTP probe technology, which is a monitoring utility provided by the TPTP testing platform [62]. Both Aspectwerkz and TPTP probes can extract runtime data from Java systems without requiring the availability of the source code, and can capture the method invocations and the associated attribute values. BCT captures only inter-component method invocations, and ignore the details of the internal computations.

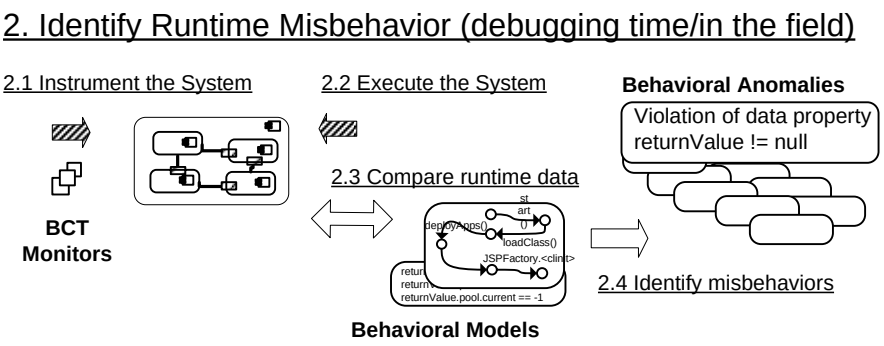
During testing BCT automatically records information about components behavior. After tests have been executed, developers can infer behavioral models. The inference is completely automated by BCT. BCT infers two kinds of models: *data properties*, i.e. properties of the parameters exchanged in method invocations, and *interface method models*, i.e. finite state automata that generalize the sequences of methods called by a component when a certain method of its interface is invoked (see Section 3.3). BCT provides graphical editors for the visualization and editing of the inferred models.

Identify Runtime Misbehavior

The inferred models are used to identify the behavioral anomalies that caused a failure. To identify anomalies, developers instrument the application with monitors that at runtime compare components behavior with the inferred models as described in Section 1.9.

The identification of anomalies can take place either when the application is running in the field or during the replication of failing runs in the development environment. In order to identify anomalies directly in the





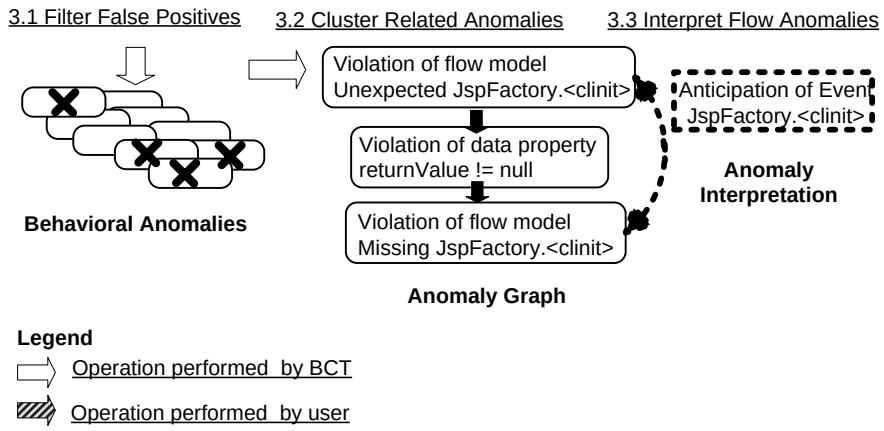


Figure 6.1: *Fault Diagnosis with BCT.*

field developers instrument the software before its deployment. The instrumented application automatically identifies behavioral anomalies while the software is used by the final user, thus permitting developers to diagnose the causes of eventual failures by retrieving the set of anomalies detected in the field. In case the failing application was not instrumented with BCT, developers can identify anomalies during debugging. In this case they need to instrument the application and then reproduce the failure in the development environment. The instrumented application will generate the anomalies that can be analyzed to diagnose the fault.

In order to identify data anomalies, BCT extracts for each method call the values of the parameters referred in the data models, checks the models and identifies the models that do not hold. When one or more parameters violate a model BCT records the parameter identifier, the model violated and the current application stack trace as described in Section 4.1.

In order to identify events flow anomalies BCT proceeds as follows. BCT monitors the interface methods of the target software components. When a monitored method, m, is invoked BCT loads the corresponding finite state automata, a_m, i.e. the interface method model that describes the sequence of methods that can be legally invoked when m is executed. BCT keeps in memory both the automaton and its current state, which initially is the initial state of the automaton, by pushing them on a stack structure. This operation permits BCT to monitor nested method invocations and still keep in memory both the automaton of the invoking method and its current state. The top of the stack contains the automata associated to the method currently in execution. If a monitored method n is invoked during the execution of method m, BCT checks if the current state of the automata a_m accepts the signature of n, i.e. BCT checks if the current state of a_m has an outgoing transition, t_n, labeled as the signature of n. If method n is accepted by a_m BCT updates the current state of a_m: the new current state is the state that receives transition t_n. Then BCT loads the automaton associated to method n and pushes it on the stack. The same operations are performed also for all the methods invoked by n and so on. Once the execution of method n terminates BCT checks if the current state of the automata a_n is final. If it is final BCT pops the automata out of the stack and proceeds with the checking of method m. For each monitored method, M, BCT also keeps a list, L_M, that contains the sequence of monitored methods

it invokes[1]. The list is kept in memory till method M terminates.

BCT detects an anomaly when a method invocation is not accepted by the current automaton, or when the current method execution terminates in a state which is not final. After detecting an anomaly BCT records the current stack trace and the following information about the anomaly: the signature of the monitored method, the state of the automaton when the anomaly is detected, and the signature of the method not accepted or the indication that the execution has finished in a state which is not final. After BCT has detected that the invocation of method t within method m is anomalous, it does not further check if the methods invoked within m after t accepted by the automaton a_m. BCT just keeps trace of them. Following, or concurrent, executions of method m are instead checked. Once the anomalous method execution m finishes, BCT records the list of methods invoked during the anomalous execution of m. This information is used during the diagnosis phase in order to apply *fine grained detection of flow anomalies*.

Diagnose Faults

The anomalies detected by BCT are analyzed by developers to diagnose the fault. Since anomalies are collected during legal and failing runs, developers must manually specify the set of failing executions they want to analyze. BCT provides a list with monitored executions that present some anomalies and developers chose the one they want to analyze, tipically a failing one. For failures due to unmanaged `RuntimeExceptions`, or for JUnit test cases, BCT is able to automatically detect that the execution failed. In this cases BCT highlights the ids of the executions that failed. In case failures are not automatically identified developers need to indicate which of the monitored runs failed, this activity is required in order to effectively apply the false positives filtering activity.

Automated diagnosis of failure causes consists of two tasks: filter false positives and build the anomaly graph. BCT filters false positives as described in Section 5.1. After having filtered the false positives, BCT builds the anomaly graph and presents it to developers. Developers inspect the anomaly graph by means of the BCT GUI. By default BCT identifies flow

[1]Since we monitor component interfaces the invocation can be performed not directly by M but by other methods of the same component invoked by it.

anomalies through coarse grained detection, i.e. it identifies the first anomaly that affect a model. Empirical results indicates that coarse grained detection of flow anomalies is usually effective when monitoring Java systems. In case the coarse grained detection of flow anomalies does not provide useful results developers can refine the analysis by applying the fine grained detection of flow anomalies and then rebuild the anomaly graph. Fine grained detection of flow anomalies consists in identifying all the anoamlous event subsequences that do not match a model (see Section 4.2). It can be applied because the instrumented monitors record the information about the whole anomalous sequences observed at runtime for each event flow anomaly detected.

In order to speed up the diagnosis of faults that caused event flow anomalies developers can use BCT to automatically interpret the flow anomalies. In this case, BCT internally invokes AVA, which has been implemented as an external library, and then provides developers the interpretations generated by AVA.

6.1.2 KLFA

KLFA is a toolset for the diagnosis of faults through the analysis of log files. Figure 6.2 shows the activities implemented by KLFA to analyze log files and diagnose faults. KLFA implements the three phases of the analysis framework described in this PhD Thesis: it captures the legal application behavior from log files recorded during legal executions, it analyzes logs recorded during failing executions to identify the anomalous behaviors that caused the failure, and finally automatically interprets the identified anomalies. Following paragraphs describe KLFA in detail.

Capture Legal Behavior

In the first phase developers collect logs recorded during successful executions, and use KLFA to infer models that summarize the legal behavior of the system.

There exist several technologies that support generation of logs [12, 62, 76]. Our approach does not refer to any specific monitoring solution and can be applied to any system independently from the logging technology.

KLFA infers models that generalize the legal data and event flow of the system. In order to infer models, KLFA parses the logs to identify events

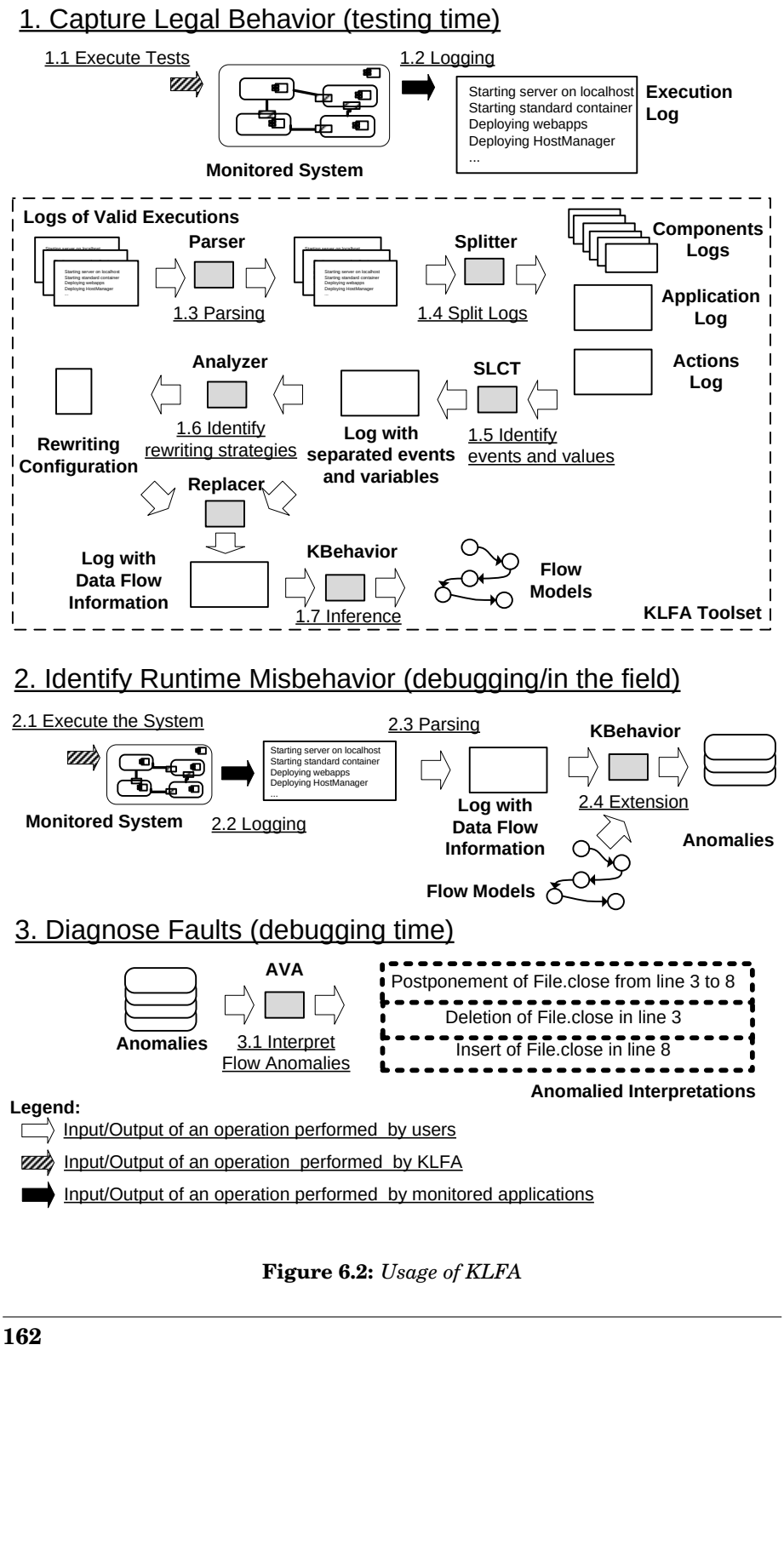


Figure 6.2: *Usage of KLFA*

and variables, identifies the proper rewriting strategy, and finally infers the models. The technique only requires the existence of a character that separates events (one event here is intended as the composition of the event name its parameter values). Developers can specify the granularity level to infer models: application, component, or action. In order to infer component level models, each log message must indicate the name of the component that generated the event. To infer action level models, developers must indicate the lines in which different actions start or the regular expression that matches the start of an action in the log.

In case of component level analysis, KLFA splits the log grouping together messages generated by a same component. KLFA then runs SLCT in order to extract events and data values as indicated in Section 3.2.1. KLFA then analyzes the extracted data to identify which are the transformation rules that better capture values flow, and then processes the data to generate a sequence of events with concrete values replaced by abstract symbols. In order to infer models of the system KLFA runs KBehvaior over the processed sequence. In case developers have chosen to infer models with component granularity, KLFA infers an automaton for each component of the system by running KBhavior multiple times. In each run KBehavior analyzes the subsequences of events that belong to a same component. In case of action granularity, KLFA infers an automaton for each monitored action. In order to infer the automaton that correspond to a particular *user action* KLFA runs KBehavior on the sequences of events that correspond to that action. Finally in order to infer the automaton that models the whole behavior of the applictaion KLFA runs KBehavior over the whole sequences of events reported in the logs.

Identify runtime Misbehavior

In order to diagnose faults, developers collect the logs recorded during the failing execution and use KLFA to identify anomalies and interpret them. KLFA automatically preprocess the faulty logs using the same configuration used to infer the models. Anomaly detection is performed according to *fine grained anomaly detection* by using KBheviour to identify anomalous subsequences of events.

Diagnose Faults

KLFA automatically runs AVA to interpret anomalous sequences. The results produced by AVA are then reported to developers to facilitate the diagnosis of the faults that caused the failure.

6.2 Effectiveness of anomalies correlation

6.2.1 Goal

The goal of the experiment presented in this Section is to evaluate the effectiveness of the combined false positives filtering and anomaly graphs detection strategies to help developers in the diagnosis of faults.

In particular we aim at answering the following research questions:

- Does the filtering strategy effectively remove anomalies that are unrelated with faults in the application?
- Does the filtering strategy remove anomalies that are related with faults in the application?
- Do the anomaly graphs cluster related anomalies thus helping the diagnosis of the fault?
- Does inspecting connected components in an anomaly graph from the biggest to the smallest is an effective strategy?
- Are the anomaly graphs effective to locate faults?

6.2.2 Subjects

To empirically answer to the previous research questions, we used BCT to diagnose faults affecting different versions of three systems: NanoXML [109], JXML2SQL [5], Eclipse [40] and Tomcat [118]. NanoXML is a XML parser of about 8 thousand lines of code. Jxml2sql is a simple Java application of about 3 hundred lines of code which uses NanoXml to convert XML documents in other formats. Eclipse is a development platform extendible with plug-ins provided by third parties (OTS components). Eclipse consists of about 17 million lines of code. Tomcat is and application server of about 3 hundred thousand lines of code.

We considered two kinds of faults: regression faults and field faults. Regression faults are consequences of components updates or replacements identified either during testing or after the deploy of the system in the field. Field faults instead represent functional faults not detected during the testing process which are experienced in the field by the final users.

The regression faults were introduced in updates of NanoXML, Tomcat and Eclipse. In the case of NanoXML, we analyzed faults injected by third

parties [47]. In the case of Jxml2sql we analyzed failures caused by the faults injected in NanoXML. In the cases of Tomcat and Eclipse, we analyzed real problems experienced with the released versions of the software that caused failures in the field.

Case study	Description	Fault Source	Fault Type
X1	Jxml2sql, NanoXML v4 to v5, CR_HD_1	Injected	Regression
X2	Jxml2sql, NanoXML v4 to v5, NV_HD_1	Injected	Regression
X3	Jxml2sql, NanoXML v4 to v5, SR_HD_1	Injected	Regression
N1	NanoXML v5, all_f	Injected	Regression
N2	NanoXML v4 to v5, SR_HD_1	Injected	Regression
N3	NanoXML v4 to v5, XER_HD_1	Injected	Regression
N4	NanoXML v4 to v5, CR_HD_3	Injected	Regression
N5	NanoXML v4 to v5, NV_HD_1	Injected	Regression
N6	NanoXML v4 to v5, all_f	Injected	Regression
T1	Tomcat 6.0.4	Real	Field
T2	Tomcat 5.5.12 to 5.5.13	Real	Regression
E1	Eclipse, WTP, EMF, GEF, JEM	Real	Regression
E2	Eclipse, GMF, EMF, OCL	Real	Regression
E3	Eclipse TPTP ProbeKit (chmod)	Real	Field
E4	Eclipse TPTP ProbeKit (EMT64)	Real	Field
E5	EMF 2.2.1, WTP	Real	Field

Table 6.2: *Case studies summary. Each case is identified by an id. For each case we indicate the versions of the software components monitored, if the monitored faults are injected by a third party or are real and if they are regressions or field faults.*

Table 6.2 shows the details of the case studies considered for the experiment. The table indicates the versions of the applications, considered components and the type of fault. The identifier of the Jxml2sql and NanoXML case studies are the names assigned to the different versions of the application in the SIR repository [47]. Two case studies are based on the Tomcat application server. The first case study with Tomcat (*T1*) indicates a fault experienced with Tomcat version 6.0.4 [13] (this case corresponds to the ex-

ample we described in Section 2.1). The second case study with Tomcat (*T2*) is the case of a regression fault from version 5.5.12 to version 5.5.13 [14]. Five case studies are based on the Eclipse IDE. The first Eclipse case study (*E1*) is an update from Eclipse 3.2.1 with the plug-in EMF version 2.2.1 to Eclipse 3.2.2 with EMF 2.2.2 and the addition of the plug-ins WTP 1.5.1, GEF 3.2.1 and JEM 1.2.1 (we discovered this undocumented issue). The second Eclipse case study (*E2*) is an update of GMF version 2.0.0M5 and EMF version 2.3.0M5 to GMF 2.0.0M6 and EMF 2.3.0M6 within Eclipse 3.3 with OCL version 1.1.0 [42]. The third Eclipse case study (*E3*) indicates a fault experienced with the Eclipse TPTP plug-in version 4.4.1 [43]. The fourth (*E4*) is the case of a fault experienced with the Eclipse TPTP plug-in version 4.3 [41]. Finally the last case (*E5*) is the case of a fault experienced with Eclipse 3.2.1 with plug-ins EMF 2.2.1 and WTP 1.5.1 (we discovered this undocumented issue).

6.2.3 Data Collection

In order to execute the empirical study we followed the three phases of the technique for every case study: capture legal behavior, identify misbehaviors, diagnose faults.

For all the case studies, we collected data about the legal behaviors by monitoring executions during testing. For the two Tomcat and Eclipse TPTP cases (T1, T2, E3, E4 respectively), we designed test cases that exercise the Tomcat web application manager and the TPTP static instrumentation functionality, using the category partition method [98]. For the other case studies, we run the test cases (unit and system test cases when available) provided together with the applications. We inferred models at the granularity of interface methods. For NanoXML, which is a small sized project we threated every class of the system as a single component, for Jxml2sql we monitored the whole NanoXML as a component, for the Eclipse case studies we monitored the plug-ins indicated in Table 6.2, while for Tomcat we monitored the principal components of the system (indicated in the Tomcat documentation) and the libraries that they use. We treated the monitored components as third party OTS components, thus ignoring both the source code even when available, and the information about the analyzed components themselves.

For the analysis of regression faults we inferred the models by monitor-

ing test cases executed on the fault free version of the software. We used the inferred models to identify anomalies when tests are executed on the faulty version of the system. We then used BCT to analyze the anomalies generated in the failing tests.

For the analysis of field failures we monitored the same version of the system to both capture the legal behavior and to identify misbehaviors. We derived models from data recorded during valid tests execution and then we used the models to identify misbheviors in failing executions not observed at testing time.

In order to enable the filtering of false positives we reused the inferred models to monitor the passing test cases and collect false misbehaviors. The monitoring of passing test cases permits to identify anomalies generated by models that overfit the training data: a data model, for example, could include the process id of the test which can vary from one execution to another. In case of regression faults the monitoring of the passing test cases permits to identify anomalies caused by new legal behaviors introduced by the changes in the software.

The anomalies collected during failing executions are then analyzed by BCT which filters and aggregates them to produce the results.

6.2.4 Results and Analysis

To correctly classify anomalies and clusters (including the ones automatically filtered by BCT) as either false/true positives/negatives, we manually inspected the applications to determine the presence of any relation between anomalies and the faulty behaviors. Following paragraphs describe the results obtained to respond to the different research questions.

Does the filtering strategy effectively remove anomalies that are unrelated with faults in the application?
Does the filtering strategy remove anomalies that are related with faults in the application?

In order to respond to the first two research questions we inspected the anomalies filtered out by the technique. We counted the number of *meaningful anomalies* filtered by the technique, i.e. the anomalies that are related to the fault but have been filtered out. We counted the number of *false anomalies* (the anomalies that correspond to legal behaviors) correctly fil-

tered by the technique and the number of the ones that were not filtered. Finally we calculated the two measures commonly used to evaluate data classifiers and machine learning techniques: *precision* and *recall*. We calculated *precision* as the ratio between the false positives that have been filtered and the total anomalies that have been filtered. We calculate *recall* as the ratio between false anomalies that have been filtered and the total number of false anomalies generated by the anomaly detection technique.

Case study	Meaningful Anomalies Filtered	False Anomalies Filtered	False Anomalies Not Filtered	Precision	Recall
X1	0	0	2	-	0.00
X2	0	0	0	-	-
X3	0	0	0	-	-
N1	0	4	0	1.00	1.00
N2	0	19	0	1.00	1.00
N3	0	18	1	1.00	0.95
N4	0	17	1	1.00	1.00
N5	0	21	1	1.00	0.95
N6	0	16	4	1.00	0.80
T1	0	3	1	1.00	0.75
T2	0	2	1	1.00	0.67
E1	0	0	0	-	-
E2	0	48	17	1.00	0.74
E3	0	0	0	-	-
E4	0	0	0	-	-
E5	0	0	2	-	0.00

Table 6.3: *Evaluation of precision of false positives filtering.*

Table 6.3 shows the results. The filtering strategy did not remove anomalies related to faults under investigation, thus not negatively affecting the final diagnosis of the faults. Furthermore the technique removed the majority of the false positives: In most cases the 90% of the false positives were filtered, with an average of 83% false positives correctly filtered out.

Do the anomaly graphs cluster related anomalies thus helping the diagnosis of the fault?
Does inspecting connected components in an anomaly graph from the biggest to the smallest is an effective strategy?

In order to respond to the third and fourth research questions, "Does the anomaly graph cluster together related anomalies thus helping the diagnosis of the fault?", we counted the number of connected components inspected by developers to diagnose the fault and compared this number with the overall number of connected components returned. The anomaly graph effectively clusters related anomalies when developers have to inspect only a small number of connected components in order to diagnose the fault.

In order to respond to the fourth research question, "Does the inspection of the biggest anomaly graphs first prevent the inspection of eventual false positives?", we counted the number of false positives among the anomalies inspected by developers to diagnose faults. Developers diagnose faults by traversing (and analizing) the anomalies reported in the anomalies graphs starting from the roots of the biggest connected component. If the fault cannot be diagnosed using the information present in the biggest connected component developers inspect the second connected component in the same manner. The process is repeated till the fault is diagnosed or all the anoamlies have been inspected. We counted the number of false positives (anomalies not related with the failure) and true positives (anomalies related with the failure) developers inspect before being able to diagnose the fault. The strategy used to present connected components to developers is effective if they have to inspect only a small number of false positives before diagnosing the fault.

Table 6.4 presents the results.

Column *Successfully diagnosed* indicates whether the inspection of the anomaly graph permitted to diagnose faults. Anomaly graphs suitably supported software engineers in diagnosing faults in most of the case studies: the connected componentscorrectly describe the events that caused the failures in 10 out of 13 case studies.

Column *Inspected CC* indicates how many connected components were inspected by engineers in order to diagnose fault. We report the total numer of Connected Components developers inspected to identify the fault (column *Total Inspected CC*), and the number of useless components that

Case study	Successfully diagnosed	Connected Components (CC)	Inspected CC		Inspected Anomalies	
			Total	Useless	TP*	FP*
X1	Yes	2	1	0	1	0
X2	No	0	0	0	0	0
X3	Yes	1	1	0	3	0
N1	Yes	3	3	0	10	0
N2	Yes	1	1	0	0	1
N3	No	1	1	1	0	1
N4	Yes	1	1	0	2	0
N5	No	1	1	1	0	1
N6	Yes	6	2	0	6	2
T1	Yes	2	1	0	3	0
T2	Yes	2	1	1	2	0
E1	Yes	2	2	0	2	1
E2	Yes	11	2	0	4	5
E3	Yes	1	1	0	3	1
E4	Yes	1	1	0	4	1
E5	No	2	2	2	0	6

* TP and FP indicate the number of true and false positives among the inspected anomalies respectively.

Table 6.4: *Efficacy of Anomaly Graphs for Fault Diagnosis.*

were inspected (column *Useless Inspected CC*). In 9 out of the 10 successful cases developers only need to inspect one or two connected components in order to diagnose the fault. Out of the 30 connected components of the anomaly graphs that permitted to correctly diagnose the faults only 15 were inspected, the 50%. This result indicates that the technique successfully aggregates anomalies related with the fault.

Column *Useless Inspeted CC* indicates how many useless connected components developers had to inspect. Out of the total 21 connected components inspected by software engineers to diagnose faults, only 6 were useless, less than 30%. If we consider only cases where a correct interpretation of the failure causes has been determined, only 1 useless connected component has been inspected (less than 6% of the totally inspected components). If we consider the unsuccessful cases, software engineers had to inspect at most 2 useless connected components before understanding that BCT was not able to explain the fault, which indicates that even when the technique does not diagnose the fault case the effort required to developers to inspect anomalies is extremely limited.

Column *Inspected Anomalies* indicates how many anomalies have been inspected by developers before diagnosing the fault. Column *TP* indicates how many true positives (anomalies caused by the fault) developers inspect before being able to identify the fault. Column *FP* indicates how many false positives (anomalies that do not depend on the fault) developers encounter before they diagnose the fault. The table shows that the number of inspected false positives is low. Of the 59 anomalies inspected, only 19 are false positives, less than 33%. In the cases in which BCT succesfully diagnoses faults the percentage of analyzed false positives is even lower, 10 false positives over a total of 51 anomalies inspected, less than 20%. Furthermore if we count the total number of false positives present in the anomaly graphs wher e BCT succeeds in identifying the fault, which are 21 (we obtained this value by summing the columns *False Anomalies Not Filtered* of Table 6.3), we see that the inspected false positives (10), are less than the 50% of the false positives in the anomaly graphs. These data stress the effectiveness of the strategy adopted to inspect the anomaly graphs, which clearly reduce the number of false positives developers have to inspect.

We would like to emphasize that without an efficient filter of false positives, software engineers would waste an enormous amount of time in inspecting many useless model violations. In fact, in our as well as in other

state-of-art techniques, false positives represent a large portion of the detected model violations. The result reported highlight both the efficacy of our filter for false positives and the enhancements for false positives filtering produced by our aggregation and prioritization technique.

Are the anomaly graphs effective to locate faults?

In order to respond to the fifth research question we compared the results obtained with BCT with the results obtained with Tarantula [72], to our knowledge one of the most effective fault localization techniques [4]. In order to perform the experiment we built a tool that implements the Tarantula scoring algorithm by the using coverage information provided by ConTest [63], a monitoring and testing library.

The main goal of the anomaly detection approach implemented in BCT is not the localization of faults but the identification of behavioral anomalies that help developers understanding the fault. However, every anomaly detected by BCT pinpoints the code instruction that generated the anomaly itself. For example BCT identifies that a certain instruction regard the invocation of a method returning a value that violates a data model, or the invocation of a component method not accepted by the event model. The code location provided by the anomaly identified by BCT is often the location from which developers start looking for faults.

In order to evaluate BCT fault localization capabilities we compared the effort required to locate faults with BCT and Tarantula by adopting the score method described in [72].

Tarantula ranks the lines of code executed in monitored executions according to their suspiciousness: the first lines presented by Tarantula are the ones which more likely contain the fault. For every line monitored in failing or valid execution Tarantula calculates a score that measure the likelyhood of the line to be faulty, then it sorts the different lines in descending order according to this score. For every case study in Table 6.2 we identified the lines covered in passing and failing executions using ConTest, and then calculated the Tarantula score according to the formula in Table 1.5. In order to measure the effort required to locate faults with Tarantula we counted the lines of code to inspect following the ranked list returned by Tarantula before reaching the faulty line. The number of lines to inspect corresponds to the position of the faulty statement in the ranked

list. In case the faulty statement has the same score than other statements in the ranked list, we consider the average number of statements that need to be inspected before getting into the faulty statement. For instance, if the faulty statement is ranked at the third position together with other two statements, the number of statements that need to be inspected in average before reaching the faulty statement is 4.

In the case of BCT, we have neither a ranked list nor an indication of the fault location at the level of statements internal to components. What we expect is that software engineers start their investigation from the statements that generated the true positives and then backwardly inspect the failing execution to find the fault location. Since software engineers may not immediately recognize that an anomaly is a false positive, for the purpose of the empirical investigation, we consider all the anomalies within the connected components as candidate starting points for the inspected, thus deriving empirical data that are not better than the real scenario. For the purpose of measuring the number of statements inspected by software engineering when looking for the fault location, we adopt the strategy also used in [72]. In practice, we first inspect all the statements that generated the anomalies that are present in the inspected connected components (corresponding to anomalies reported in columns *Inspected Anomalies TP* and *FP* in Table 6.4). The inspection order is the following. Connected components are inspected from the largest to the smallest. Anomalies within a component are inspected starting from the root nodes, and then continuing with the nodes linked by the edges with the lowest weight, until all the nodes in a connected component has been inspected. If the fault location is not found, we consider the just inspected code locations in the same order, and we move one statement backward according to the Program Dependency Graph (PDG), as suggested in [72] and [34]. If the fault location has not been found, we repeat the process of moving backwardly one statement according to the PDG until we found the fault location. When multiple equivalent choices are possible, e.g., multiple root nodes or multiple edges with the same weight exist, we report the average number of inspected statements, assuming the same probability for each choice.

In both cases, the distance between the output produced by BCT and Tarantula from the fault location is measured as the number of statements that are inspected before discovering the fault location, according to the respective strategies. When more than 50 code locations are inspected we

give up the inspection process, and we classify the output of the technique as not precise enough to facilitate the localization of the fault, according to the considered inspection strategy.

Note that we know the location of the fault because for all the case studies the fix is available, with the exception of the two issues that we discovered in Eclipse. In these two cases, we did not develop the fix. However, we ran multiple debugging sessions and we determined with good confidence the fault location.

Case study	BCT	Tarantula
X1	32	>50
X2	-	>50
X3	1	26.5
N1	>50	>50
N2	-	>50
N3	-	12.5
N4	1	47
N5	12	>50
N6	>50	>50
T1	>50	>50
T2	>50	>50
E1	>50	>50
E2	>50	>50
E3	1	36
E4	1	19.5
E5	>50	>50

Table 6.5: *Number of statements inspected with BCT and Tarantula to reach the fault location.*

Table 6.5 reports the data we obtained in the different case studies.

BCT performed better than Tarantula, according to our empirical results. This result is not surprising because the available test cases are quite limited in number compared to the size and number of functionalities implemented by the systems that we analyzed. Tarantula suffered the limited sampling of the code. On the contrary, BCT concentrates on

the differences between the behaviors shown when the same test suites are re-executed to discover the failure causes. The homogeneity guaranteed by the re-execution of the same test suites is enough to produce good results, even if the entire systems are not well sampled. Moreover, the case studies where BCT provided useful results include faults that impact on the integration between the components, which is an ideal context for BCT. Tarantula can work better with unit faults, which cannot be addressed by BCT if they do not impact on the interfaces of the components. In fact, for 2 out of the 5 case studies that are not successfully analyzed by BCT due to the absence of useful anomalies at the components' interfaces, Tarantula located the fault with a distance smaller than 50.

Note that the distance between the output produced by BCT and the fault location vary significantly from case to case. In particular, in several cases the distance is 1, while in several other cases the distance is greater than 50. This happens because BCT can only suggest code locations that are at the boundaries of the components, i.e., they are statements with calls to other components. Thus the distance depends on the PDG distance from the fault location to the statement with the call pinpointed by BCT, which strongly depends on the nature of the components. For example, this distance tends to be large when the components are complex, see the case studies based on Tomcat and Eclipse, while it is often small when the components are rather simple, see the case studies based on NanoXML.

Finally, note that when techniques based on code coverage fail in indicating a statement that is close to the fault location, software engineers have to act independently from the technique. On the contrary, the anomaly detection strategy presented in this PhD Thesis is useful to identify the failure causes independently from the distance between the anomalies and the fault location. Thus, software engineers can "jump" between different code locations according to their understanding of the failure causes and their understanding of the application, independently from the PDG, but according to their intuition. For instance, if a failure is caused by a non-initialized variable, the tester can jump directly to the function that initializes the components, independently from the PDG. This aspect can relevantly improve the time required for fault localization and is not taken into consideration in the reported empirical data.

6.2.5 Threats to Validity

Internal validity

We evaluated the quality of the results reported by BCT by manually inspecting the violations reported in the anomaly graphs in order to determine if they are related to the fault. The results of the empirical study could be biased in case of incorrect classification of the anomalies. To reduce this risk we threated all the doubtful cases in this way: in the case of the evaluation of the filtering strategy we consider the filtered anomaly correlated with the fault, in the case of the evaluation of the effectiveness of the anomaly graphs we consider the anomaly not related to the fault.

Faults in the implementation of BCT coud bias the results too. In order to reduce this risk we extensively tested the tool and fixed all the faults we found.

External validity

All the case studies are related to failures experienced with popular systems during regression testing or during field executions. In all the cases, we could count on a thorough set of executions to build our dynamic models. The technique may be less effective when dealing with systems that have not been thoroughly executed. We believe that even if the technique would work only in presence of a thorough set of executions, like in the case of regression testing and field execution, it would be of great value for software engineers.

We report the results of a relatively small set of empirical studies (14 cases in total). The results are good, but may not generalize. We executed a relatively small set of studies because of the costs of their set up. To be able to generate dynamic models, produce the refined anomaly graph, and examine the results to understand if the graphs provide enough information to diagnose the fault or are false positives, we must generate a test suite (if not already available), deploy the systems, execute the test cases and reproduce the failure. The core parts of the studies are not expensive: the technique has a low overhead with respect to the whole process, and it reduces the cost of locating faults with respect to classic inspection of failures, but the empirical validation setup can be extremely time consuming. We have chosen a set of real failures reported for popular systems to reduce the bias that may be induced by a relatively small number of empirical studies. We are aware that the quantitative results may be imprecise,

but the trends are valid.

We conducted the empirical studies on well known systems that span from few hundreds to million lines of code, and monitored faults that affected a few provided functionalities. The quality of the results depends on the size of the monitored components in fact in case of huge components the method in which the anomaly has been identified could be far from the faulty interaction. A consequence of this is that diagnosis could be more difficult in case of huge components. We diagnosed faults affecting systems of different sizes and monitored components composed from few to hundred classes. Results show that both in case of small (like in the NanoXml case studies) or huge components (like in the Eclipse case studies) the technique provides useful results.

6.3 Effectiveness of data flow models and models granularity

6.3.1 Goals

The goal of the experiment presented in this Section is to evaluate the effectiveness of data flow models and analysis granularity on the identification of behavioral anomalies.

In particular we aim at answering the following research questions:

- Are data-flow models useful to diagnose faults?
- Is fine grained analysis more effective than coarse grained analysis to identify useful anomalies?
- Does the granularity of the analysis have any impact on the quality of the results?
- Is values flow information relevant for the identification of useful anomalies?

6.3.2 Subjects

Data flow models can be inferred either from data collected from log files or from instrumented monitors. For this experiment we focused on data collected from logs. We used KLFA to analyze the subject programs. We conducted the analysis by inferring models of the legal behavior of a set of software systems and then used the models to identify the misbehaviors that lead to a fault.

Table 6.6 summarizes the considered case studies: CASS, a commercial air traffic control simulator developed by Artisys [16]; Glassfish, which is a J2EE application server (2.000.000 lines of code) [50]; an aircraft simulator produced by Israely Aerospace Industries (IAI) [67]; and Tomcat, which is a JSP/Servlet server (300.000 lines of code) [118]. We evaluated our technique against a set of known faults affecting these systems: an algorithmic fault manually injected in CASS, two Glassfish configuration problems [52, 53], two faults affecting Glassfish and Tomcat [51, 13], and a configuration problem affecting Tomcat [119]. The fault injected in the CASS system represents a typical example of an algorithmic fault in such application, according to indications of Artisys software engineers. The

	Case study	Failure type	Log size
A1	Air traffic control simulator. Log format is syslog.	Wrong system behavior [110].	1Mb
G1	GlassFish (v. 2-GA). Log format is uniform.	Petstore not deployed [52].	1Mb
G2	GlassFish (v. 2-GA). Log format is uniform.	Petstore not deployed [53].	84Mb
G3	GlassFish (v. 3-b01). Logs format is java simple log.	Server hangs [51].	47Mb
I1	Flight simulator. Log format is csv.	System does not perform required realtime tasks on time [111].	66Mb
T1	Tomcat (v. 6.0.4). Log format is java simple log.	Web application not started [13].	17Mb
T2	Tomcat (v. 6.0.14). Log format is java simple log.	Tomcat is not starting because the default port is already in use [119].	1Mb

Table 6.6: *Case studies considered to evaluate data flow models and analysis granularity efficacy.*

other faults were selected among the ones reported in bug reports submitted by the final users of the applications, by choosing the faults originally debugged by developers through log analysis, which thus represents a realistic application of our technology.

6.3.3 Data Collection

In order to collect behavioral data we followed the first two phases of KLFA: capture legal behavior and identify runtime misbehavior.

In order to *capture the legal application behavior* we first derived test cases for the target systems by using the category partition method [98] and we collected log files from successful executions. We collected log files using the highest verbosity because it permits to collect more information to diagnose the fault, with the exception of T2 and G2, where the default verbosity has been used because in these cases we analyzed the logs collected by the final users of the systems, which were recorded with default verbosity. The collected logs were then used to infer models of the legal behavior of the system.

For each case study we derived models according to the three granularity levels supported by the technique: component, action and application. In order to infer models with component level granularity we used the name of the reporting component of each log message as the indicator of the component. In order to infer models according to the action granularity we identifies the start of the actions according to systems specifications for all the case studies with the exception of A1, where the user action granularity level is not applicable. For the Tomcat and Glassfish case studies we derived models for the following actions: server start, deploy of web applications, access to server manager interface, restart of a web application, stop of a web application, server shutdown. In order to derive application models we considered the log content discarding information about the identify of the logging component. Models have been automatically inferred by KLFA, the rewriting strategies were automatically selected by the system as described in Section 3.3.3.

In order to identify the runtime misbehaviors that caused the failure, we collected the logs recorded during failing executions and we analyzed them with KLFA. For two of the case staudies, G2, and T2, we used the logs recorded by the real users that experienced the failure. For the other

cases we reproduced the failures and we collected the logs generated.

6.3.4 Empirical Results and Analysis

To distinguish true from false positives, we manually inspected both the applications and the anomalies reported by KLFA to testers. Among true positives we distinguished *fault detectors*, anomalies that point the developer to the problem, from *contextual anomalies*, anomalies which are related with the failure but just provide contextual information, for example a `NullPointerException` that caused the failure, but that does not help the diagnosis of the fault.

Are data-flow models useful to diagnose faults?
Is fine grained analisis more effective than coarse grained analysis to identify useful anomalies?

In order to respond to the first two research questions we measured and compared the precision of the results for both fine grained and coarse grained anomaly detection. Furthermore we calculated the percentage of events selected for inspection by KLFA with respect to the entire set of anomalous events that have been detected. This measure is an indicator of the effort saved with this analysis with respect to manual analysis of the logs, which usually consist of sequentially analyzing all the anomalies until an explanation to the failure is found.

Tables 6.7 and 6.8 show the results obtained with coarse and fine anomaly detection respectively.

The results reported in the two tables indicate that when models are inferred with component level granularity the precision of results obtained with *coarse grained* and *fine grained* anomaly detection is similar. *Fine grained* anomaly detection performs slightly better in 5 cases out of 6, while in one case, T2, *coarse grained* anomaly detection generates less false positives. For T2 *coarse grained* anomaly detection provides better results because 6 false positives affect the execution of a same component and are thus identified by fine grained anomaly detection only. Since coarse grained anomaly detection identifies only the first anomaly, the other 5 false positives are discarded.

When models are instead inferred with action or application level granularity, *coarse grained* anomaly detection produces worse results results

Case study	Reported events	False Positives	True Positives	Precision	Fault Detectors
	Component Level Analysis				
A1	0.78%	4	1	0.20	1
G1	28.57%	3	5	0.63	1
G2	0.02%	14	7	0.33	7
G3	0.01%	15	1	0.06	1
I1	0.01%	1	1	0.50	0
T1	0.36%	3	3	0.05	0
T2	28.78%	0	5	1.00	3
	Action Level Analysis				
A1	-	-	-	-	-
G1	3.57%	0	1	1.00	1
G2	0.00%	2	0	0.00	0
G3	0.00%	3	0	0.00	0
I1	0.00%	1	0	0.00	0
T1	0.18%	3	5	0.00	0
T2	5.56%	0	1	1.00	1
	Application Level Analysis				
A1	0.63%	3	1	0.25	1
G1	3.57%	1	0	0.00	1
G2	0.00%	2	0	0.00	0
G3	0.00%	1	0	0.00	0
I1	0.18%	3	0	0.00	0
T1	0.18%	3	0	0.00	0
T2	5.56%	0	1	1.00	1

Table 6.7: *Results obtained with coarse grained anomaly detection using models with different granularity.*

Case study	Reported events	False Positives	True Positives	Precision	Fault detectors
	Component Level Analysis				
A1	1.88%	9	3	0.25	3
G1	28.57%	3	5	0.63	1
G2	0.03%	17	10	0.37	7
G3	0.04%	44	13	0.16	13
I1	<0.01%	3	2	1	
T1	1.07%	6	12	0.67	4
T2	50.00%	0	9	1.00	3
	Action Level Analysis				
A1	1-	-	-	-	-
G1	28.25%	0	5	1.00	1
G2	0.05%	37	8	0.18	4
G3	0.07%	79	10	0.11	0
I1	<0.01%	5	17	0.78	7
T1	1.25%	16	5	0.24	1
T2	11.11%	0	2	1.00	2
	Application Level Analysis				
A1	1.73%	5	7	0.64	2
G1	3.57%	1	0	0.00	0
G2	0.05%	34	7	0.17	4
G3	0.03%	39	7	0.15	4
I1	<0.01%	53	2	0.04	0
T1	2.79%	4	8	0.67	4
T2	11.11%	0	2	1.00	2

Table 6.8: *Results obtained with fine grained anomaly detection using models with different granularity.*

than fine grained anomaly detection: in almost all the cases precision for *coarse grained* anomaly detection is 0. Since application and action models are larger and more complex than component models they are more likely affected by false positives. The presence of false positives at the beginning of the analyzed sequence prevents the identification of the useful anomalies that follow. These results confirm our assumption that *fine grained* anomaly detection is necessary to mitigate the effect of false positives. Furthermore the results obtained with coarse grained anomaly detection applied to component models confirm the results in Section 6.2 which show that *coarse grained* anomaly detection is often sufficient to succesfully diagnose faults when component models are used.

Since *fine grained* anomaly detection generally performs better than *coarse grained* anomaly detection we focus the analysis of results obtained on the former type of analysis.

The percentage of events identified as suspicious by our technique gives an indication of the effort required to debug faults: the user has to manually inspect each set of suspicious events in order to identify the fault.

In all the case studies, our technique presented to testers a small fraction of suspicious sequences from the overall set of detected anomalies investigated (from 0.03% in the best case, to 28.57% in the worst case), and the suspicious events always included an explanation to the investigated failure. In G2 our technique identified an exception caused by a configuration problem; in G3 and T1, it identified faulty class initializations and faulty load events; in A1, it detected a wrong sequence of values that caused the system failure; finally in T2 it detected a message indicating that the port address is already in use. All these issues are related to unexpected combinations of events and attribute values. The high reduction in the number of events to be inspected results in an important save of effort and time by testers.

Data report a moderately high number of false positives. They are caused by incompleteness of the samples used for model inference and limited generalization in the inference process. For instance, if some events in log files always appear with same attribute values, SLCT can imprecisely partition event names from attributes. Similarly, a failure that executes a part of the system that has been never executed before would generate several false alarms.

Even if some false positives are often present, the technique still results

in a important advantage for testers, who need to analyze a small percentage of the faulty logs to diagnose failure causes.

We experimentally detected that testers can reduce the number of experienced false positives by restricting the analysis to the events generated in failing user actions. If the events that indicate beginning and termination of user actions are defined, this reduction can be always applied, independently from the granularity level selected for the analysis. It is worth to mention that this optimization can be applied only to reactive systems, where user actions can be defined. Table 6.9 summarizes the results when this reduction is applied. In the majority of the cases, precision improves substantially at the cost of losing some true positives. This is due to the restricted scope of the optimized analysis that may miss some relevant anomalies that are outside the scope of the failing user action.

Does the granularity of the analysis have any impact on the quality of the results?

In order to respond to the third research question we simply compared the precision of results obtained with the different granularities (application, component and user action levels).

The three granularity levels provided different results in the six case studies. In cases T1, G2, and G3, where we analyzed verbose logs, component component level analysis performs better than the others. Since component level analysis focuses on one component at time, the derived models are more compact and precise than the ones derived with the other two granularity levels. In case G1, the action level analysis works better than the others. In this case the number of events recorded in log files are limited, thus allowing the generation of precise models that describe the events generated in response of user actions.

In case A1, the application level analysis provided the best results. In this case, most of the false positives have been generated by a part of the system that has been never executed in the behavior capturing phase. The application granularity collapsed all the new events in a single extension of the model, which can easily and quickly be identified as a false alarm. Models generated with the component level granularity, instead, produced several false positives, because the failing execution traversed many functionalities never monitored before, thus resulting in a high number of model

Case study	Reported events	False Positives	True Positives	Precision
	Component Level Analysis			
A1	1.88%	9	3	0.25
G1	28.57%	3	5	0.63
G2	0.01%	2	6	0.75
G3	0.03%	29	13	0.31
I1*	<0.01%	0.1	0.2	0.67
T1	0.71%	4	8	0.67
T2	50.00%	0	9	1.00
	Action Level Analysis			
A1	-	-	-	-
G1	28.25%	0	5	1.00
G2	0.02%	9	6	0.35
G3	0.04%	43	7	0.14
I1*	<0.01%	0.3	1.7	0.85
T1	0.65%	5	6	0.55
T2	11.11%	0	2	1.00
	Application Level Analysis			
A1	1.73%	5	7	0.64
G1	3.57%	1	0	0.00
G2	0.01%	7	4	0.36
G3	0.02%	23	7	0.23
I1*	<0.01%	5.1	0.2	0.04
T1	1.90%	8	14	0.44
T2	11.11%	0	2	1.00

*In case I1 we have multiple failing user actions thus we considered the average number of reported false and true positives.

Table 6.9: *Results obtained by restricting the fine grained analysis to the failing user action.*

violations.

In T2 all the three results presented a precision equals to 1. This depends on the fact that the failure causes an anomalous behavior which is easily detected by AVA: Tomcat abruptly stops during the first stage of the startup thus AVA is able to detect that the anomalies in the logs are early terminations. startup and consists in the termination execution abruptly stops and as a consequence, anomalous terminations are detected for all the models considered.

The component level granularity performs better than the others on average and can be applied even when a rich set of events and attributes is logged. User action granularity scales worse than component level because many events and attributes can be observed when a user action is executed, but it is effective in detecting the causes of failures when they are spread between multiple components but concentrated on a single user action. Finally, application level granularity is the one which provides the best results when few events and attributes are logged per execution, but suffers of scalability problems when large systems or huge log files are considered.

Is values flow information relevant for the identification of useful anomalies?

In order to respond to the last research question we considered for every case study the analysis that performs better and count the number of true and false positives detected thanks to data flow information. An anomaly is detected thanks to data flow information if the anomalous event is expected by the model but the values do not correspond with the ones in the model.

Table 6.10 shows data about the number of true positives identified thanks to the analysis of data values. The number of true positives presented to testers thanks to the analysis of data flow relations is relevant and demonstrates the importance of working on both events and attribute values, which is one of the distinguishing characteristics of our solution.

Finally Table 6.11 shows information about the complexity of the models used for the diagnosis in order to highlight the capability of the technique to both handle and derive models of non-trivial size for real systems. For each case study we indicate the average number of transitions, states and distinct symbols in each inferred model.

Case study	Granularity	True Positives	True Positives violating Values Flow
A1	application	7	7 (100%)
G1	action	5	0 (0%)
G2	component	10	2 (20%)
G3	component	13	2 (15%)
I1	action	17	0 (0%)
T1	component	11	7 (64%)
T2	action	9	0 (0%)

Table 6.10: *Results about relevance of attribute values in the analysis.*

Case study	Granularity	Number of FSA	Average number of states	Average Number of Transitions	Average Number of Symbols
A1	application	1	338	373	117
G1	action	2	84	130	74
G2	component	67	23	56	19
G3	component	41	8	27	9
I1	action	1	47	69	21
T1	component	37	32	76	47
T2	component	7	7	9	4

Table 6.11: *Complexity of the models used for the analysis.*

6.3.5 Threats to validity

Internal validity

The results of the empirical study could be biased in case we incorrectly classify of the anomalies identified by KLFA. To reduce this risk we threated all the doubtful anomalies as false positives, and all the true positives that were not clearly fault detectors as contextual anomalies.

Faults in the implementation of KLFA coud bias the results too. In order to reduce this risk we extensively tested the tool and fixed all the faults we found.

External validity

In our experiments we evaluated the effectiveness of coarse and fine anomaly detection when data flow models are used. We executed experiments in which behavioral data is extracted from log files but we did not consider any case in which data is extracted from instrumented monitors. Three out of six experiments used logs recorded with the finest granularity. Since the finest granularity presents messages about behavior of single classes we expect that the results we obtained are extendible to cases in which data is gathered from instrumented monitors (instrumented monitors in fact capture parameters passed to method of single classes).

We run experiments involving four systems of different complexity and size. Even if we did not cover all the possible kinds of systems, for example we did not consider any desktop application, we think that the complexity of the inferred models, shown in Table 6.11, indicates that the technique can scale to large and complex systems. Furthermore, since we considered faults already diagnosed by application developers thorough log analysis, we are confident that the technique could be of practical usage in contexts in which log file analysis is already adopted to diagnose faults.

6.4 Evaluation of Flow Anomalies Interpretations

6.4.1 Goals

The case studies presented in this Section aim at demonstrating that the flow anomalies interpretations generated by AVA can describe real failure causes and that AVA can effectively discover them.

In particular, we want to respond to the following research questions:

- Does AVA discover interpretations that describe the differences between observed and expected behaviors, when these differences match our patterns?
- Do automated interpretations ease the failure analysis activity?

To this end, we designed two empirical investigations. The first investigation aims at responding to the first question, it focuses on a set of synthetic cases that have been automatically generated by our toolset. These cases cover the possible structure of the differences between expected and observed behaviors. We show that AVA effectively analyzes these cases. The second investigation aims at responding to the second question, it focuses on a set of third-party systems that are affected by known issues. We show that AVA effectively produces correct interpretations that can support and ease failure analysis activities.

6.4.2 Synthetic Subjects

To evaluate the capability of AVA to correctly interpret differences between legal and failing executions, we designed a set of synthetic cases that stress the set of possible interpretations defined in this paper. In particular, we obtained the set of cases to be investigated by automatically generating expected and observed sequences that differ according to the following dimensions: the type of the anomaly, the starting point of the anomaly, the length of the anomaly, and, only for composite anomalies, the number of events that separate the two basic anomalies that compose the composite anomalies.

The type of the anomaly can be any of the basic and composite interpretations with the exception of the *early termination* interpretation that can

be trivially identified by AVA. The starting point of the anomaly indicates the first event that occurs in the anomalous behavior and it can be at any of the following positions with respect to events in the observed sequence: 0 (intended as the first event of the sequence), 1, 2, 3, end of the sequence minus 3, and end of the sequence minus 2. The length of the anomaly, intended as the set of symbols in the anomalous behavior, can be 1, 2, 3, 7 and 12. The distance between two basic interpretations that are part of a composite interpretations can be 1, 2, 3, 10, or 15 events.

If we combine the parameters above, we obtain 36 cases for each basic pattern and 180 cases for each composite pattern. We analyzed each case with AVA with $k_1 = k_2 = 4$ ($k_1 and k_2$ are the constants used that determine the anomaly neighbourhood as described in Section 5.3).

6.4.3 Synthetic Subjects: Results and Analysis

In order to respond to the research question "Does AVA discover interpretations that describe the differences between observed and expected behaviors, when these differences match our patterns?" we analyzed the interpretations produced by AVA for the synthetic cases. For each case study we checked if the interpretation produced by AVA corresponded to the synthesized anomaly. Since certain anomalies can be correctly interpreted in multiple ways (for example an anticipation corresponds to both a deletion and an insert of events) we consider both the position of the *perfect interpretation*, i.e the one that perfectly describes the anomaly, and the position of the other valid but less complete interpretation.

Table 6.12 summarizes the results we obtained with the synthetic cases. We can notice that all basic interpretations are successfully recognized and presented to testers at the top of the ranking, with few exceptions regarding the delete interpretations. In particular, for 2 out of the 36 cases related to the delete interpretation, AVA classifies replacement as a better description of the differences than delete. This happens when the deleted events occur at the end of the sequence.

We can also notice that it seldom occurs that composite interpretations are reported at the first position of the ranking. This is inherently related to complexity of these interpretations. However, most of the composite interpretations are reported at the second position, and in all cases within the fifth. Moreover, in a relevant number of cases, composite interpretations

Anomaly Type	Cases	Perfect Interpretation Ranking			Related Basic Patterns Found Before
		1	<= 2	<= 5	
delete	36	34	36	36	-
finalState	36	36	36	36	-
insert	36	36	36	36	-
replacement	36	36	36	36	-
anticipation	180	0	158	180	87
postponement	180	0	111	180	73
swap	180	0	157	180	123

Table 6.12: *Results with ad-hoc cases*

are overcome by the basic interpretations that generate the composite interpretations (see last column in Table 6.12). In such cases, it is extremely simple for testers to recognize that the composite interpretation is the most interesting description of the failure cause.

In summary, synthetic cases show that AVA can effectively recognize the interpretations defined in Section 5.3.

6.4.4 Third-Party Subjects

To show that the interpretations defined in this paper can describe failure causes of real systems and AVA can discover them, we used the 6 case studies presented in Section 6.3.

The objective of the empirical validation is to show that AVA generates correct and useful interpretations of the observed failures and presents to testers a limited number of false positives that do not hinder effectiveness of the technique.

The empirical study consisted in applying AVA to the anomalies detected by KLFA: we run KLFA to derive models of the six case studies and to identify the anomalies, then AVA analyzed the anomalies detected and generates the interpretations. We used models derived with component level granularity, and, to limit the length of the analyzed behaviors, we use a configuration of AVA with $k_1 = k_2 = 2$.

6.4.5 Third-Party Subjects: Results and Analysis

In order to respond to the research question "Do automated interpretations ease the failure analysis activity?" we simulate developers activity by manually inspecting the interpretations provided by AVA in order to determine if they were correct and permitted to diagnose the problem faster than just the analysis of anomalies. We measured the quality of AVA results by looking at the interpretation provided by AVA and detecting, by comparing the corresponding log entries with the model, if they provide information which is useful to understand and diagnose the fault, i.e., they are true positives, or not, i.e., they are false positives. Since AVA sorts interpretations according to their likelyhood of being captured by a pattern, we begin the analysis from the interpretation with the highest score and we analyzed all the anomalies till we reached the one that permitted to diagnose the problem.

We have a true positive when the interpretation is correct, for example AVA detects a deletion of events and the log present missing events, and the interpreted anomalies are related with the fault. We defined two classes of relations: *fault detectors*, i.e., anomalies that permit to diagnose the fault, or *contextual anomalies*, i.e., anomalies that provide information useful to understand the problem and the context of the failure, but are not sufficient to diagnose the fault. We have a false positive when the interpretation of an anomaly is not correct, or when the anomaly itself is a false positives.

In order to estimate the effectiveness of the technique, we counted for each case study the number of false positives that developers had to inspect before reaching the *fault detector* anomaly. We also traced the position of the first true positive that is reported: we expect that developers stop the inspection of results if they have to inspect too many false positives before finding a true anomaly.

In order to determine if interpretations ease the diagnosis activity, we compared the results obtained with AVA with the results obtained just by applying the fine grained anomaly detection.

Fine grained anomaly detection does not sort results. In order to compare interpretations produced by AVA with the anomalies detected with fine grained anomaly detection, we compare AVA results with result obtained in two different situations: and the case in which developers inspect the anomalies in order of appearance stopping when a *fault detector*

anomaly is found and the case in which developers inspect all the anomalies reported. For the first case we inspected the anomalies in order of appearance and we traced the number of false positives, the position of the *fault detector*, and the position of the first true positive as we did for AVA. For the second case we compare the number of false positives caused by AVA with the number of false positives already reported in Section 6.3.4.

Case study	Position of fault detector		False Positives		False Positive Rate		Position of first true positive	
	Sorted KLFA	AVA	Sorted KLFA	AVA	Sorted KLFA	AVA	Sorted KLFA	AVA
A1	10	10	9	7	0.9	0.7	10	3
G1	6	5	3	2	0.5	0.4	4	1
G2	15	16	9	4	0.6	0.25	1	1
G3	10	5	9	4	0.90	0.80	10	5
T1	2	1	1	0	0.50	0.00	2	1
T2	2	1	1	0	0.50	0.00	2	1

Table 6.13: *Results obtained by applying AVA to third-party systems*

Table 6.13 presents the comparison between AVA interpretations (column *AVA*), and the ones obtained by inspecting the sorted anomalies (column *Sorted Anomalies*). The first part of Table 6.13, *Position of fault detector*, indicates the position of the anomaly that permits to diagnose the fault. In almost all the cases, with the exception of G2, the sorting of interpretations produced by AVA permits developers to diagnose faults faster, by analyzing less anomalies than just applying the *fine grained anomaly* detection strategy. Note that AVA provides a description of the failure cause at one of the first 5 positions of the ranking for 4 case studies out of 5 (in 2 cases is at the top of the ranking). Moreover, in all the cases testers have to analyze at most 4 false interpretations. These results confirm effectiveness of AVA when analyzing real failures.

The second part of Table 6.13, *False positives*, reports how many false positives developers have to inspect before diagnosing a fault. Results indicate that AVA interpretations reduce the number of false positives developers have to inspect. This result is underlined also by the *false positive rate*

reported in the third part of the table.The false positive rate obtained with the interpretation of anomalies is always lower than the false positive rate obtained without interpretations.

The last part of the table, *Position of the first true positive*, indicates how many false anomalies developers have to inspect before identifying a meaningful one. We consider this measure as an indicator of the practical applicability of the tool, if no true positives are detected within the first results developers are discouraged to continue inspecting the remaining anomalies. Furthermore the presence of anomalies that provide information about the failure, even if they do not permit to diagnose the fault, could ease the diagnosis of the fault when the *fault detector anomaly* is encountered.

Case	Anomalous Events	False Positives	False Positive Rate
A1	12	9	0.75
G1	8	5	0.36
G2	27	10	0.63
G3	57	13	0.84
T1	18	12	0.32
T2	13	1	0.00

Table 6.14: *Summary of fine grained anomaly detection obtained by applying KLFA to third-party systems.*

Table 6.14 summarizes the results obtained with fine grained analysis without AVA: the overall number of anomalies, the false positives found and the false positive rate. If we compare the false positives that developers have to inspect without AVA with the ones they have to inspect with AVA we detect that in all the cases AVA performs better.

Columns *FSA*, *States*, *Transitions*, and *Transitions Rate* in Table‘6.15 indicate the number of FSAs generated by KLFA, and the average number of states, transitions and transitions per state in the inferred FSAs (calculated by dividing the number of transitions by the number of the states in the automata), respectively. These data show that the AVA technique is able to work with multiple models of relevant size and complexity.

Case study	FSA	States	Transitions	Transitions Rate
G1	7	7	8	1.14
G2	16	56	442	7.75
G3	41	8	27	3.38
T1	6	32	76	2.38
T2	9	6	8	1.33

Table 6.15: *Size of the automata considered in the case studies.*

6.4.6 Threats to validity

Internal validity

The results of the empirical study could be biased in case we incorrectly classify the anomalies sorted by AVA. To reduce this risk we threated all the doubtful anomalies as false positives.

Faults in the implementation of AVA coud bias the results too. In order to reduce this risk we extensively tested the tool and fixed all the faults we found. Furthermore we manually inspected all the results reported in the Synthetic cases permitting us to verify the capability of the technique to correctly interpret all the kinds of anomalies it interprets.

External validity

We considered case studies involving four systems of different complexity and size.

We did not cover all the possible kinds of systems, for example we did not consider any desktop application, but we think that the results obtained can be generalized because the logging frameworks adopted by the monitored systems are often used also by desktop applications.

In our case studies we considered large systems that generated faily large logs. The complexity of the inferred models, shown in Table 6.15, indicates that the technique scale to complex systems generating large logs even if further experiments with larger logs could be executed to better evaluate the scalability of the approach.

Furthermore, since we considered faults already diagnosed through log analysis by the applications developers, we are confident that the technique could be of practical usage in contexts in which log file analysis is already adopted to diagnose faults.

Conclusions

Software faults have a relevant impact on today economy. Both the US National Institute of Standards and Technology (NIST) [95], and European Commission [35] stress the need for countermeasures to reduce the negative impact of software faults on today economy. Unfortunately despite the willings of politicians and software users the complete removal of software faults still remain almost impossible today. Even if extensive validation and verification activities could improve the quality of many software systems, the limitations of the existing verification and validation techniques, which often require source code, specifications, or present technological limitations, prevent them to be successfully applied on a wide range of systems, for example the many that integrate *Off the Shelf Components* released without source code or with incomplete specifications.

A possible solution for the reduction of the costs caused by software faults is the adoption of techniques that automate fault diagnosis. Fault diagnosis is one of the activities that impact more on software development and maintenance costs [128]. The automation of fault diagnosis activities could both reduce maintenance costs and permit the release of fixed software versions faster, thus reducing downtime. Different automated and semi-automated fault diagnosis solutions have been proposed by researchers but they have been scarcely adopted by developers because of their limitations. Certain techniques pinpoint faulty code fragments but do not provide any additional information about the faulty behavior thus not helping developers in the interpretation of the problem and in the identification of the fix. Other techniques instead describe the misbehaviors that characterize a faulty execution by identifying violations of models of the legal behavior, but these techniques usually produce a relevant number of false positives, thus significantly limiting their applicability in the practice.

This PhD Thesis describes an anomaly detection framework that automates the diagnosis of faults by identifying the software misbehaviors that lead to a failure, by automatically filtering false positives, by identifying relations among valid anomalies, and finally by providing easy to understand interpretations of the observed problems. The framework identifies anomalies by comparing actual data with models inferred from data recorded during successful executions. The anomalies identified during failing executions are automatically analyzed by the framework and presented to developers in compact representations that highlights relationships between them. Finally the framework generates interpretations of the detected anomalies in terms of human understandable patterns.

6.4.7 Contributions

This PhD Thesis advances the state of the art of automated fault diagnosis techniques by:

- Presenting a technique that captures the application data flow [37, 88]. The technique generates finite state automata that model the relationships between events observed at runtime and variable values associated to those events.
- Describing a technique to automatically filter false positives [89]. The technique automatically removes anomalies that likely are not caused by the fault.
- Presenting a technique for the generation of anomaly graphs [89], structures that explicitly show the cause effect relationships between multiple heterogeneous anomalies, and that identify clusters of related anomalies thus simplifying the diagnosis of faults in presence of multiple misbehaviors.
- Defining the AVA technique for the automated interpretation of anomalous event sequences [20]. AVA automatically interprets anomalous event sequences, by providing interpretations of the observed issues according to a predefined set of patterns which are easy to be understood by humans.
- Empirically validating the solution with real faults affecting medium and large size software systems and demonstrating the effectiveness

of both the false positives filtering strategy and the anomaly graph representation.

- Empirically validating the solution with open-source and industrial systems and demonstrating the effectiveness of both data flow models and fine grained detection of anomalies for the diagnosis of faults.
- Empirically validating the solution with open-source and industrial systems and highlighting the usefulness of automated anomalies interpretations to simplify the diagnosis of faults.

The framework has been adopted by the industrial partners of the SHADOWS project [112] to diagnose faults. SHADOWS is a EU funded project that defined a self-healing platform and a methodology for the development of self-healing systems [49], i.e. software systems capable of identifying and fixing functional, performance and concurrency faults.

We are currently working on the use of the techniques developed in this PhD Thesis as enabling work for further solutions in different domains. For instance we are integrating the fault diagnosis framework with multiple failure detection and healing techniques [56].

Bibliography

[1] A Developer's blog. Why EU Software Liability Bill is Not Going to Work, http://itscommonsensestupid.blogspot.com/2009/06/why-eu-software-liability-bill-is-not.html, visited in 2009.

[2] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. (TAICPART-MUTATION 2007)*, pages 89–98, Sept. 2007.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An observation-based model for fault localization. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 64–70. ACM, 2008.

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC '06: 12th Pacific Rim International Symposium on Dependable Computing.*, pages 39–46, Dec. 2006.

[5] Adam VanderHook. Jxml2sql. http://jxml2sql.sourceforge.net/, visited in 2009.

[6] H. Agrawal, R.A. De Millo, and E.H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[7] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE'95: Proceedings of the Sixth IEEE International Symposium on Software Reliability Engineering*, pages 143–151. IEEE, 1995.

[8] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI'90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256. ACM Press, 1990.

[9] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. ACM, 1997.

[10] J. H. Andrews and Y. Zhang. Broad-spectrum studies of log file analysis. In *ICSE'00: Proceedings of the 22nd international conference on Software engineering*, pages 105–114. ACM, 2000.

[11] Apache Software Foundation. Apache http server web site. https://httpd.apache.org, visited in 2009.

[12] Apache Software Foundation. Log4java. http://logging.apache.org/log4j/, visited in 2009.

[13] Apache Software Foundation. Tomcat bug 40820. https://issues.apache.org/bugzilla/show_bug.cgi?id=40820, visited in 2009.

[14] Apache Software Foundation. Tomcat bug 41939. https://issues.apache.org/bugzilla/show_bug.cgi?id=41939, visited in 2009.

[15] Apache Software Foundation. Tomcat web site. https://tomcat.apache.org, visited in 2009.

[16] Artisys. Cass, air traffic control simulator. http://www.artisys.aero/, visited in 2008.

[17] Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 542–565. Springer-Verlag, 2008.

[18] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 5–15. ACM, 2007.

[19] Aspectwerkz. Aspectwerkz web site. http://aspectwerkz.codehaus.org/, visited in 2009.

[20] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. Ava: automated interpretation of dynamically detected anomalies. In *ISSTA'09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 237–248. ACM, 2009.

[21] R. M. Balzer. Exdams: Extendible debugging and monitoring system. In *AFIPS Proceedings, Spring Joint Conference*, volume 34, pages 567 – 580, 1969.

[22] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972.

[23] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 1999.

[24] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 44. IEEE Computer Society, 2003.

[25] Lionel C. Briand, Yvan Labiche, and Xuetao Liu. Using machine learning to support debugging with tarantula. In *ISSRE'07: Proceedings of the 18th IEEE International Symposium on Software Reliability*, pages 137–146. IEEE, 2007.

[26] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture A System of Patterns*. John Wiley and Sons Ltd, 1996.

[27] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *ICAC'04: Proceedings of the 1st International Conference on Autonomic Computing*, pages 36–43. IEEE, 2004.

[28] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN'02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604. IEEE Computer Society, 2002.

[29] Wai-Kai Chen. *Graph Theory and Its Engineering Applications*. Advanced Series in Electrical and Computer Engineering. World Scientific Publishing Company, 1997.

[30] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE'09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society, 2009.

[31] Sampath K. Chilukuri and Kalpana Doraisamy. Symptom database builder for autonomic computing. In *ICAS'06: Proceedings of the IEEE International Conference on Autonomic and Autonomous Systems*, page 32. IEEE Computer Society, 2006.

[32] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA'02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 210–220. ACM, 2002.

[33] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *ICSE'07: Proceedings of the 29th IEEE and ACM SIGSOFT International Conference on Software Engineering*, pages 261–270. IEEE Computer Society, 2007.

[34] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE'05: Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

[35] CNet. EC wants software makers held liable for code, http://news.cnet.com/8301-10013-10237212-92.html, visited in 2009.

[36] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[37] Domenico Cotroneo, Roberto Pietrantuono, Leonardo Mariani, and Fabrizio Pastore. Investigation of failure causes in workload-driven reliability testing. In *Fourth international workshop on Software quality assurance (SOQUA'07)*, pages 78–85. ACM, 2007.

[38] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *ECOOP'05: Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 528–550. Springer-Verlag, 2005.

[39] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *ASE'09: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[40] Eclipse Foundation. Eclipse. http://www.eclipse.org/, visited in 2009.

[41] Eclipse Software Foundation. Eclipse bug 156532. https://bugs.eclipse.org/bugs/show_bug.cgi?id=156532, visited in 2009.

[42] Eclipse Software Foundation. Eclipse bug 181288. https://bugs.eclipse.org/bugs/show_bug.cgi?id= 181288, visited in 2009.

[43] Eclipse Software Foundation. Eclipse bug 221738. https://bugs.eclipse.org/bugs/show_bug.cgi?id=221738, visited in 2009.

[44] Fisher M. Ellis and J. Bruce J. *JDBC API Tutorial And Reference*. Prentice Hall, 3 edition, 2003.

[45] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.

[46] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[47] Galileo Research Group. Software-artifact infrastructure repository (SIR). http://esquared.unl.edu/sir.

[48] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95: Pro-*

ceedings of the 17th international conference on Software engineering, pages 179–185. ACM, 1995.

[49] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[50] Glassfish application server. Glassfish. https://glassfish.dev.java.net/, visited in 2008.

[51] Glassfish JIRA bug database. Glassfish issue 4255. https://glassfish.dev.java.net/issues/showbug.cgi?id=4255, visited in 2008.

[52] Glassfish user forum. Glassfish configuration issue. http://forums.java.net/jive/thread.jspa?messageID=252898, visited in 2008.

[53] Glassfish user forum. Glassfish configuration issue. http://forum.java.sun.com/thread.jspa?threadID=5249570, visited in 2008.

[54] GNOME. Web site. http://www.gnome.org/, visited in 2009.

[55] A. Gordon. *Classification*. Chapman and Hall/CRC, 2 edition, 1999.

[56] Alessandra Gorla, Leonardo Mariani, Fabrizio Pastore, Mauro Pezzè, and Jochen Wuttke. Achieving cost-effective software reliability through self-healing. *Computing and Informatics*, 2:1001–1022, 2010.

[57] Rajiv Gupta and Mary Lou Soffa. Hybrid slicing: an approach for refining static slices using dynamic information. In *FSE '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 29–40. ACM, 1995.

[58] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM, 2002.

[59] Stephen E. Hansen and E. Todd Atkins. Automated system monitoring and notification with swatch. In *Proceedings of the 7th USENIX conference on System administration*, 1993.

[60] Stephen E. Hansen and E. Todd Atkins. Automated system monitoring and notification with swatch. In *7th USENIX conference on System administration (LISA'93)*, pages 145–152. USENIX Association, 1993.

[61] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *PASTE'98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90. ACM, 1998.

[62] IBM. Eclipse test & performance tools platform. http://www.eclipse.org/tptp/, visited in 2009.

[63] IBM HRL. ConTest. https://www.research.ibm.com/haifa/projects/verification/contest/, visited in 2009.

[64] Timea Illes and Barbara Paech. An analysis of use case based testing approaches based on a defect taxonomy. *IFIP International Federation for Information Processing*, 227:211–222, 2007.

[65] InfoQ. EU Software Libability lawsuit: half say unit testing is the answer, http://www.infoq.com/news/2009/07/half-think-UT-will-help, visited in 2009.

[66] Infoworld. Software liability will hurt open source vendors, http://www.infoworld.com/d/open-source/software-liability-will-hurt-open-source-vendors-152, visited in 2009.

[67] Israeli Aerospace Industries. www.iai.co.il, visited in 2009.

[68] Peter Jackson. *Introduction to expert systems*. Addison Wesley Longman, 1999.

[69] Jean-Marc Jèzèquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[70] Lingxiao Jiang and Zhendong Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 184–193. ACM, 2007.

[71] James A. Jones, James F. Bowring, and Mary Jean Harrold. Debugging in parallel. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26. ACM, 2007.

[72] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM Press, 2002.

[73] S. Joshi and A. Orso. SCARPE: A technique and tool for selective record and replay of program executions. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 234–243. IEEE, 2007.

[74] Mariam Kamkar, Peter Fritzson, and Nahid Shahmehri. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming*, 38(1-5):625–636, 1993.

[75] Kerry Thomson. Logsurfer log monitoring tool. http://www.crypt.gen.nz/logsurfer/, visited in 2008.

[76] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th Euro-*

pean Conference on Object-Oriented Programming, pages 327–353. Springer-Verlag, 2001.

[77] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313. ACM, 2001.

[78] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[79] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[80] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, pages 6–6. USENIX Association, 1998.

[81] Bil Lewis. Debugging backwards in time. *CoRR*, cs.SE/0310016, 2003.

[82] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *ASID'06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM Press, 2006.

[83] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26. ACM, 2005.

[84] Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2006.

[85] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, 2005.

[86] C. Lonvick. The bsd syslog protocol, 2001.

[87] L. Mariani and M. Pezzè. Dynamic detection of COTS components incompatibility. *IEEE Software*, 24(5):76–85, 2007.

[88] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE'08)*, pages 117 – 126. IEEE Computer Society, 2008.

[89] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. A toolset for automated failure analysis. In *ICSE'09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 563–566. IEEE Computer Society, 2009.

[90] Leonardo Mariani and Mauro Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS'05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[91] C. L. Mendes and D.A. Reed. Monitoring large systems via statistical sampling. *International Journal of High Performance Computing Applications*, 18(2):267–277, 2004.

[92] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

[93] NASA Ames Research Center. Java Path Finder. http://javapathfinder.sourceforge.net/, visited in 2009.

[94] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[95] M. Newman. Software errors cost u.s. economy 59.5 billion annually. Technical report, NIST, 10 2002.

[96] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. Call-mark slicing: an efficient and economical way of reducing slice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 422–431. ACM, 1999.

[97] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance*, page 158. IEEE Computer Society, 2001.

[98] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[99] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notes*, 19(5):177–184, 1984.

[100] H. Pan, R. A. DeMillo, and E. H. Spafford. Failure and fault analysis for software debugging. In *COMPSAC'97: Proceedings of the 21st International Com-*

puter Software and Applications Conference, pages 515 – 521. IEEE Computer Society, August 1997.

[101] H. Pan and E. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, 1992.

[102] J. Prewett. Analyzing cluster log files using logsurfer. In *4th Annual Conference on Linux Clusters*, 2003.

[103] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003: Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, 2003.

[104] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[105] J. Rilling and B. Karanth. A hybrid program slicing framework. In *SCAM'01: Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 12–23, 2001.

[106] Charles Romesburg. *Cluster Analysis for Researchers*. Lulu.com, 1984.

[107] RTI. The economic impacts of inadeguate infrastructure for softwrae testing. Technical report, NIST, 2002.

[108] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256. IEEE Computer Society, 2004.

[109] Marc De Scheemaecker. Nano XML. http://nanoxml.cyberelf.be/, visited in 2009.

[110] SHADOWS Consortium, visited in 2009. Deliverable 7.2.4b: Report on aviation software case study, https://sysrun.haifa.il.ibm.com/shadows/publicdeliverables.html.

[111] SHADOWS Consortium. Deliverable d7.2.2b: Definition and report on iai case study. https://sysrun.haifa.il.ibm.com/shadows/publicdeliverables.html, visited in 2009.

[112] SHADOWS Consortium. SHADOWS project web site. https://sysrun.haifa.il.ibm.com/shadows/, visited in 2009.

[113] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *ICSE'99: Proceedings of the 21st international conference on Software engineering*, pages 432–441. ACM, 1999.

[114] Chad D. Sterling and Ronald A. Olsson. Automated bug isolation via program chipping. *Software: Practice and Experience*, 37(10):1061–1086, 2007.

[115] Sun. Glassfish v3 application server administration guide. http://docs.sun.com/doc/820-4495, visited in 2008.

[116] Sun. Java logging overview. http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html, visited in 2008.

[117] Sun Microsystem. Mysql web site. http://www.mysql.org/, visited in 2009.

[118] The Apache Software Foundation. Tomcat. http://tomcat.apache.org/, visited in 2009.

[119] Tomcat bug database. Tomcat configuration issue. http://www.blogjava.net/haix/archive/2008/01/16/175592.html, visited in 2009.

[120] Fumiaki Umemori, Kenji Konda, Reishi Yokomori, and Katsuro Inoue. Design and implementation of bytecode-based java slicing system. In *SCAM'03: Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, page 108. IEEE Computer Society, 2003.

[121] R. Vaarandi. Sec - a lightweight event correlation tool. In *Proceedings of the 2002 IEEE Workshop on IP Operations and Management*, pages 111–115, 2002.

[122] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IPOM'03: Proceedings of the 3rd IEEE Workshop on IP Operations and Management*, pages 119–126, 2003.

[123] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. *Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999*, pages 133–145, 1999.

[124] M. Weiser. *Program Slicing: Formal , Psychological and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, The University of Michigan, 1979.

[125] M. Weiser. Program slicing. *IEEE Transactions on Softwrae Engineering*, 10(4):352–357, 1984.

[126] Mark Weiser and Jim Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on Empirical studies of programmers*, pages 187–197. Ablex Publishing Corp., 1986.

[127] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[128] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufman, 2005.

[129] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE'99: Proceedings 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–267. ACM, 1999.

[130] Andreas Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.

[131] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[132] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42. ACM, 2005.

[133] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM, 2006.

[134] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *FSE'04: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 97–106. ACM, 2004.