Design and Development of a Probabilistic Framework for Automatic Software Fault Localization

BY

DAVIDE PAGANO
B.S., Politecnico di Milano, 2013

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2016

Chicago, Illinois

To My Family, for supporting me during Life.

ACKNOWLEDGMENTS

My sincere thanks goes to my advisor, Dr. Mark Grechanik, for offering me the opportunity to work on this project and for his support, motivation, and immense knowledge. I could not have imagined having a better advisor and mentor.

My sincere thanks also goes to Yiji Zhang, that collaborated with me through the design and implementation of our solutions. Her energy and insights helped me in all the time of research.

Last but not the least, I would like to thank my family for supporting me since I was a child and for providing me the best possible life.

DP

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (continued)

FIGURE **PAGE**

LIST OF ABBREVIATIONS

JDK	Java Development Kit
AUT	Application Under Test
MLN	Markov Logic Network
OJ	OpenJava
API	Application Program Interface
JAR	Java Archive
CIT	Combinatorial Interaction Testing
XML	EXtensible Markup Language
YAML	YAML Ain't Markup Language
AST	Abstract Syntax Tree
JDT	Eclipse Java development tools
SD	Statistical Debugging
SBFL	Spectrum-Based Fault Localization
MBD	Model-Based Diagnosis
SDO	Software Developer Organizations
UIC	University of Illinois at Chicago
AGP	Auto Generated Program

LIST OF ABBREVIATIONS (continued)

PGM	Probabilistic Graphical Model
LH	Likelihood

SUMMARY

Despite large investments in different areas of software engineering, many deployed software applications fail at some point. Even though most software applications are tested before they are released to customers, these applications still contain production (or field) functional faults that result in field failures, which have exorbitant cost and sometimes lead to the loss of human lives. Existing automatic debugging approaches are rarely applied to localizing production faults for field failures due to their limitations. The goal of this thesis is therefore to create a novel theoretical foundation that allows stakeholders to predict and localize faults for field failures automatically with a high degree of precision using symptoms only (e.g., the sign of the output value is incorrect) and without instrumenting deployed applications to collect runtime data, thus avoiding the overhead, and without having any tests with oracles to uncover the fault, without performing contrasting successful and failed runs, and without collecting runtime data from field failures. With this theoretical foundation, researchers can collaborate more closely in planning the future of fault localization by expanding each other's results based on probabilistic graphical models as common abstractions.

We propose a novel framework for *Automatically Localising Faults For Functional Field Failures in Applications (ALFFFFFA or α^5)* that enables stakeholders to enter symptoms of a failure that occurs during deployment of a given application and the values of the input and configuration parameters, and α^5 will return not only locations in the source code that are likely to contain specific faults, but also it will show navigation paths from a suggested faults to the failure and it will recommend modifications to the code that can fix these faults. We created, evaluated and deployed:

- new theories, algorithms and techniques for automatically obtaining probabilistic graphical models that approximate specific fault models for software applications,
- a novel way in which model-based differential diagnoses are used to perform abductive reasoning to localize production faults given symptoms for field failures,
- a comprehensive experimentation framework for evaluating the algorithms for localizing production faults in α^5.

In addition to localizing production faults, α^5 can be used as a broad experimental platform for creating and testing hypotheses for various software debugging and testing ideas, e.g., for guiding test selection and prioritisation. Broader impacts include advances that enhance the quality of software applications by enabling stakeholders to quickly localise production functional faults in deployed software applications and new educational course content. The educational innovation of this project is in developing an integrated approach to teaching by applying probabilistic graphical models to software engineering problems.

Intellectual Merit involves:

- a novel abstraction that not only enables speculative construction of the predictive fault model that is specific for every software application, but also it allows stakeholders to obtain specific information on how a suggested fault results in an error state that propagates through the application to cause this field failure. To the best of our knowledge, this abstraction has never been created and applied to automatic debugging of field failures.

- a novel combination of our abstraction, model-based diagnosis, fault injection and program analysis and that will lead to practical tools for improving the quality and reducing the cost of engineering software, and
- new framework α^5 developed, evaluated, that will be applied to different open-source and commercial applications, and made available to the broader community.

CHAPTER 1

OVERVIEW

Imagine the world where each human has a distinct anatomy and reacts to viruses and bacteria in ways that are totally different from other humans. The medical science, as we know it, will cease to exist. When we get sick, we will pray that the medical staff will be able to understand the nature of our diseases that uniquely affect each individual and ways to cure them, which will be specific for each human. Does it sound horrible? This is the state of the art, and very few problems impact people more negatively than field failures, where deployed software behaves incorrectly. Just like distinct human anatomies would prevent medical professionals from quickly diagnosing diseases using symptoms, production fault localisation requires a huge effort from software professionals, since each software application has its own unique structure and programmers must spend a lot of time to understand it even for smaller applications. Not only do field failures zap every shred of customer's confidence in software applications, but also they cost dearly, sometimes in human lives, since software applications support all aspects of our lives.

Nevertheless, despite large investments in different areas of software engineering, many deployed software applications still fail at some point. Even though a majority of software applications are tested before they are released to customers, these applications contain *production functional faults* that result in field failures [1–3]. Multiple reasons exist why software testing is not fully effective at finding all faults [4–8]. It suffices to say that production faults have exorbitant cost and sometimes lead to the loss of human lives [9].

The cost of finding and fixing a production fault is about 300 times higher than during requirements and coding phases [10]. Whereas it takes minutes to localize and fix a bug during unit testing, it often takes days, weeks, or months to localize faults for field failures. The U.S. average for defect removal efficiency is around 85% with approximately five defects per function points on average are delivered to customers in applications [6, pages 288-291]. Once a field failure occurs, customers are highly likely to stop using their buggy software applications until the fault is fixed and they apply large financial penalties to their *Software Developer Organizations (SDOs)* that built these applications. The average downtime costs vary across industries, from approximately $90,000 per hour in the media sector to about $6.5 million per hour for large online brokerages, and therefore, field failures cost billions of dollars annually just to the U.S. economy alone [11].

CHAPTER 2

THE PROBLEM STATEMENT

Ideally, we need an approach for production fault localisation which takes as inputs symptoms of a field failure entered by stakeholders, who are notified what statements and what specific operations and commands in these statements are likely to contain production faults that cause this failure. Moreover the navigation and control flow paths will be shown from suggested faults to failures. The approach should not require stakeholders to build any additional tests with oracles to localize faults, and it should not require customers to run the instrumented application to collect execution traces (different studies show that average performance overhead of instrumentation is between 30% and 150% [12–14] and call-level only instrumentation in BugRedux has at least 17% overhead [4]), or to carve large runtime states from the application to deliver them to the SDO to reproduce the failure. Instead, this ideal approach should use a medical analogy of diagnosing human diseases – a patient enters symptoms and the Model-Based Diagnosis (MBD) system uses its knowledge base and its reasoning engine to issue specific diagnoses with suggested treatments and analyzes the results of the treatment to localize problems further. Despite hundreds of different automated debugging approaches, some of which deal with different aspects of fault localisation for field failures [4, 15–22], none of them works even remotely similar to this ideal approach.

A problem of localising production faults quickly and without incurring significant cost is pervasive. Yet, in addition to our findings, the recent investigation by Parnin and Orso also reveals that programmers use their intuition instead of relying on automated debugging tools [23, 24]. Thus, a fundamental problem of automated debugging is how to automatically localise functional production faults in deployed software applications with a high degree of precision:

- using only symptoms of the field failures and input values,
- without deploying *instrumented* applications for customer's use,
- without collecting any runtime data from the customer,
- without having any tests with oracles,
- without performing successful and failed runs at the customer's site,
- without collecting large amounts of state information from field failures.

To the best of our knowledge, there is no solution to this big and important problem.

CHAPTER 3

OVERVIEW OF AUTOMATIC SOFTWARE FAULT LOCALISATION APPROACHES

In software engineering finding root causes of functional failures in software applications automatically is the main goal [25]. Existing fault-localisation approaches can be roughly divided into three categories: *Spectrum-Based Fault Localization (SBFL)*, *Statistical Debugging (SD)* and *Model-Based Diagnosis (MBD)* (see detailed analysis of related work in Section 4). SBFL is a collection of statistical techniques and algorithms for correlating the behaviors of software applications with their failures by ranking statements with likelihoods of them containing faults [26]. Typically, with SBFL, a test suite is run to collect the Application Under Test (AUT)'s execution traces for passing and failing tests (i.e., program spectra), and SBFLs algorithms use this spectra to pinpoint locations of faults in the source code of the AUT with some degree of accuracy.

SBFL approaches are poorly suited for localising faults for field failures for two main reasons. First, since SBFL approaches require one or more tests that fail for a production fault, this fault can be discovered at the SDO and fixed before the application is deployed – hence there will not be any production faults by definition. Second, it takes a manual and laborious effort to develop tests that reproduce the field failure, thus increasing the time of finding the fault and subsequently the cost of fixing this fault for deployed software applications. Even though this problem can be alleviated by using automatic test generation approaches [27–30], creating tests is still expensive, since it is difficult ① to generate tests that trigger faults that will result in specific field failures and ② to determine that the AUT fails, since these tests must contain meaningful *oracles* (i.e., methods for checking whether the AUT has behaved correctly for a particular execution [31]). Creating oracles automatically is one of the most challenging problems of software testing [31–35]. One way or another, SBFL is not typically used for field failures.

Statistical Debugging (SD) is one of the most promising automatic approaches for localising production functional faults [36–43]. In SD, a software application is instrumented and the values of program predicates are collected (e.g., results of evaluations of conditional expressions). Then, a statistical model is built by contrasting the information about predicates for successful and failed application runs. The precision of SD approaches depends on three main factors:

- the granularity of the instrumentation of the application and its libraries, which imposes significant overhead and may be difficult to use for many performance-sensitive applications;
- deep insight that is required for selection of the meaningful predicates whose evaluations should be representative of properties of specific faults;
- the need for many successful and failed runs that exercise these predicates, which is partially alleviated using tools like MIMIC [44].

Using too many predicates for instrumentation in SD results in a significant performance penalty for an application and using too few predicates results in a poor quality statistical model that is less effective in localising faults. SD is used in Microsoft Visual Studio IDE for `.NET` [38] and in the *Cooperative Bug Isolation (CBI)* project for some flavors of the Linux operating system whose packages are pre-instrumented [45–47].

Finally, in *model-based diagnosis (MBD)*, a diagnostic reasoning engine performs a variety of diagnostic tasks to infer application's behavior from its model [48]. In the traditional MBD, models for applications should be created by stakeholders and supplied to a diagnostic reasoning engine, and it is an intellectually intensive, error-prone and expensive process. These models should include symptoms of failures that the application exhibits for different faults and input values. At this point, MBD has a very limited use in localising faults in software applications – notable exceptions are the database RETAIN at IBM [49] and the approach AFID [50] where failure symptoms and corresponding faults for software applications are recorded, however, to the best of our knowledge these approaches do not use automated fault localisation.

CHAPTER 4

BACKGROUND AND RELATED WORK

Localising production faults for field failures is a relatively new area of research in automated debugging. F3 is the closest related approach to α^5, since it uses BugRedux [4] to generates multiple failing and passing executions that are similar to the observed field failure to localise faults [15]. Other field failure debugging aids include CHRONICLER, a tool that captures non-deterministic inputs to applications to reproduce of client executions [51]; a dynamic symbolic execution tools called SymCon that selects functions that are difficult to execute and generates input values to reach these functions and reproduce crashes [17]; ReCrash, an approach to reproduce failures efficiently with low execution overhead [18]; ADDA, a technique for recording, reproducing, and minimizing failing executions that enables and supports in-house debugging of field failures [19], an approach for detecting hardware faults that propagate to software layers in field [20] and a framework for in-field carving and replaying differential unit tests [21]. A tool that records crashes with the input enables stakeholders to study these faults and keep their history [22]. Many ideas that are proposed in these approaches can be used in the context of α^5; however, unlike these approaches α^5 does not require programmer to capture failed executions or to instrument the application and to carve its state or to collect execution traces at the customer's site, and α^5 uses a fault model that requires only symptoms of failures to localise faults.

Markov Logic Networks (MLNs) are used to detect malware in Android software [52], in smart home systems that react to human voice [53], to detect topics [54], to localise vehicles on the road [55], to reason about software goal models [56], to correct layouts of documents [57], to extract biomolecular events [58], to detect malicious behavior against software [59], to generate realistic grammatical errors for computer-assisted language learning [60], and for many other tasks. Some related research uses probabilistic models that are extracted from applications or supplied by stakeholders [61–64]. Zhang suggested to use MLN for software bug localisation by manually creating an MLN where program features (e.g., test coverage and prior information about bugs) are encoded in logic formula [65]. Unlike α^5, the proposed approaches are based on SBFL or manually extracted artifacts and require running tests with oracles, and it is unclear how scalable and generalizable this use of MLN is for bug localisation. In contrast, α^5 makes a fully automated use of MLNs that it creates without human intervention using sensitivity analysis of the AUT.

As we already stated in Section 3, SBFL approaches are poorly suited for production fault localisation even though it is a large research area that includes Δ-debugging [66] and its variations [26, 66–72]. Various extensions of SBFL include localising concurrency bugs [73–79] and various improvements to increase the precision and speed of fault localisation given program spectra for functional and performance failures [80–124]. Somewhat complementary to α^5 approaches from which we borrow some ideas alleviate the problem of coincidentally correct test cases [125–129] and use the similarity coefficients for different artifacts e.g., execution traces [130–135]. We will utilize ideas from FAULTTRACER and its extension use change-impact analysis with SBFL to rank program edits [136, 137] and BUGEX that generates test cases to systematically isolate faults from a single failing test run [138]. In contrast to all these approaches, α^5 does not require test cases with oracles, which are not available anyway for production faults. Bayesian networks and reasoning techniques are used to learn statistical properties using AUT's spectrum [139, 140]. Many other approaches improve various aspects of SD and SBFL or combinations thereof and applying them to different domains [141–187]. However, some ideas can be used to extend α^5; e.g., combining test prioritization and fault localization techniques [188] can be used to prioritize tests based on the MLN fault model, since some modules are more sensitive to injected faults.

A number of approaches use heuristic search methods to localise and repair faults, e.g., evolutionary tools for fault localisation and repair [189–196] including a method for repairing Java bytecode and x86 assembly code [197] and a technique for searching what predicate to switch [198]. These approaches are related to α^5, since it attempts to temporarily inoculate software against some field failures; however, these approaches do not provide explanations of relations of suggested faults to failures and many of them require tests with oracles.

Many fault localisation approaches use models, some of which are created by stakeholders and others are derived from software artifacts, some of them probabilistic to check if the AUT behaves correctly or it regresses with respect to the model and then provide ranked diagnoses that specify where faults are in the AUT [199–221]. Some approaches specifically use MBD and SBFL to derive models [199, 222–246]. Similar to α^5, these approaches combine the idea of MBD with other fault localisation techniques that use machine learning; however, in contrast to α^5, many of these approaches require tests with oracles, manually derived sophisticated logic constraints or instrumenting the application to collect passing and failing runs during its deployment.

Some related work, like α^5 uses hybrid methods with probabilistic models that are created using a spectrum of AUT's executions with test cases [103, 217, 227, 232, 237, 247–279]. Also related to α^5 in this category are BARINEL that deduce multiple-fault candidates and their probabilities [280] and Zoltar that localises multiple faults [234, 281]. Key differences between these approaches and α^5 is that these approaches either requires passing and failing runs for training using test cases with oracles or via running instrumented applications and collecting information about predicates thus incurring performance penalty or by building fault models and symptoms databases manually.

CHAPTER 5

AN ILLUSTRATIVE EXAMPLE OF LOCALIZING SOFTWARE FAULTS

To understand the idea behind α^5, we demonstrate now how it works through an illustrative example using the following code:

// Inputs: dividend, divisor; // Outputs: quotient, remainder; public static void execute(int dividend, int divisor) 1) int tmpQuotient=0; 2) int tmpRemainder = dividend; 3) do 4) tmpQuotient = tmpQuotient +1; 5) tmpRemainder = tmpRemainder - divisor; 6) while(tmpRemainder ¿= divisor); 7) int quotient = tmpQuotient; //System.out.println(”Quotient = ” + quotient); 8) int remainder = tmpRemainder; //System.out.println(”Remainder = ” + remainder);

The program takes two input variables, $\mathit{dividend}$ and $\mathit{divisor}$ and it computes the division $\frac{\mathit{dividend}}{\mathit{divisor}}$ by subtracting $\mathit{divisor}$ from $\mathit{dividend}$ and assigning the result of the subtraction to $\mathit{divisor}$ in a loop in lines 3– 6 until the value of $\mathit{divisor}$ is greater or equal to zero and less than $\mathit{dividend}$.

The program already has a latent bug – the loop should start with the `while` condition. Consider the execution with the following inputs: $\mathit{dividend} = 3, \mathit{divisor} = 5$. The correct output values are $\mathit{quotient} = 0, \mathit{remainder} = 3$, however, the program returns incorrect results on both outputs $\mathit{quotient} = 1, \mathit{remainder} = -2$.

With the inputs: $\mathsf{dividend} = 1, \mathsf{divisor} = 1$, the output values are correct $\mathsf{quotient} = 1, \mathsf{remainder} = 0$, however, with the inputs: $\mathsf{dividend} = 1, \mathsf{divisor} = 0$ and $\mathsf{dividend} = 1, \mathsf{divisor} = -1$ the loop is never exited.

Suppose that the mutation phase includes two mutants:

- A first mutant (like the AOR operator), able to switch the sign of divisor at line 5 (solving in this way the first bug):

```
do {
    tmpQuotient = tmpQuotient +1;
    tmpRemainder = tmpRemainder + divisor; //line 5
} while( tmpRemainder >= divisor );
```

- A second mutant (like the COR operator), able to switch loop conditions, solving the second bug:

```
while( tmpRemainder >= divisor ) {
    tmpQuotient = tmpQuotient +1;
    tmpRemainder = tmpRemainder - divisor;
}
```

5.0.1 Execution

After the mutation phase, the original and the mutants code (obtained by applying the two mutants above described) will be executed. Table I, Table II, Table III, Table IV, Table V, Table VI show the execution with 6 different inputs, where $dividend \in [0, 1]$ and $divisor \in [-1, 1]$.

TABLE I

AUT EXECUTED WITH INPUTS (0,-1)

Original	Mutant 1	Mutant 2
1. tmpQuotient=0 2. tmpRemainder=0 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=0-(-1)=1 6. while(1>=-1) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=1-(-1)=2 6. while(2>=-1) true 3. do . . . timeout	1. tmpQuotient=0 2. tmpRemainder=0 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=0+(-1)=-1 6. while(-1>=-1) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=-1+(-1)=-2 6. while(-2>=-1) false 7. quotient=1 8. remainder=-2	1. tmpQuotient=0 2. tmpRemainder=0 3. while(0>=0) true 4. tmpQuotient=0+1=1 5. tmpRemainder=0-0=0 3. while (0>=0) true 4. tmpQuotient=1+1=2 5. tmpRemainder=0-0=0 . . . timeout
Predicates produced	HasFault(5,mutant1,input1) $loop()$ $!timeout()$ $Diff(7)$ $Diff(8)$	HasFault(3,mutant2,input1) $loop()$ $timeout()$

TABLE II

AUT EXECUTED WITH INPUTS (0,0)

Original	Mutant 1	Mutant 2
1. tmpQuotient=0 2. tmpRemainder=0 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=0-0=0 6. while (0>=0) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=0-0=0 6. while(0>=0) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=0 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=0+0=0 6. while (0>=0) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=0+0=0 6. while(0>=0) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=0 3. while(0>=0) true 4. tmpQuotient=0+1=1 5. tmpRemainder=0+0=0 3. while(0>=0) true 4. tmpQuotient=1+1=2 5. tmpRemainder=0+0=0 . . . timeout
Predicates produced	HasFault(5,mutant1,input2) $\mathtt{loop}()$ $\mathtt{timeout}()$	HasFault(3,mutant2,input2) $\mathtt{loop}()$ $\mathtt{timeout}()$

TABLE III

AUT EXECUTED WITH INPUTS (0,1)

Original	Mutant 1	Mutant 2
1. tmpQuotient=0 2. tmpRemainder=0 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=0-1=-1 6. while(-1>0) false 7. quotient=1 8. remainder=-1	1. tmpQuotient=0 2. tmpRemainder=0 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=0+1=1 6. while(1>=1) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=1+1=2 6. while(2>=1) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=0 3. while(0>=1) false 7. quotient=0 8. remainder=0
Predicates produced	HasFault(5,mutant1,input3) $\mathtt{loop}()$ $\mathtt{timeout}()$	HasFault(3,mutant2,input3) $!\mathtt{loop}()$ $!\mathtt{timeout}()$ $\mathtt{Diff}(7)$ $\mathtt{Diff}(8)$

TABLE IV

AUT EXECUTED WITH INPUTS (1,-1)

Original	Mutant 1	Mutant 2
1. tmpQuotient=0 2. tmpRemainder=1 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=1-(-1)=2 6. while(2>=-1) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=2-(-1)=3 6. while(3>=-1) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=1 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=1+(-1)=0 6. while(0>=-1) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=0+(-1)=-1 6. while(-1>=-1) true 3. do 4. tmpQuotient=2+1=3 5. tmpRemainder=-1+(-1)=-2 6. while(-2>=-1) false 7. quotient=3 8. remainder=-2	1. tmpQuotient=0 2. tmpRemainder=1 3. while(1>=-1) 4. tmpQuotient=0+1=1 5. tmpRemainder=1-(-1)=2 3. while(2>=-1) 4. tmpQuotient=1+1=2 5. tmpRemainder=2-(-1)=3 . . . timeout
Predicates produced	HasFault(5,mutant1,input4) $\mathsf{loop}()$ $!\mathsf{timeout}()$ $\mathsf{Diff}(7)$ $\mathsf{Diff}(8)$	HasFault(3,mutant2,input4) $\mathsf{loop}()$ $\mathsf{timeout}()$

TABLE V

AUT EXECUTED WITH INPUTS (1,0)

Original	Mutant 1	Mutant 2
1. tmpQuotient=0 2. tmpRemainder=1 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=1-0=1 6. while(1>=0) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=1-0=1 6. while(1>=0) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=1 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=1+0=1 6. while(1>=0) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=1+0=1 6. while(1>=0) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=1 3. while(1>=0) 4. tmpQuotient=0+1=1 5. tmpRemainder=1-0=1 3. while(1>=0) 4. tmpQuotient=0+1=1 5. tmpRemainder=1-0=1 3. while(1>=0) . . . timeout
Predicates produced	HasFault(5,mutant1,input5) $\mathit{loop}()$ $\mathit{timeout}()$	HasFault(3,mutant2,input5) $\mathit{loop}()$ $\mathit{timeout}()$

TABLE VI

AUT EXECUTED WITH INPUTS (1,1)

Original	Mutant 1	Mutant 2
1. tmpQuotient=0 2. tmpRemainder=1 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=1-1=0 6. while(0>=1) false 7. quotient=1 8. remainder=0	1. tmpQuotient=0 2. tmpRemainder=1 3. do 4. tmpQuotient=0+1=1 5. tmpRemainder=1+1=2 6. while(2>=1) true 3. do 4. tmpQuotient=1+1=2 5. tmpRemainder=2+1=3 6. while(3>=1) true . . . timeout	1. tmpQuotient=0 2. tmpRemainder=1 3. while(1>=1) 4. tmpQuotient=0+1=1 5. tmpRemainder=1-1=0 3. while(0>=1) 7. quotient=1 8. remainder=0
Predicates produced	HasFault(5,mutant1,input6) $\mathit{loop}()$ $\mathit{timeout}()$	HasFault(3,mutant2,input6) $\mathit{loop}()$ $!\mathit{timeout}()$

5.0.2 PGM Weights

After the AUT and its mutants gets executed, traces gets produces and analyzed. For this example, the traces will be utilized in order to instantiate the weights for nodes and edges between them, but in the actual inference process a more complicated algorithm is executed behind the scenes. Moreover, using a simple method as counting makes straightforward for the reader to understand the process in its entireness.

From the execution traces we obtain the following tables:

- Table VII contains the counting of the occurrences of each predicates. For example HasFault(5,mutant1) appears in 6 of the 12 executions showed in 5.0.1, therefore the counting is 6.
- Table VIII contains the counting of the occurrences of each couples of predicates. For example HasFault(3,mutant2) appears altogether with !`loop`() only in 1 executions, therefore the value in the table is 1.
- Table IX contains the same counting of Table VIII, but dividend by the counting of the predicate appearing in the row. For example HasFault(3,mutant2) appears altogether with !`loop`() only in 1 executions out of 6 executions in which HasFault(3,mutant2) was produced, therefore the value corresponds to $1/6 = 0.1\overline{6}$

TABLE VII

OCCURRENCES OF PREDICATES DURING EXECUTION PHASE

Predicate	**# Occourences**
HasFault(5,mutant1)	6
HasFault(3,mutant2)	6
loop()	11
!loop()	1
timeout()	8
!timeout()	4
Diff(7)	3
Diff(8)	3

TABLE VIII

OCCURRENCES OF PREDICATES AT THE SAME TIME

# (PredA, PredB)	loop()	!loop()	timeout()	!timeout()	Diff(7)	Diff(8)
HasFault(5,mutant1)	6	0	4	2	2	2
HasFault(3,mutant2)	5	1	5	1	1	1
loop()			9	2	2	2
!loop()				1	1	1
timeout()						
!timeout()					3	3
Diff(7)						3

TABLE IX

CONDITIONAL LIKELIHOOD (LH) FOR EACH COUPLE OF PREDICATES

LH (PredA|PredB)	loop()	!loop()	timeout()	!timeout()	Diff(7)	Diff(8)
HasFault(5,mutant1)	1	0	0.67	0.33	0.33	0.33
HasFault(3,mutant2)	0.83	0.17	0.83	0.17	0.17	0.17
loop()			0.82	0.18	0.18	0.18
!loop()			0	1	1	1
timeout()					0	0
!timeout()					0.75	0.75
Diff(7)						1

5.0.3 Markov Logic Network

With the structured we just introduced, we are now able to build and ground our PGM. Figure 1 shows the PGM, including weights on edges (Table IX) and weight on the nodes (Table VII, where the values have been divided by the total number of executions).

A main features that distinguish MLNs from Bayesian Networks, is the possibility of introducing first order logic rules, that will be activated during the navigation of the graph.

This rules are part of the MLN, since if there is a possible grounding of two predicates in a rule, then there is an edges between the two predicates in the graph. Therefore they need to be consider in conjunction with the graph in Figure 1.

Four logic rules for reasoning about the execution of the program are shown in Table X. Each rule has some confidence value expressed as a number or the infinity ∞. The latter means that the rule always holds, whereas the former means that the rule may hold with some likelihood.

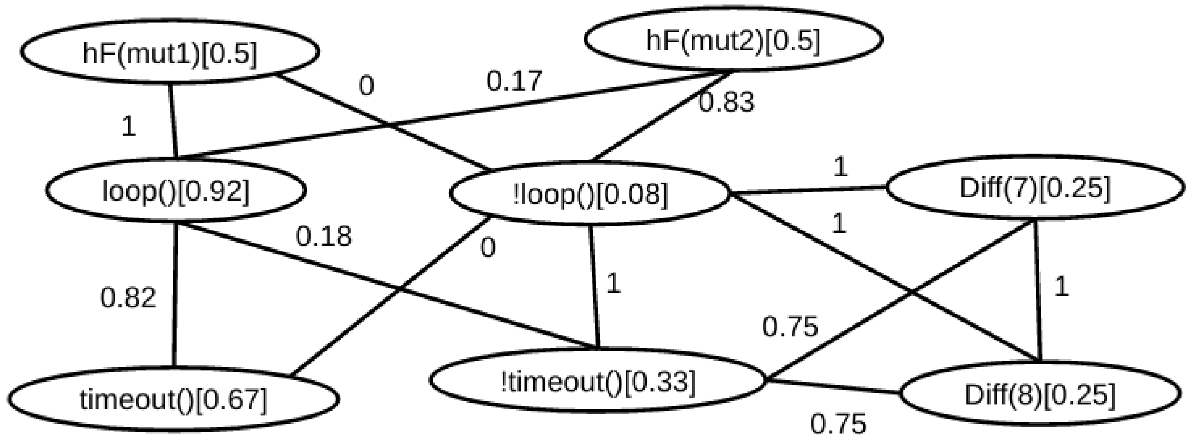


Figure 1. Grounded PGM

The first rule says that if a statement has a fault (i.e., `hF`) for some type of mutant, then the loop body will be entered in, say, 60% of executions with different combinations of input values. The term variable `mut` is not grounded, so it can represent any mutant. Instantiation of a rule is accomplished by executing this program with some input values and substituting actual values for term variables. We say that a rule is grounded if its predicates are instantiated.

Conversely, the second rule says that the opposite is true in 40% of cases.

The third rule says that if the state differs from the state of the execution of the original (unmutated) program with the same input values, then the state will differ at some other statement in 80% of cases.

Finally, the last rule states that if the body of the loop is not entered and there is no timeout, then the values of the output `remainder` and the output `quotient` are correct.

In general, these rules reflect the summary of the knowledge base of stakeholders about the behavior of the program.

TABLE X

RULES USED FOR MLN

Rule	**Weight**
HasFault(mut) =>loop()	60%
HasFault(mut) =>!loop()	40%
Diff(stmt) =>Diff(stmt)	80%
!loop() ˆ!timeout() =>!Diff(7)ˆ!Diff(8)	∞

We are now going to analyze two use cases in which the user reports different faults.

5.0.4 Use case 1

We consider now the case in which the user reports Diff(7), meaning that he experienced a difference in the value at line 7. We want to suggest the most probable location to change in the program in order to fix this bug.

As user reports Diff(7), the likelihood of the respective node is set to 1, since at this point we know for sure that the difference happened. However, we can't update the likelihood of Diff(8), since the user didn't report anything about the node (as will be in case 2), so he may not know if there is or not a difference in the value at line 8.

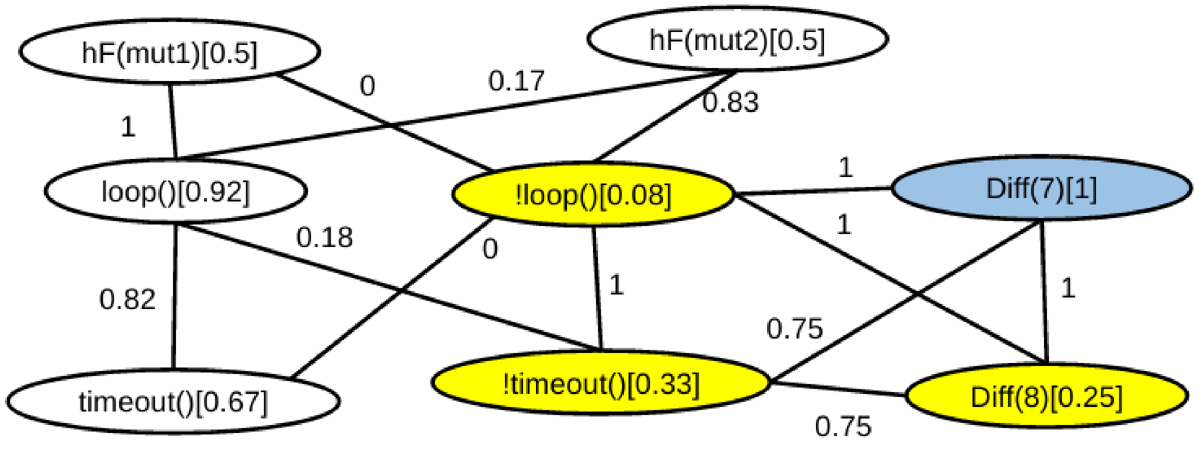


Figure 2. User reports Diff(7)

From the node corresponding to Diff(7) we can navigate to three different node and the likelihood of each one of them will be updated according to a variant of the Bayesian chain rule [282]:

$NewLH(next\_node) = LH(starting\_node) * Weight(starting\_node, next\_node) * OldLH(next\_node) * Weight(rules\_firing)$

Of course, different PGM-based systems use different algorithms to recalculate the likelihoods, and our goal is to illustrate one of the examples to show how faults can be localized.

For the node corresponding to the Diff(8) statement, considering that rule number 3 fires with weight 0.8 (Table X) the computations are as follows:

$NewLH(Diff(8)) = LH(Diff(7)) * Weight(Diff(7), Diff(8)) * OldLH(Diff(8) * Weight(rule3) = 1 * 1 * 0.25 * 0.8 = 0.2$

Similarly, the new LH for !timeout() and !loop() yield 0.248 and 0.08 as result respectively. At this point the node with the highest likelihood is chosen (!timeout) and the same algorithm is applied again. Figure 3 shows the update situation after the choice, where timeout is now excluded from the set of choices since !timeout() was included.

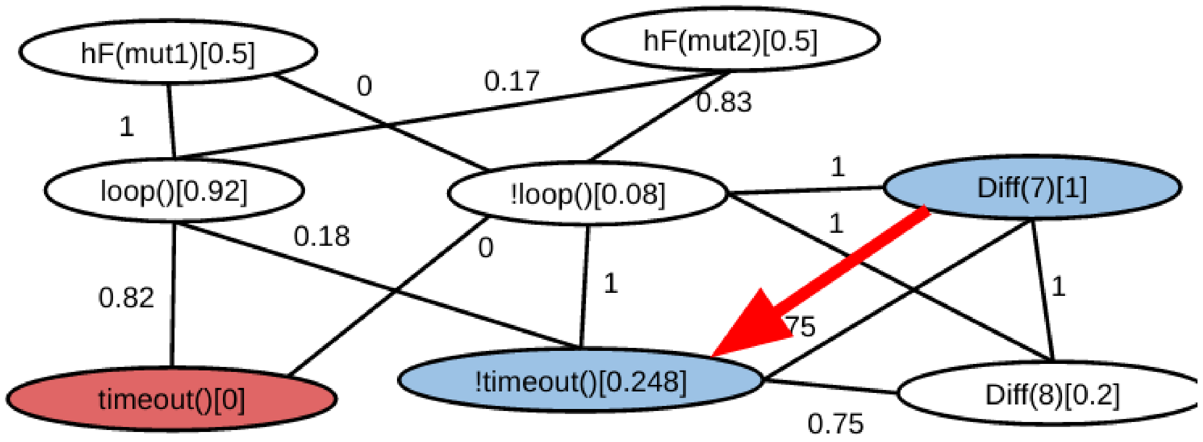


Figure 3. !timeout() is chosen as next node after updating LHs

At the next step, from !timeout() three different nodes can be chosen next: loop(), !loop() and Diff(8). Diff(7) cannot be chosen anymore since it was already selected. Updating the LHs yield to the choice of loop() and the exclusion of !loop() as shown in Figure 4.

Again, the same process is applied from the loop() node, resulting in the winning of the HasFault(5,mutant1) node. Figure 5 shows the final situation. At this point, since mutant1

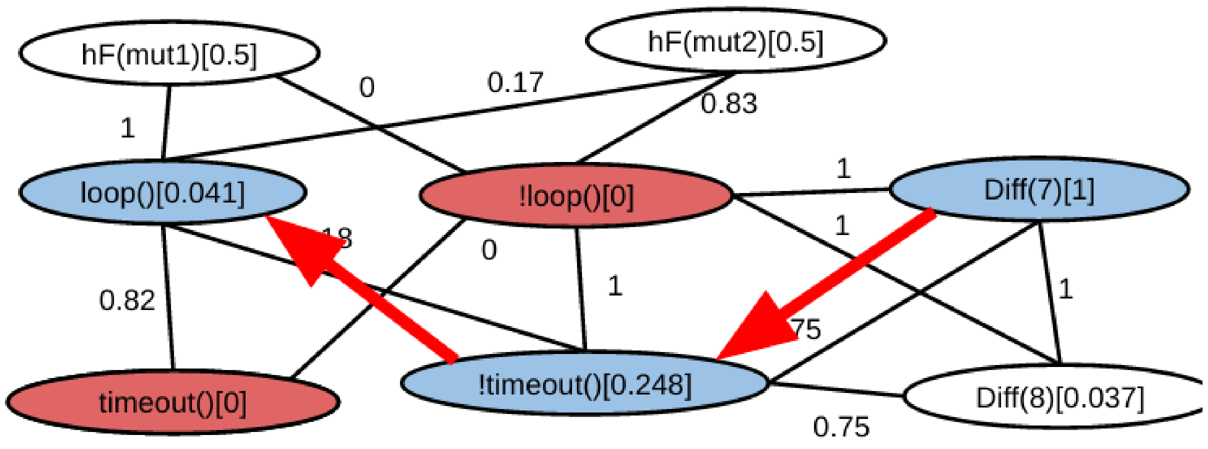


Figure 4. `loop()` is chosen as next node after updating LHs

was applied at line 5, the line of code 5 is suggested to the developer as most probable bugged statement and ranked at the top in the list of statements.

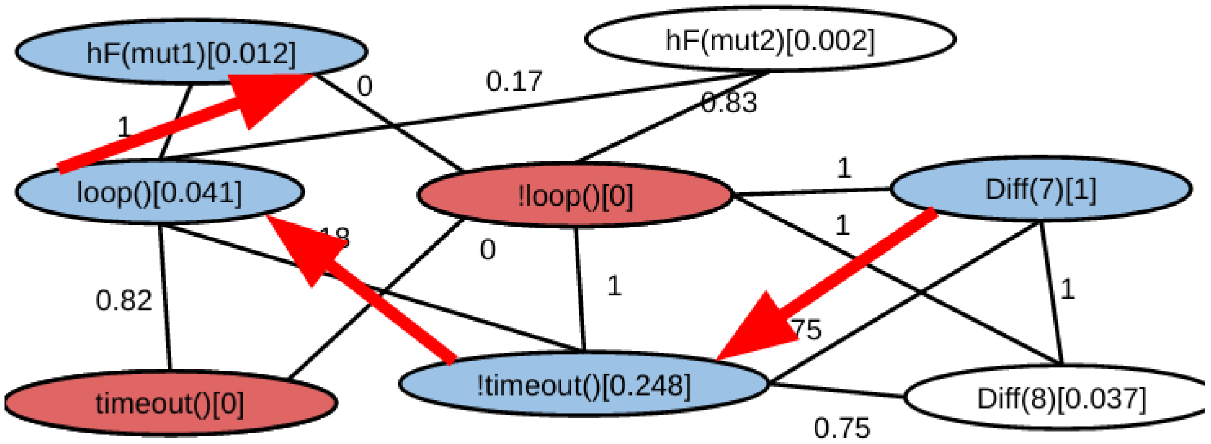


Figure 5. Mutant 1 is suggested as location to investigate to fix the bug.

5.0.5 Use case 2

Let's consider now the case in which the user experienced not only a difference in the value at line 7, but also a different to the value at line 8, therefore reporting not only Diff(7), but also Diff(8). We want to suggest the most probable location to change in the program in order to fix this bug.

As user reports Diff(7) and Diff(8), the likelihood of both nodes is set to 1, since at this point we know for sure that the difference happened. The initial situation is shown in Figure 6.

In the initial step we can navigate to both !`loop`() and !`timeout`(). After updating the scores by adding the new LH calculated from `Diff`(7) and the new LH calculated from `Diff`(8), !`loop`() results in the highest LH, and is chosen as next node, while `loop`() gets excluded.

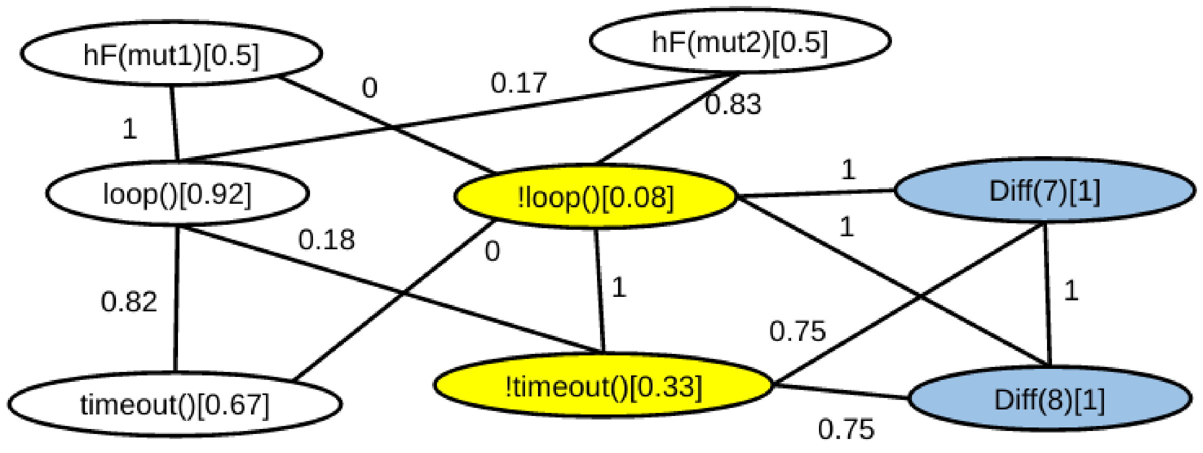


Figure 6. User reports Diff(7) and Diff(8)

The same algorithm gets repeated for !$\mathtt{loop}$() and new LHs are computed as shown in Figure 7. $\mathtt{HasFault}(5, \mathtt{mutant1})$ and $\mathtt{timeout}()$ is now 0 since the weights on the edges starting from !$\mathtt{loop}$() are 0.

Again, the same process is applied from the !$\mathtt{loop}$() node, resulting in the winning of the HasFault(3,mutant2) node. Figure 8 shows the final situation. At this point, since mutant2 was applied at line 3, the line of code 3 is suggested to the developer as most probable bugged statement and ranked at the top in the list of statements.

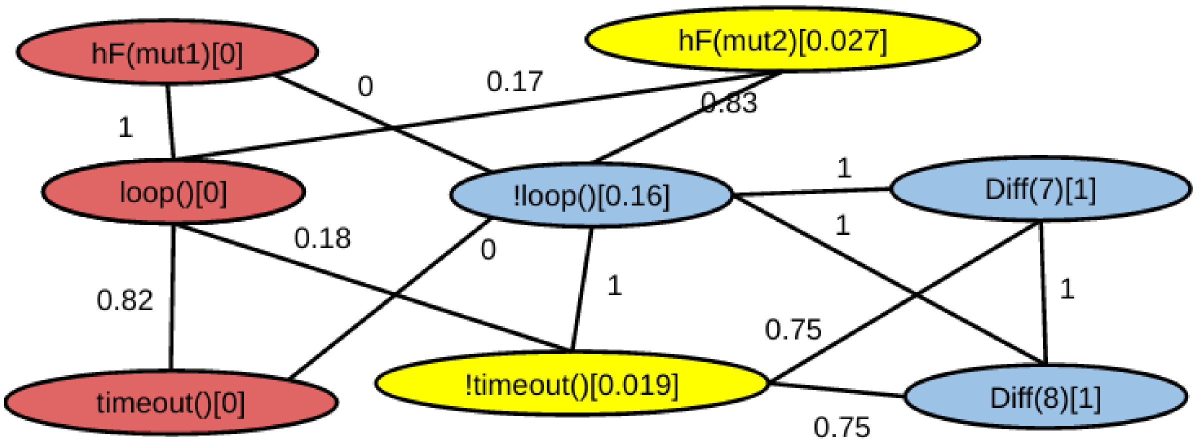


Figure 7. !`loop()` is chosen as next node and LHs are updated

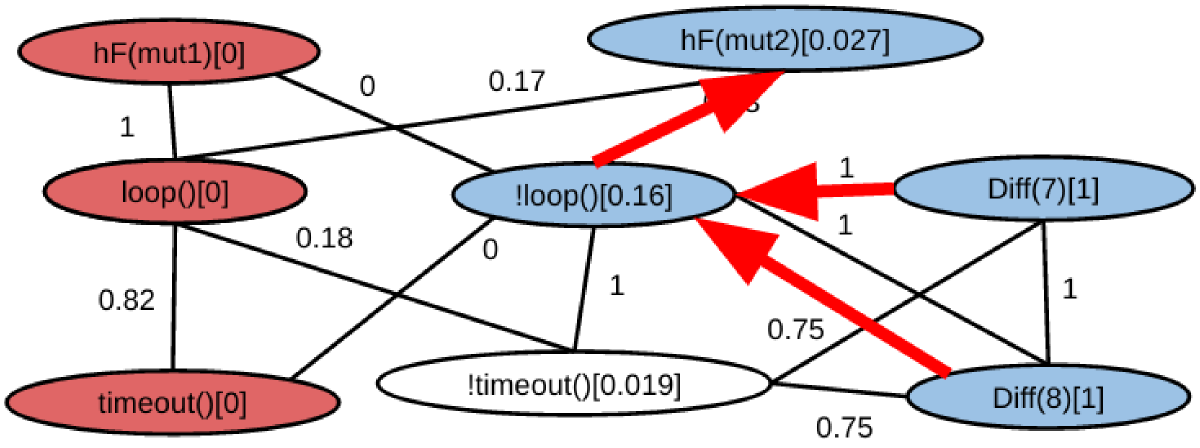


Figure 8. Mutant 2 is suggested as location to investigate to fix the bug.

CHAPTER 6

α^5 MAIN IDEA

Keeping the example in mind let's now formalize the concepts behind it. A key idea for α^5 is to enable Model-Based Diagnosis (MBD) for production fault localisation by automatically creating a fault model for a software application. MBD works in many engineering disciplines and medical sciences because effective models are used to obtain diagnoses from symptoms [283–287]. A fault model includes constraints, abstractions, and actions that specify incorrect or unacceptable behavior of an engineered system [288,289] and in medical sciences fault models of humans include their anatomy and physiology and they show how human bodies react to different viruses and bacteria that result in diseases. However, when it comes to software, a generic fault/failure model is routinely used to specify that executing code that contains a fault results in an error state that propagates to some output to cause a failure [7, page 12] [3]. In contrast, highly specific fault models are created for various devices and constructions in electrical and mechanical engineering to enable MBD that applies reasoning to these specific models [290–296]. We extend the medical analogy with *differential diagnosis (DDX)*. Similar to how a medical diagnostician asks a patient additional questions, we will explore as part of our research program how to collect additional information to further fault localization using DDX. In this thesis, creating and maintaining fault models automatically that are highly specific to given software applications is the central idea.

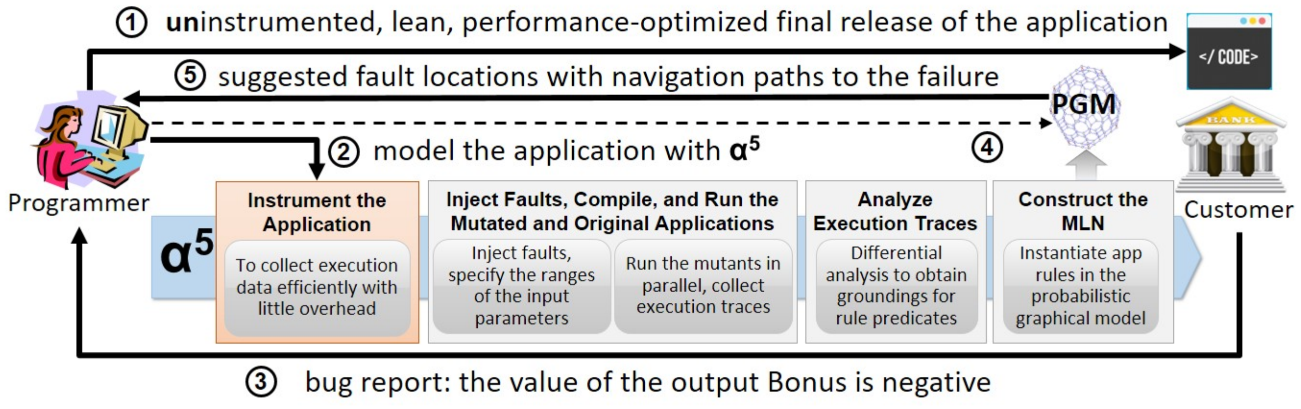

① uninstrumented, lean, performance-optimized final release of the application
⑤ suggested fault locations with navigation paths to the failure
PGM
② model the application with α⁵
④
Programmer
Customer
</ CODE>
BANK
α⁵
Instrument the Application
To collect execution data efficiently with little overhead
Inject Faults, Compile, and Run the Mutated and Original Applications
Inject faults, specify the ranges of the input parameters
Run the mutants in parallel, collect execution traces
Analyze Execution Traces
Differential analysis to obtain groundings for rule predicates
Construct the MLN
Instantiate app rules in the probabilistic graphical model
③ bug report: the value of the output Bonus is negative

Figure 9. A workflow for localizing production faults with α^5.

We introduce a novel abstraction that is expressed as a template for a *Probabilistic Graphical Model (PGM)* in which nodes represent random variables and edges specify dependencies among nodes [297]. Random variables specify distributions of faults among statements in the software application and the distributions of the effects of these faults on control flow, dataflow and the output values. For example, one node may specify injections of arithmetic faults into certain program statements and the other node may specify that some output will change the sign of its values. The edge that connects these nodes determines the cause and effect with some probability. After a concrete PGM is instantiated for some application using a template, the task of production fault localisation is reduced to abductive reasoning on the PGM to navigate to nodes that represent random variables for injected faults starting with the nodes that represent distributions of changes in the output values. Not only does this abstraction enable speculative construction of the predictive fault model that is specific for every software application, but also this model enables stakeholders to obtain specific information on how a suggested fault results in an error state that propagates through the application to cause the failure. To the best of our knowledge, this abstraction has never been created and applied to automatic debugging of field failures. Indeed from a deeper perspective injecting a fault in the vicinity of the location of a latent bug leads to changes in the output values that are dual to the symptoms of the failure that results from activating this latent bug. In some cases, injecting faults led to inadvertent fixes of some latent bugs. The idea of using fault injection for automatically fixing faults is not new. Debroy and Wong proposed strategies for automatically fixing software faults by combining the processes of mutation and fault localization [298, 299]. In their evaluation using 19 programs with eight mutation operators, over 20% of the faults were fixed automatically. As part of investigating the positive effect that mutation has on fault localisation, Papadakis et. al. showed that faults are accidentally fixed by applying mutants [300–303]. Recent work of Moon, Kim and Yoo on MUSE also showed that not only fault localisation is more effective using mutants but also some of the faults are fixed automatically by applying mutants [304]. We hypothesize that by injecting faults in the vicinity of latent bugs and observing the changes in the output values, it is possible to create a predictive fault-localization framework.

CHAPTER 7

HIGH-LEVEL OVERVIEW OF α^5

A high-level overview of our approach, α^5 is shown in Figure 9. The programmer creates a software application, and `(1)` she releases its *uninstrumented* and performance-optimized deliverable to a customer. Of course, the programmer knows that statistically, there are many latent bugs in the application that have not been caught during testing. Therefore, `(2)` the programmer deploys this application at α^5 in parallel to and independently from its customer deployment. Once at α^5, the program statements are mutated and the mutants are run on a large number of sample inputs along with running the original unmutated program to introduce the baseline. The executions are monitored to construct a PGM that describes the causality between injected faults and output changes by instantiating a set of predefined logical formulae that capture correlations between statements and components. These correlations are weighted by observing their frequencies during the sample executions. The model is then reversed for abductive reasoning, so that potential faults can be inferred from symptoms.

At some point,`(3)` the customer submits a bug report to the programmer that describes symptoms of the failure, say, the value of the output `Bonus` is negative whereas it should be positive, and `(4)` the programmer enters this symptom into α^5 that performs the abductive reasoning, localizes potenial faults and `(5)` returns their locations to the programmer along with the navigation paths that show how these faults are activated into error states and how they propagates to the failure.

7.1 The Architecture And Workflow of α^5

The architecture and the workflow of α^5 are shown in Figure 10. When the coding and testing tasks are completed for a software application, it is moved to a cloud infrastructure where it is plugged into α^5. The input to α^5 is the configuration file that specifies the main class and the main method(s) of the application, the ranges of the values of their input parameters and configuration options and the output variables. To obtain an application-specific fault model, the Sensitivity Analyzer `(1)` injects different faults in the AUT and runs an instrumented modified version of the AUT using sampled input and configuration values on a separate cloud-based testbed independently of and in parallel to running the original version of the AUT with the same input/configuration settings. That is, **this step is done before or in parallel to deployment of the application at customers' sites** to speculatively determine the effects of possible faults.

Then, the Sensitivity Analyzer performs differential analysis between the original and faulty runs using collected runtime information to determine how different faults affect the control and data flows as well as the values of its outputs. A goal of the sensitivity analysis [305–307] is not in trying to localize a fault, but in determining the sensitivity of the application's behavior to different faults and in summarizing this behavior in a fault model. In doing so, we utilize similar ideas from DARWIN, an automatic debugging approach for programs that evolve from a stable version to a new version [308]. The Sensitivity Analyzer `(2)` outputs Generalized Ground Facts that state how injected faults affect the behavior of the AUT using template logic formula from the Knowledge Base. An example of a Generalized Ground Fact is a formula that states

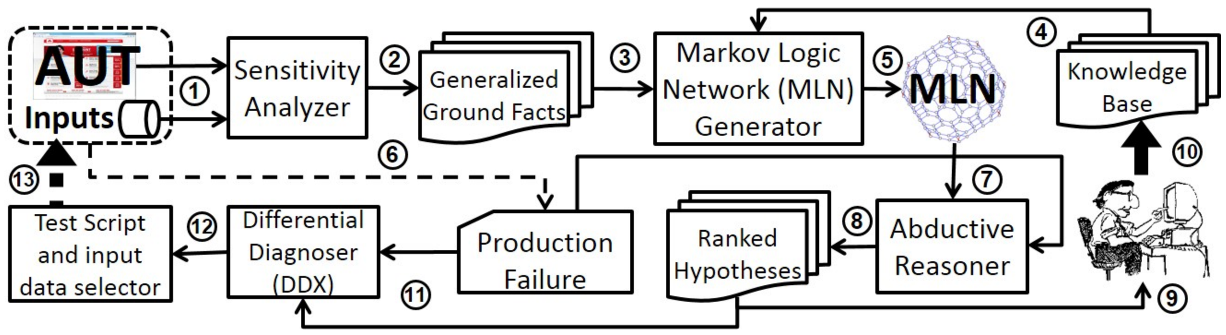


Figure 10. The architecture and workflow of α^5.

$I(x, 0, 10) \wedge M(s, t) \rightarrow O(y, -)$, i.e., for the range of input values for the variable $x \in [0..10]$ and the injected fault (i.e., producing a mutant) of the type, t into the statement, s, the value of the output variable, y changes to negative. While injecting faults, running the AUT and performing the subsequent differential analysis (see Section 11.1), α^5 creates and updates a PGM that is a fault model of this AUT. This process continues for some time; the more faults are explored the bigger and more sophisticated the PGM becomes. Essentially, the cloud infrastructure is utilized to process and store a large PGM and to localize production faults.

Our key idea is to express the extracted fault model using a PGM, specifically using a formal representation called *Markov Logic Network (MLN)*, which combines first-order logic and probabilistic graphical models. An MLN is a set of pairs, (F_i, w_i), where F_i is a formula in first-order logic and w_i is a real number that designates a weight for the corresponding formula [309, 310]. Depending on the frequency of instantiations of each clause for a given application with a set of input values, these clauses will be assigned different weights. Applying different faults, M to different statements, s will instantiate these clauses and create a ground MLN, a very large graph, whose nodes are instances of clauses (i.e., ground predicates) and the edge exists between a pair of nodes that contain predicates that appear together in some grounding of one of the corresponding formula, F_i. With MLN, the probability can be inferred that the value of some output will change (i.e., a possible failure) if some statements contain faults (see Section 7.2).

These Generalized Ground Facts (3) and the Knowledge Base (4) are the inputs to the MLN Generator that (5) outputs an MLN for the AUT. Once a production failure is observed (6), its symptoms are inputted (7) to the Abductive Reasoner along with the precomputed MLN. The Abductive Reasoner computes ranked hypotheses by navigating the MLN that describe how faults in different statements in the AUT can result in the observed failure. These hypotheses (9) are given to stakeholders who may use them (10) to update the Knowledge Base - user feedback is collected to improve fault localisation [311].

More importantly, Ranked Hypotheses (11) serve as the input to the Differential Diagnoser that guides the α^5 process to collect more information from the AUT into the MLN to reduce the number of hypotheses and pinpoint precise faults that cause the failure. That is, our other idea is to automate the process of *differential diagnosis* where the probability of different faults are weighted against each other by perturbing the system to collect additional evidence. As it often happens, fault localisation approaches often given a list of multiple possible faults that can account for a particular failure, and α^5 drills down further by injecting targeted faults and performing remedial actions close to the suggested fault locations to evaluate the effect of these actions on the symptoms of the failure. The Differential Diagnoser `(12)` tells the Test Script and Input Data Selector what inputs `(13)` to select for the AUT and what faults to inject to test the hypotheses. The process repeats and the MLN grows more precise and more powerful.

7.2 Modeling Faults With Markov Logic Networks

In this section, we describe how MLN is constructed automatically for application-specific fault modeling. First, we show a shortened list of the predicates and the logic formulae that constitute a template for the MLN. We used these formulae to obtain our preliminary results, and these formulae can be modified in order to improve the precision of fault localisation. Simply put, it is a tradeoff between efficiency and precision of fault localisation – more logic formulae lead to collecting more detailed information about the AUT, more complex and larger MLN and longer inference executions; however, the precision of fault localisation improves. Fewer logic formulae make the inference more efficient, but the precision of fault localisation suffers. Here are example of rules that describe the generic fault/propagation/failure model.

hasFault$(\mathtt{s}, \mathtt{m})\wedge$**StateDiffers**$(\mathtt{s}) \Rightarrow$**Infected**$(\mathtt{s})$**:** it is the infection rule. If the fault, $\mathtt{m}$ is injected into the statement, $\mathtt{s}$ and the state of the mutated application after executing $\mathtt{s}$ differs from the state of the original application, then the state is infected at the statement, $\mathtt{s}$.

Infected$(\mathtt{s})\wedge($**cfDepends**$(\mathtt{s}, \mathtt{t})\vee$**dfDepends**$(\mathtt{s}, \mathtt{t})) \Rightarrow$**Infected**$(\mathtt{t})$**:** it is the propagation rule. If the statement, $\mathtt{s}$ is infected and some other statement, $\mathtt{t}$ control and dataflow dependent on $\mathtt{s}$, then $\mathtt{t}$ is infected.

Once the template formulae are defined, instantiating them for a software application results in a grounded MLN with specific likelihood weights assigned to each logic formula. Quantifiers for these logic formulae are intentionally omitted, since weights will determine the probabilities of these logic formulae to occur. If a formula is never instantiated for specific values of parameters, then its weight will be zero, meaning that the formula is not grounded in the MLN. A key rule for assigning weights to logic formula is the frequency of its instantiations for a set of sampled parameters for the AUT. This process of weight assignment is similar to assigning probabilities in probabilistic program analyses [312, 313]. We already described the process of assigning weights using the illustrative example in Chapter 5. Using MLN enables stakeholders to create application-specific rules, e.g., Rule 4 in the illustrative example that links the timeouts to the outputs.

We selected MLNs to model faults in software applications for the following two reasons. First, expressing navigation and control dependencies requires first-order logic using which observations can be deduced and abduction can be applied to determine root causes (i.e., faults) of the effects (i.e., symptoms of field failures). However, given uncertainty with which background knowledge and evidence are obtained (e.g., an incomplete set of inputs which are used to obtain navigation and control dependencies), it is impossible to use first-order logic without assigning likelihoods to logic rules that allow the reasoning engine to use alternative explanations. Thus, representation is needed to handle uncertainty in addition to logic rules [314].

Second, Bayesian networks are a traditional approach; however, they are propositional and they do not handle relations among multiple entities. In addition, MLNs allow loops while Bayesian networks do not, which limits the utility of modeling software applications, although recent attempt is made to apply Bayesian network to dataflow analyses of probabilistic programs [315]. While many PGMs exist that handle different aspects of uncertainty and logic, only MLNs have full facilities needed for a solution to the fundamental problem of automated debugging that we address in this thesis.

7.3 Using Differential Diagnosis to Improve Fault Localisation

In medical sciences, *differential diagnosis (DDX)* is a set of procedures to distinguish a particular disease or condition from others that present similar symptoms [316]. More generally, DDX involves a combination of information elimination and acquisition to reduce the probabilities of candidate conditions to some infinitesimal values [49]. DDX uses the *hypothetico-deductive method* that works by formulating a hypothesis that can be falsified under specific conditions [317]. That is, if a failure can be explained by two or more different faults with certain probabilities, a hypothesis is formulated that by injecting a specific fault in the propagation path from one fault to the failure, the nature of the failure should change (e.g., the sign of the output value will be reversed). Further injecting new faults along the causal path from one of the original potential faults to the symptom, the change in the output values is observed to determine if they are described by the observed symptoms. If it does not happen, the hypothesis that suggests this fault is marked as potentially invalid. If this hypothesis is falsified eventually, then the probability of this fault is reduced and the probabilities of the other faults are increased. This is the essence of using DDX for fault localisation.

Suppose that some X faults have a common part of the failure propagation path and some Y faults have a different common part of the failure propagation path. Running all tests for all faults will result in $\mathsf{X} + \mathsf{Y}$ runs. Instead we inject a fault in one common path and see if this fault changes the output by running a test only for one of Xs or one of Ys. In α^5, we use DDX to increase the precision of fault localisation by injecting faults along specific propagation paths to add more grounded rules to the MLN. We will study empirically the effect of selecting different faults and how well they help to differentiate among failure hypotheses.

DDX is organically integrated into the workflow of α^5, since re-running its components offline does not require instrumenting the program or performing any experiments at the customer deployment site. It is generally acknowledged that the computation time is much cheaper than the human cost of manual debugging. Moreover, fault injection could be done by modifying the program bytecode and avoiding recompilation of the source code, but regardless, DDX can save many man-months of expensive debugging and fault localization effort for a single failure. Most importantly, no customer involvement is required in the DDX process.

CHAPTER 8

α^5 TECHNICAL WORKFLOW

In this chapter we will present in depth the architecture of α^5 as illustrated in Figure 10. In order to produce ground facts the AUT runs through different main phases:

- Mutation: this phase produces mutants that will stimulate different behaviors of the program.
- Instrumentation: this phase instruments the code to produce statement to be analyzed during the database generation.
- Execution: this phase executes the instrumented code to collect dynamic information about the AUT.
- Differential Execution Analysis: this phase exploits differential analysis on the data produced from the above phases to instantiate ground predicates used in the MLN
- MLN generation: this phase uses the predicates above produced to create a new MLN, used to apply inference and get the final result.

Each one of this phases will be further described in their specific section.

8.1 Phase 1: Mutation

In this section we are going to describe the first phase of the underlying process for α^5: the mutation phase.

This phase heavily relies on a component written by Prof. Jeff Offutt [318]: MuJava [319]. This is not a necessary dependency, but to the best of our knowledge this is the most complete and efficient tool available at the time of writing.

MuJava operations consist in parsing the source code of the AUT and apply mutants while evaluating the tree. There two different kind of mutants: class-level and method-level mutants. For each one of this categories several operators exists.

Class level operators apply mutants on features typical of Object Oriented languages (e.g.: changing a field from public to private). A brief description of them is given in [320].

Method level operators instead apply mutants on expressions, mostly arithmetic. A brief description of them is given in [321].

8.1.1 Taming complexity

An issue α^5 needs to face is how to achieve scalability. One first measure to lower the complexity of the application is to control the number of mutants. This can be achieved in two indisputable ways:

1. Reducing the number of operators to apply.

2. Reducing the number of locations where to apply the operators.

Each of these solutions was implemented with using a random strategy where the probabilities can be controlled. It's worth notice that the effectiveness of a mutants operator may greatly depend on the nature of the AUT and therefore is extremely difficult to note beforehand which operator will stimulate the behavior of application. However it's not important that the mutant operator exactly matches the kind of bug that the AUT is exhibiting. As long as the mutant is able to trigger the right deviation that will lead to a successful sensitivity analysis, it's not so relevant what class the mutant belongs to.

Solution number one can be improved by assigning probabilities to which mutant to apply. How to assign them can be a separated research question, but can be guided by the result in Section 10, since some operators are prone to create more mutants than others.

On the other hand, solution number two can be improved by running the original non-mutated version of the AUT and recording the location of code actually executed. At this point the strategy for selecting in which locations apply operators is straightforward: if the location was not executed in the original run, it is discarded. To understand why this claim is true, consider the following motivating example in Figure 11 where the bold circles represent the original path executed, while the dotted circle represent the point of insertion of the mutant. If the mutant is applied in a location that is never reached during execution, it cannot possibly alter the control flow and therefore it will never executed, thus not producing any changes.

8.1.2 Technical Issues

α^5 relies on MuJava, and MuJava in turn relies on OpenJava (OJ) [322], an open source parser for Java code.

Unfortunately the OJ project has not being updated for over a decade at the time of writing, so it only supports Java Development Kit (JDK) up to version 6. The project is now being extended by Professor Offutt in order to being able to analyze even more recent applications.

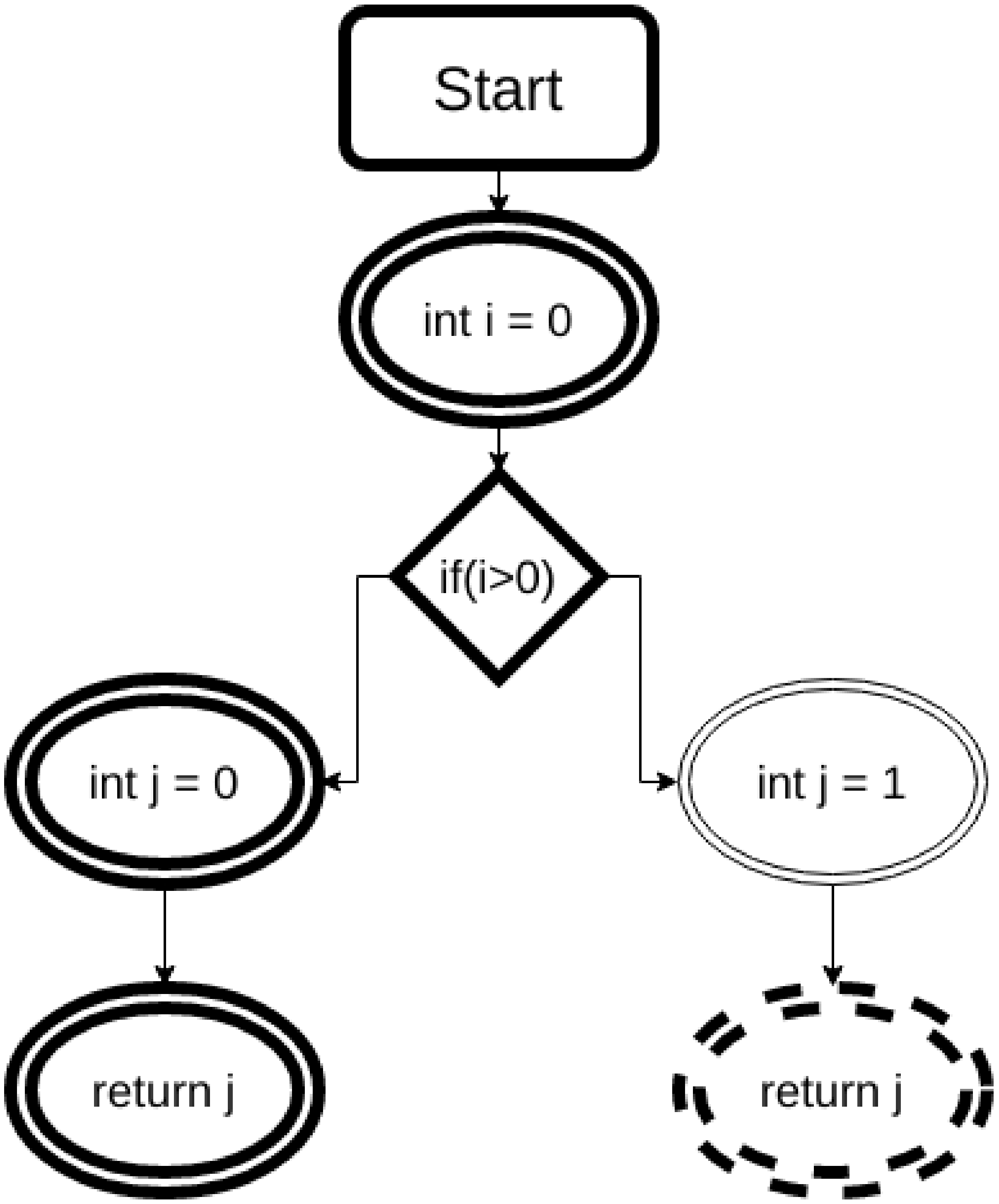


Figure 11. A simple control flow diagram.

This has lead some problem in the selection of AUTs since some of them exploits features unavailable in the previous JDKs. To read more about this please refer to Chapter 9.

Moreover we were able to find some problems in the grammar used and after identifying them we collaborated with the team of Professor Offutt (in particular with his PhD student Lin Deng)to fix them.

A first problem arose with a specific syntax of *for* statements. The following is an example of statements not correctly parsed:

for (m =0; m¡4; m++)

while instead OJ was able to correctly parse *for* loops where the induction variable is defined internally.

for (int m =0; m¡4; m++)

The solution to this problem was to revise the grammar used by OpenJava.

A second problem derive from a specific array initialization of the following kind:

String[] sentences = new String[] ;

In this case the problem was resolved by changing the syntax to the following one, accepted by OJ:

String sentences[] = ;

A third syntax problem was raised by shift operators. From the Oracle documentation [323]:

"The signed left shift operator "¡¡" shifts a bit pattern to the left, and the signed right shift operator "¿¿" shifts a bit pattern to the right. The bit pattern is given by the left-hand operand, and the number of positions to shift by the right-hand operand. The unsigned right shift operator "¿¿¿" shifts a zero into the leftmost position, while the leftmost position after "¿¿" depends on sign extension."

The following are examples of code incorrectly parsed by OJ.

- Original code:

  ```
  if (( (i>>j) & 1) !=0)
  ```

  Parsed as:

  ```
  if ((i > j & 1) != 0)
  ```

- Original code:

  ```
  h ^= (h >>> 41) ^ (h >>> 20);
  ```

  Parsed as:

  ```
  h ^= h > 41 ^ h > 20;
  ```

- Original code:

  ```
  h ^= (h >>> 14) ^ (h >>> 7);
  ```

  Parsed as:

  ```
  h ^= h > 14 ^ h > 7;
  ```

In this case the solution was again to change the syntax of the instructions. For the first example i>>j was changed into i>>=j , then the if-statement was modified as

```
if((i & 1) !=0).
```

8.2 Phase 2 & 3: Instrumentation & Execution

In this section we are going to describe the second and third phases of the underlying process for α^5: the instrumentation and execution phases.

As the mutation phase the instrumentation stages is very delicate since determining what kind of statement to instrument involves a current tuning in the trade-off between precision of the predicates and speed of the procedure.

In order to instrument a program two main strategy can be pursuit:

1. Monitor field of interest using reflections from the bytecode (after compiling)
2. Insert additional statements in the source code (before compiling)

Solution one was implemented by Saurabh Dingolia, a former student of Professor Mark Grechanik, using the Java Reflection API and JavaAssist [324]. The usage of Reflection allows the access to classes, methods, fields and constructors. It was exploited for the following instrumentations:

- Data Collection Instrumentor. Instruments the input byte-code based upon a configuration file provided by the user to collect the values of the output variables at specific execution points (entry or exit of any method).
- Return Value Instrumentor. Instruments all the methods with non-void return types of the input program to collect their return values for each execution of that method.

- Trace File Instrumentor. Instruments the main method of the input program to open a file stream to the trace file in order to write the execution trace and values of output variables to the trace file. Also, closes the file stream at the end of main method.

Unfortunately the usage of Reflections is not enough for α^5, since it cannot capture control flow, data flow and value of variables inside methods. Therefore solution two needs to be implemented in conjunction with solution one. In order to obtain this result, OJ is exploited again (as in MuJava) to build the parse tree of the application. Three additional kind of instrumentation can be applied in this way:

- Statement Identifier Instrumentor. Assigns a unique identifier to each statement in the source program. The identifier is the fully qualified name of the method containing that statement followed by a unique number. The statement Ids thus assigned produce the execution trace when the instrumented code is executed. Also, the run-time execution trace is analyzed for producing control dependencies among statements.
- Expression Value Instrumentor. Creates a unique id for the value of interests. The selected expression value is then assigned to the new generated id and printed out in the execution trace.
- Data Dependency Instrumentor. Prints on the data dependencies between variables. To know more about how they are calculated refer to Section 8.2.1

8.2.1 Taming complexity

Instrumentation and execution phases can be very time consuming. Fortunately a huge improvement can be achieved using parallelization given the independence of each mutant to instrument and execute. Assuming to have enough resources, a full parallelization of the process can indeed improve the time performances from days to few hours. To better understand the impact of parallelization please refer to results in Section 10. To meet the necessity of costumers, α^5 users can specify in the property file the desired degree of parallelization.

Another speed-up can be achieved from sampling inputs during the execution phase. The strategy currently implemented selects random values from the input range provided, but is still possible for the user to select specific values. A potential improvement to this strategy can be to select inputs using Combinatorial Interaction Testing (CIT), " a black box sampling technique derived from the statistical field of design of experiments" [325].

As stated before tuning the complexity of instrumentation is a trade-off between time and precision of the results. Since we want α^5 to be as flexible as possible for the most wide range of applications, we included two properties in the properties file.

The first property, *expressionLimit* allows the user to properly tune the length of the expressions for which apply the Expression Value Instrumentor. For example if for any application-dependent only the value of expression with more than 4 operands need to be recorded, it is possible to set the value of the property to 4. If the AUT needs to be instrumented in the least amount of time possible, is achievable by setting the property value to zero, resulting in a less precise but faster inference process.

The second property, *ddexpr* allows the user to enable/disable the Data Dependency Instrumentor. Again, disabling it will result in a less precise but faster process. However the Data Dependency Instrumentor exploits a clever algorithm to extract dependencies. The way compiler calculates data dependencies is generally expensive from the computation point of view, it requires to calculate Def/Use set by the help of a Symbol Table, where all the reference to variables are registered with their relative scope. This leads to additional overhead both in space (an hash table needs to be maintained) and both in time (different scopes needs to be taken in consideration). α^5 can overcome this difficulties since the aim of the process is to build a probabilistic graph, making possible to lose some certainty degree for a faster execution. This is reflected in the way dependencies are extracted: instead of keeping track of different scopes, dependencies are calculated by the use of the memory address of the variables. To make this more clear let's consider an example:

int i = 0; //line1 int j = i; //line2

Assuming that the memory address of variable i is \$addressI and the memory address of variable j is \$addressJ α^5 will produce the following set of predicates:

Def(line1, \$addressI) Use(line2, \$addressI) Def(line2, \$addressJ)

This predicates can be post processed after to easily infer that line2 has a data dependency on line1. The address of the variables can be obtained using the Unsafe library by Peter Lawrey [326]. There is no guarantee that the address will not change due to garbage collection, but the probabilistic graph will give smaller weights to dependencies appearing only sporadically.

8.2.2 Technical Issues

Dealing with complicated processes like parallel parsing can result in several subtle problems.

A first technical problem arises during execution phase, where mutants can exhibit different behavior, one of which can be infinite loops. For this reason we implemented a timeout in the AUT configuration, adjustable depending on application specific needs. However this generated not so evident bugs, one of which was truncated trace files that caused crashes in the system. After careful debugging all the deriving bugs were resolved.

A second technical problem was due to the use of an undocumented and deprecated [327] library: *com.sun.tools.javac.Main* used to compile mutants. This problem arose in all the phases of mutation and instrumentation. The *compile* method used indeed was not able to include external libraries and Java Archive (JAR) files in the compilation classpath, resulting in several compiling errors for AUTs. The solution to this problem was to switch to an updated library: the *(javax.tools)* package.

A third important problem was linked to the use of OJ to build the parse tree. As stated in Section 8.1.2 OJ is an old library, supporting up to JDK version 6. To overcome this limit, we try to switch to another, updated tree parser: the *org.eclipse.jdt.core* library [328] [329]. This library is indeed part of the Eclipse Java development tools (JDT) which contains the core functions used by Eclipse to build the Abstract Syntax Tree (AST) of the application [330]. At the end this resulted in an integration problem. Part of the system was developed using OJ and in particular the Statement Identifier Instrumentor described in Section 8.2, responsible for assigning unique IDs to statements. The way this Instrumentor works, relies on the *evaluateDown()* method present in OJ. The Instrumentor written using the JDT used a different *evaluateDown()* method, that resulted in misalignment of statement identifiers. Since statement IDs uniquely identify locations during the construction of MLN, this problem was not acceptable for our system and resulted in the rewriting of the parser using OJ.

8.3 Phase 4 & 5: Differential Execution Analysis & Markov Logic Network Generation

In this section we are going to describe the fourth and fifth phases of the underlying process for α^5: the Differential Execution Analysis and Markov Logic Network Generation phases.

The aim of the Analysis phase is to collect the execution traces produced in the previous stages and compare them in order to obtain the differences in control flow, data flow and values from the original execution. In particular one the control flow start to diverge from the original path, α^5 will stop producing data flow statements.

After statements are collected, they became inputs for *Alchemy* [331]. Alchemy is an open source software package able to perform statistic inference through several algorithms for statistical relational learning and probabilistic logic inference, which rely on the Markov logic representation.

The process is represented in Figure 12.

After the execution of the AUTs with different inputs, *Behaviors* are collected, for example:

HasFault(line1, mutant5) Reached (line4) DataDependency (line4,line5) ControlFlow (line4,line5) Failed (line1, symptom1)

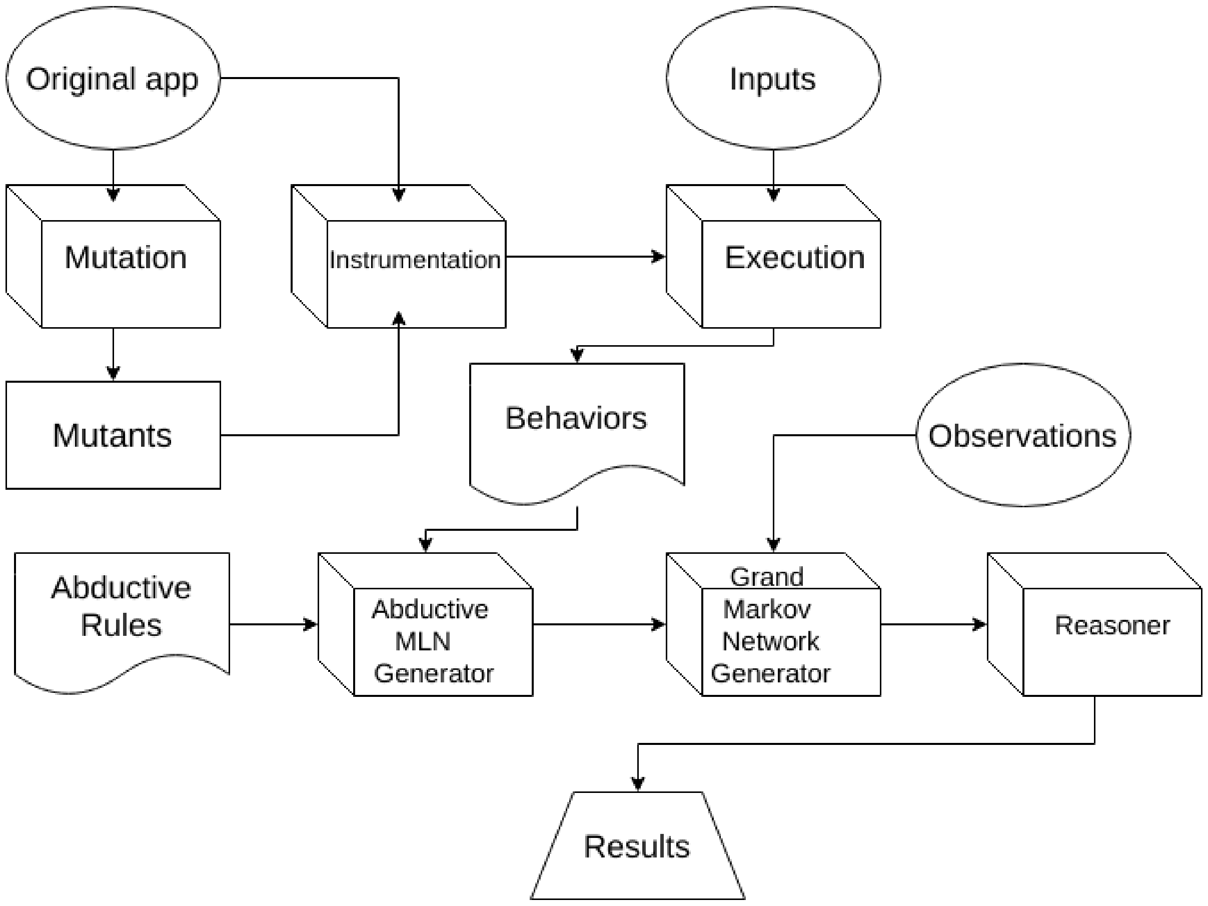


Figure 12. α^5 process visualized.

This behaviors, are processed by Alchemy in the Abductive MLN generator together with abductive rules (e.g.: DataDependency (a,b) and Infected (a) ->Infected(b)).

An important consideration to remark is that the behaviors and the rules are processed altogether in the .mln file, from which Alchemy will produce and store a MLN. This process needs to be executed **only one time**.

After the user submits the *Observations* inference can be applied on the previously generated MLN through the provided rules.

However this process is not so simple, since several features needs to consider for this procedure, including:

- Feature 1. what needs to be included as rules for inference $\in$ {only formulas, formulas+pre-collected behavior};
- Feature 2. whether weights are assigned for rules $\in$ {Yes, No};
  - Feature 2.1. if feature 2 = Yes, whether we train the weights for rules are trained or manually assigned $\in$ {Yes, No};
    - Feature 2.1.1. if we train the rules, what evidence should be used for training $\in$ {whole set of pre-collected behavior, only the subset related to specific failure(s)};
- Feature 3. what are considered as evidence for inference $\in$ {only field behavior, field behavior + pre-collected behavior};

  - Feature 3.1. if we include pre-collected behavior as part of the evidence, whether we use the whole set, or only the subset related to specific failures(s) $\in$ {whole set, subset}.

- Feature 4. what type of rules to use $\in$ {deductive, abductive}. In α^5 we decided to use deductive rules.

Since each feature above is binary, 48 possible decisions are possible (24 after fixing feature 4, since it's indipendent from all the others). However some of them are invalid, and some of them are identical. Indeed, as shown in a simplified decision tree in Figure 13 both this cases occur:

- Variation 1 (V1) and 3 are invalid, because we are not using pre-collected behaviors anywhere. This way, when we have field behaviors rules (either with weights or not), what we can get are only the query predicates grounded on the statements that are used in field behaviors. For example, if we have Failed(1), Failed(2), DataDependency(2,3) as field behaviors, the only possible predicates obtainable when we query for hasFault are hasFault(1), hasFault(2), and hasFault(3).

- V6 and V2 are identical variances: in V6, we provide formulas and field behavior, so it is redundant to include field behavior again as the evidence for inference, which essentially make it the same case as V2. Same idea applied to V8 and V4.

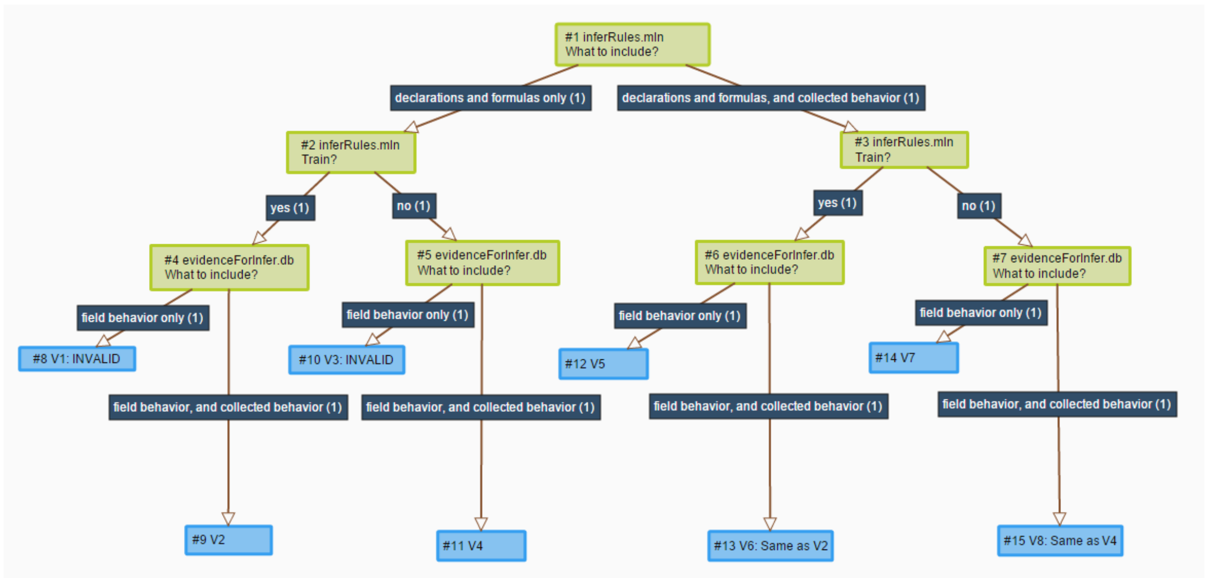


Figure 13. Simplified decision tree for features to select during the inference process.

CHAPTER 9

VALIDATION

In this chapter we will present our strategy to validate α^5. In particular we will cover the following steps:

- Validity check for Java syntax parsing
- Selection of AUTs

9.1 Java Syntax Checking

In order to validate our framework experiments needs to be run to check if all the Java syntactic constructions are correctly handled. For this reason we build different configuration files , each one of which uses a different keyword. The main class of these configurations are reported here, omitting the test class for brevity. The result of this test reveled that the keywords *native* and *strictfp* are not supported by OJ and resulted in the fix of the bugs described in Section 8.1.2.

9.2 Subject applications selection

Selecting AUTs is not an easy task and it can be considered an entire project by himself. Indeed the master project of Phani Theja Swarup Vempalli, a former Graduated UIC Master student of Dr. Mark Grechanik, was dedicated to automatically extract the source code for Java applications on GitHub, in order to be used as subjects for α^5. However extraction of application is not sufficient, since additional filter must be imposed for the app to be an actual subject. In particular the following requirements must be imposed:

1. The application must have bug repository from where the problem and the fix can be identified. This requirement is necessary in order to be able to measure the actual effectiveness of α^5 since we need to measure the 'distance' from the proposed fix location to the actual fix location.

2. The application must be supported by OJ, so it should compile with JDK 6.

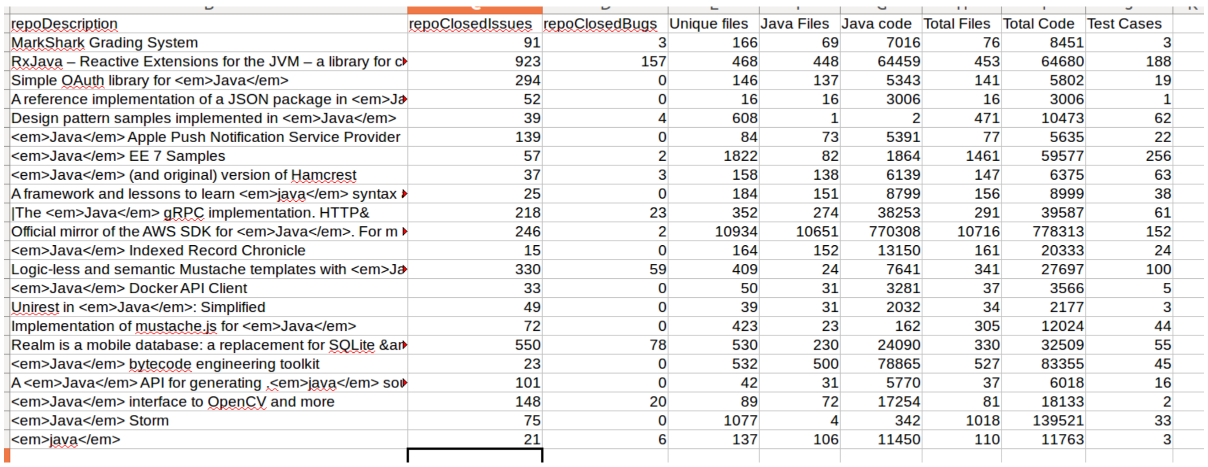
3. The application must not contain GUI or networking distributed features. This requirement is not essential. Running GUI apps with α^5 means that the test driver must handle GUI interfaces, therefore it requires a design/development effort to incorporate the driver (e.g., Selenium [333]) into α^5.

4. The application must be written mostly in Java.

5. The application must contain a test suite. This is necessary in order to build the main test class. Moreover α^5 is targeted to software applications already deployed, so the existence of a test suite is assumed.

6. The application must exhibit a clear input/output format, since it needs to be easy to monitor. If complex output is involved (e.g.: XML files) specifying symptoms for the AUT became extremely difficult.

The code is written in Python and uses the *NIX application *cloc* [334] to check if the subject is written in Java. In particular requirements 1 and 2 filtered out most of applications. Table XI show different parameters for several repositories analyzed, where column 'Issues' refers to all the GitHub bugtracker issues (including configuration problems, features extension and so on), while the column 'Bugs' refers to issue marked as bug by the developer. Most of the code not written in Java was part of EXtensible Markup Language (XML), YAML Ain't Markup Language (YAML), Bash or Maven files. The description or each repository can be found in Appendix A



repoDescription	repoClosedIssues	repoClosedBugs	Unique files	Java Files	Java code	Total Files	Total Code	Test Cases
MarkShark Grading System	91	3	166	69	7016	76	8451	3
RxJava – Reactive Extensions for the JVM – a library for c	923	157	468	448	64459	453	64680	188
Simple OAuth library for <em>Java</em>	294	0	146	137	5343	141	5802	19
A reference implementation of a JSON package in <em>Ja	52	0	16	16	3006	16	3006	1
Design pattern samples implemented in <em>Java</em>	39	4	608	1	2	471	10473	62
<em>Java</em> Apple Push Notification Service Provider	139	0	84	73	5391	77	5635	22
<em>Java</em> EE 7 Samples	57	2	1822	82	1864	1461	59577	256
<em>Java</em> (and original) version of Hamcrest	37	3	158	138	6139	147	6375	63
A framework and lessons to learn <em>java</em> syntax	25	0	184	151	8799	156	8999	38
|The <em>Java</em> gRPC implementation. HTTP&	218	23	352	274	38253	291	39587	61
Official mirror of the AWS SDK for <em>Java</em>. For m	246	2	10934	10651	770308	10716	778313	152
<em>Java</em> Indexed Record Chronicle	15	0	164	152	13150	161	20333	24
Logic-less and semantic Mustache templates with <em>Ja	330	59	409	24	7641	341	27697	100
<em>Java</em> Docker API Client	33	0	50	31	3281	37	3566	5
Unirest in <em>Java</em>: Simplified	49	0	39	31	2032	34	2177	3
Implementation of mustache.js for <em>Java</em>	72	0	423	23	162	305	12024	44
Realm is a mobile database: a replacement for SQLite &ar	550	78	530	230	24090	330	32509	55
<em>Java</em> bytecode engineering toolkit	23	0	532	500	78865	527	83355	45
A <em>Java</em> API for generating .<em>java</em> so	101	0	42	31	5770	37	6018	16
<em>Java</em> interface to OpenCV and more	148	20	89	72	17254	81	18133	2
<em>Java</em> Storm	75	0	1077	4	342	1018	139521	33
<em>java</em>	21	6	137	106	11450	110	11763	3

Figure 14. A result screen from the script to extract repositories

TABLE XI

REPOSITORIES STATISTICS

Repository	Issues	Bugs	Unique files	Java files	Java code	Total files	Total code	Test cases
/ReactiveX/RxJava	926	157	468	448	64459	453	64680	188
/hamcrest/JavaHamcrest	37	3	158	138	6139	147	6375	63
/Esri/geometry-api-java	53	25	319	306	73231	309	73496	46
/fommil/matrix-toolkits-java	54	25	240	226	14682	235	15124	104
/googlegenomics/api-client-java	42	2	76	70	3103	72	3324	34
/FasterXML/java-classmate	17	0	77	68	6690	69	6856	35
/laforge49/JActor	42	0	245	240	6716	243	6984	50
/functionaljava/functionaljava	58	29	350	278	28544	317	29998	27
/thelinmichael/spotify-web-api-java	16	0	147	108	9782	111	9928	33
/mikiobraun/jblas	35	0	130	74	11839	122	20475	16
/vkostyukov/la4j	140	23	122	115	13045	117	13135	29
/nmcl/JavaSim	28	0	69	61	3608	62	3638	20

CHAPTER 10

RESULTS

In this chapter we will describe the experiments executed on the α^5 framework. In particular two set of experiment will be presented:

- Inoculated experiments: we auto-generated Java applications and run them through α^5 in order to get performance estimate.
- 'Real world' experiments: we processed the applications described in Section 9.2 and analyzed the results produced.

10.1 Performance experiments

To analyze performance measures of α^5 we made use of a former project of Professor Mark Grechanik, a program able to automatically generate Java programs [335]. In order to evaluate different real world size, experiments were conducted with auto generated programs (AGPs) with 1k, 5k, 10k, 50k and 100k lines of code. All the experiments were evaluated on a Dell server R720 with two CPUs Intel Xeon E5-2609 2.40GHz,10M Cache, 6.4GT/s QPI, with 32GB of RAM.

Table XII shows the results for the experiments on each one of the applications considered.

- **Mutation phase:** result for the mutation phase are shown in Figure 15. The fit is almost perfectly linear, just the 100K AGP deviates a little from the fitting curve. The least-squares best fit curve it's indeed not linear, but a quadratic curve: $6.4406110^{-}9x^2 + 0.000274876x + 2.0717$

- **Instrumentation phase:** result for the instrumentation phase are shown in Figure 16. The least-squares best fit curve it's a cubic: $1.3510^{-}8x^2 - 0.000367465x + 6.0941$. However discarding the 100K AGP (since it may have degraded the performance on a single machine), a logaritmich fit achieves a R^2 measure of 0.95/1, resulting even better then a linear fit. This is visually represented in Figure 17, where the 100K AGP performances have been discarded.

- **Execution phase:** The execution phase has been executed with 100 threads in parallel as discussed in 10.1.2. A timeout of 10 seconds as been set for all the applications. The least-squares best fit curve almost perfectly fits the data and it's a quadratic: $5.2113210^{-}9x^2 + 0.0000275524x + 0.109555$. The second grade term start to be greater than 10 hour when an application reaches the size of 50KLOC.

- **Differential Execution Analysis phase:** result for the Differential Execution Analysis phase are shown in Figure 19 where the values for the 50K and 100K AGP were extrapolated from the fitting model. The least-squares best fit curve it's linear: $0.0000717459x - 0.0473115$.

- **Inference phase:** result for the Differential Execution Analysis phase are shown in Figure 20 where the values for the 50K and 100K AGP were extrapolated from the fitting model. The least-squares best fit curve it's linear: $0.0000700738x + 0.0116066$.

Figure 21 and Figure 22 summarize the total performances.

TABLE XII

EXPERIMENTS RESULT FOR AGPS

	Time (hours)				
AUT size (LOC)	1,000	5,000	10,000	50,000	100,000
Mutation time	1.091	3.688	7.069	31.374	94.085
Instrumentation time	0.842	6.082	8.603	19.642	104.753
Execution time	0.086	0.488	0.851	14.517	54.978*
Differential Execution Analysis time	0.033	0.296	0.677	3.54*	7.127*
Alchemy time	0.08	0.365	0.711	3.515*	7.019*
Total	2.132	10.919	17.911	72.587	267.962
Total (days)	0.089	0.455	0.746	3.024	11.165

* Results estimated from the 1K,5K,10KLOC performances.

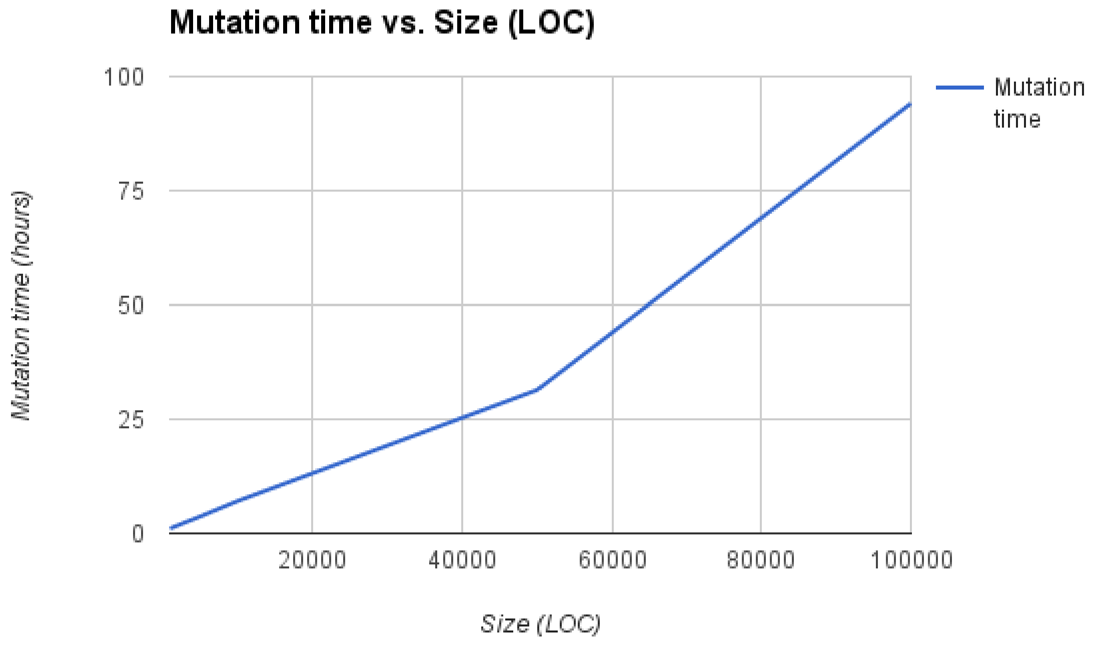


Figure 15. Mutation phase results

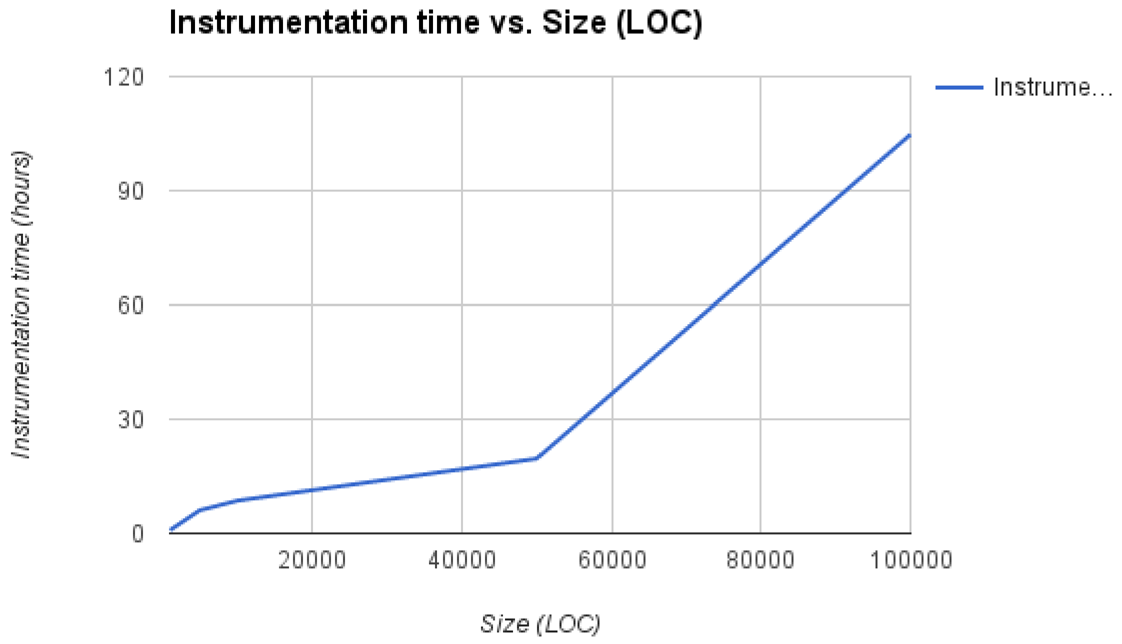


Figure 16. Instrumentation phase results

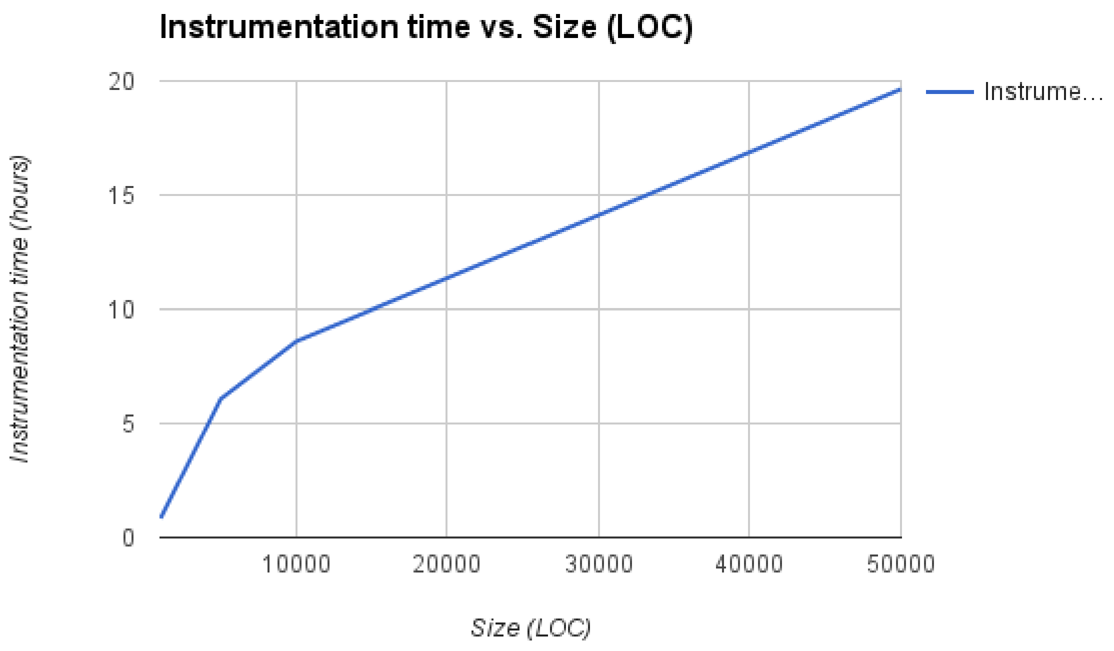


Figure 17. Inference phase results discarding the 100K AGP

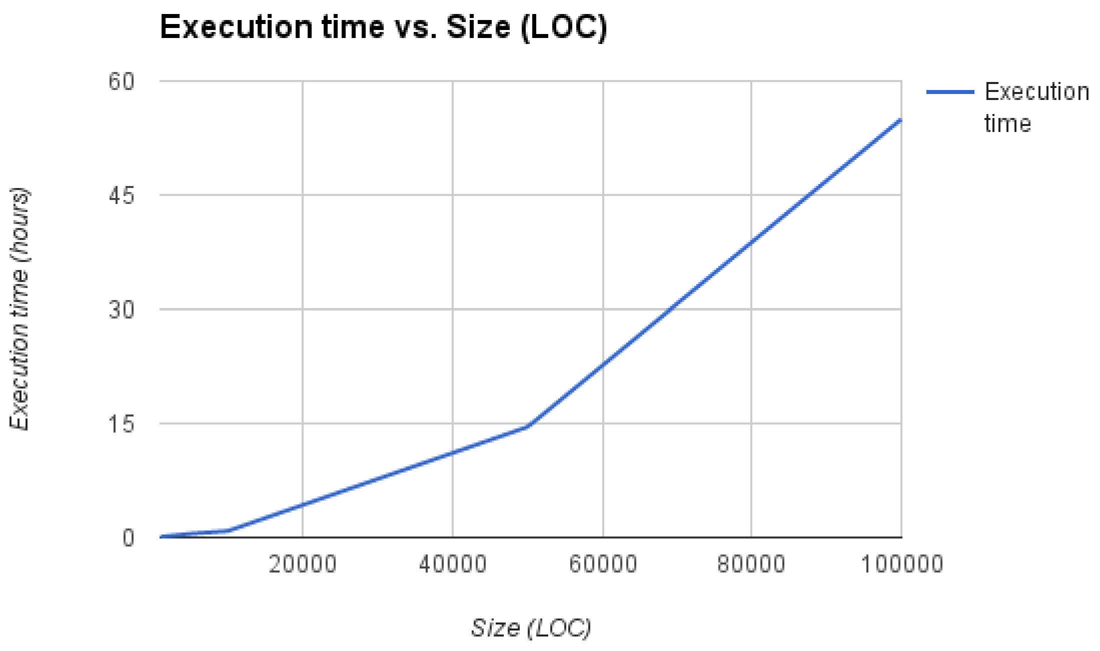


Figure 18. Execution phase results

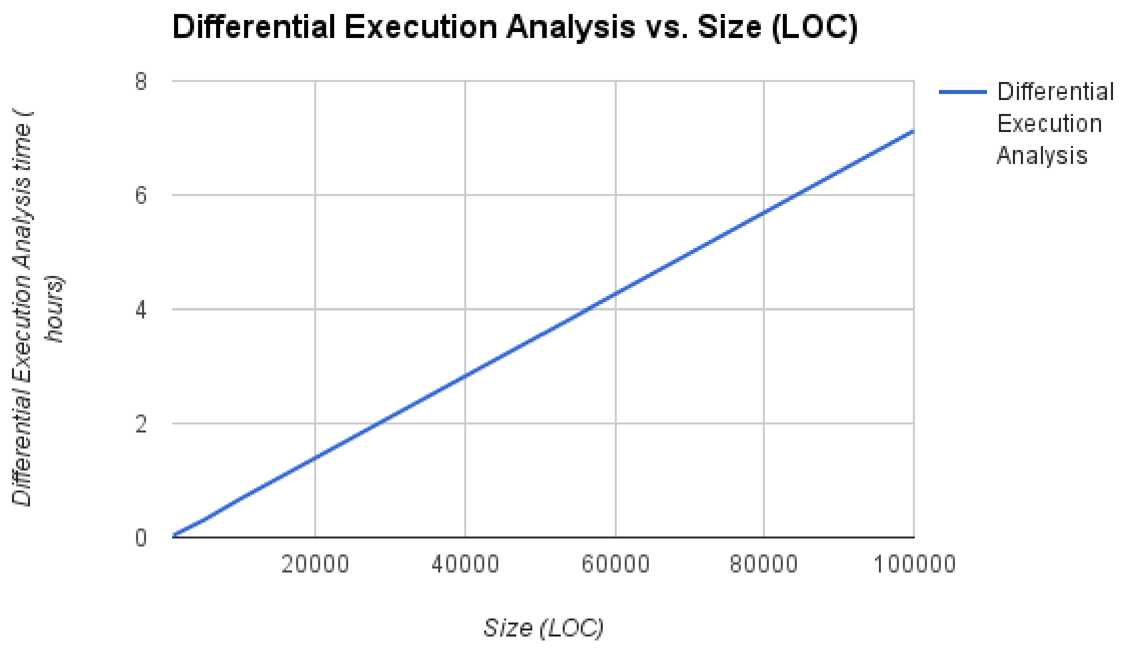


Figure 19. Differential Execution Analysis phase results

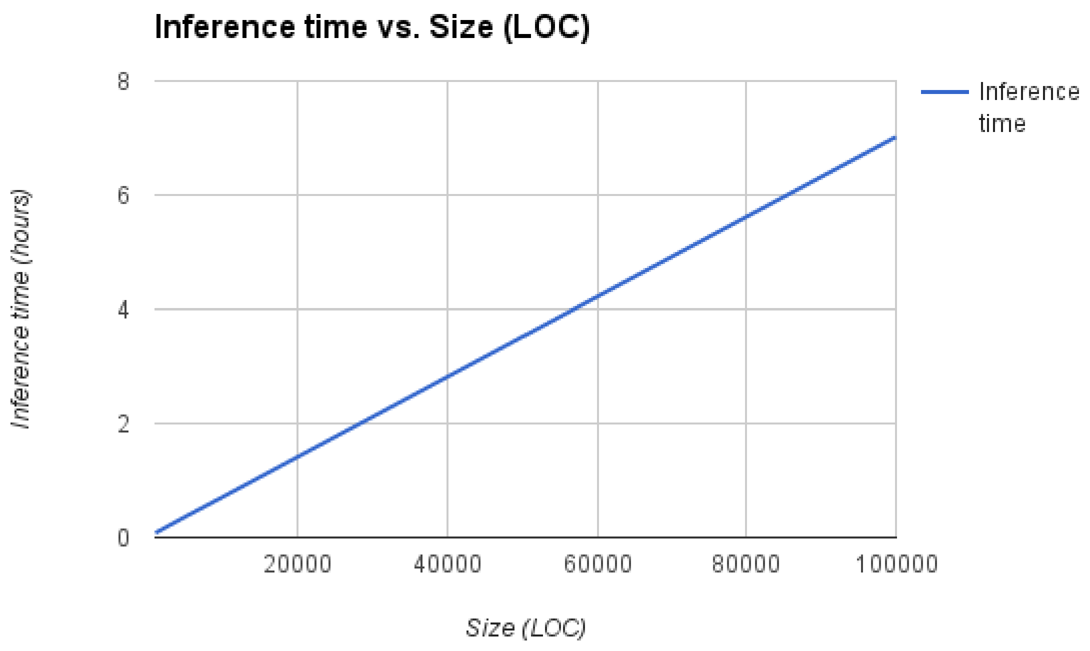


Figure 20. Inference phase results

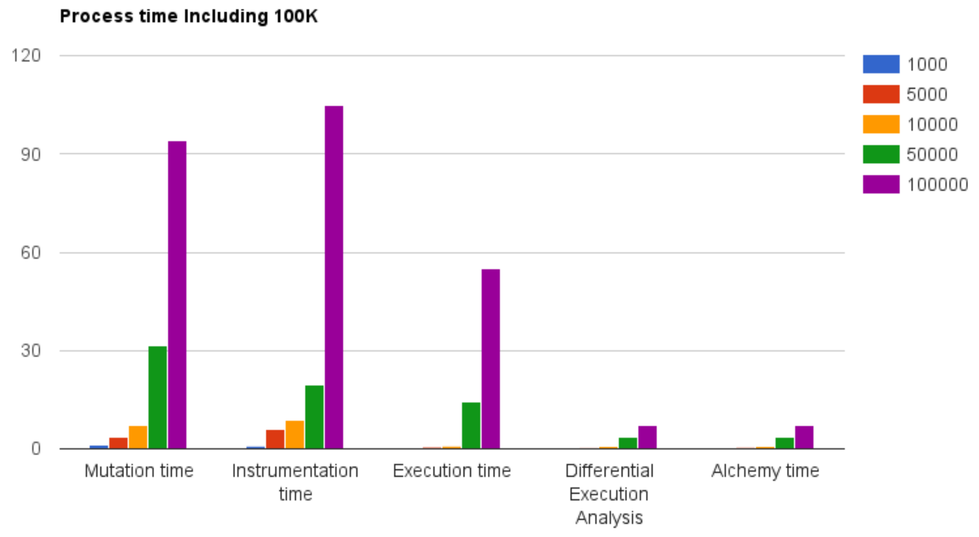


Figure 21. Time performances of the whole process (including the 100K AGP).

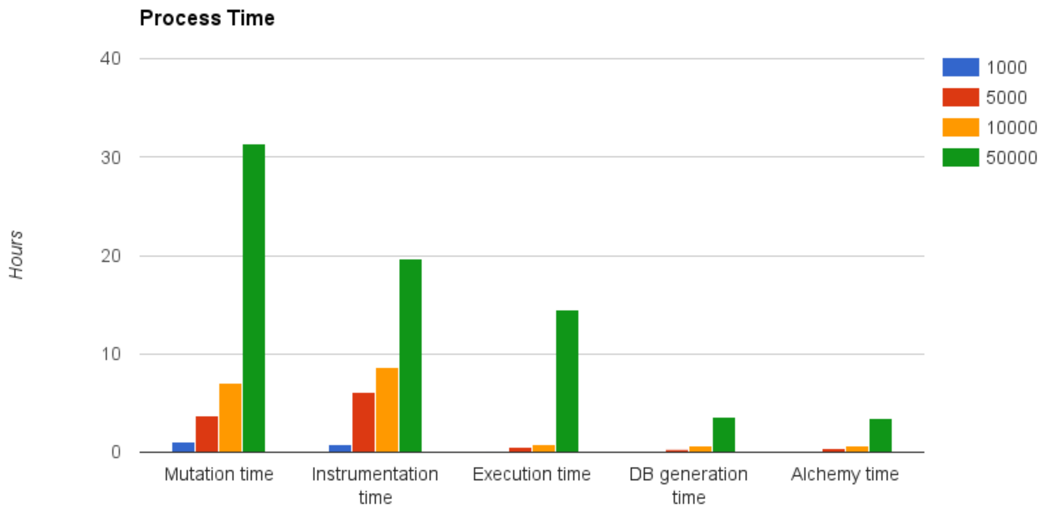


Figure 22. Time performances of the whole process (excluding the 100K AGP).

10.1.1 Mutants number

Table XIV and Table XV report respectively the number of class mutants and traditional mutants produced during the mutation phase of α^5. This numbers are fundamental for the whole process, since all the phases relies on them: instrumentation phase needs to instrument each one of this mutants, execution phase executes each one of them and the Differential Execution Analysis phase produces a database for each one mutant produced.

Therefore an extremely challenging trade-off is present: from one side we desire to keep the number of mutants as low as possible to increase time performances, from the other side we also may want as much mutants as possible to collect a variety of symptoms.

Table XIII summarize the total number of mutants for each application. A linear model fits well the data ($14.2051x + 6602.83$ produces an R^2 of 0.98) as shown in Figure 23. However, is worth noticing that the structure of AGPs is slightly different from 'real worlds' program, since it's generated from a probabilistic grammar, therefore a lower number of mutants may be expected in future experiments.

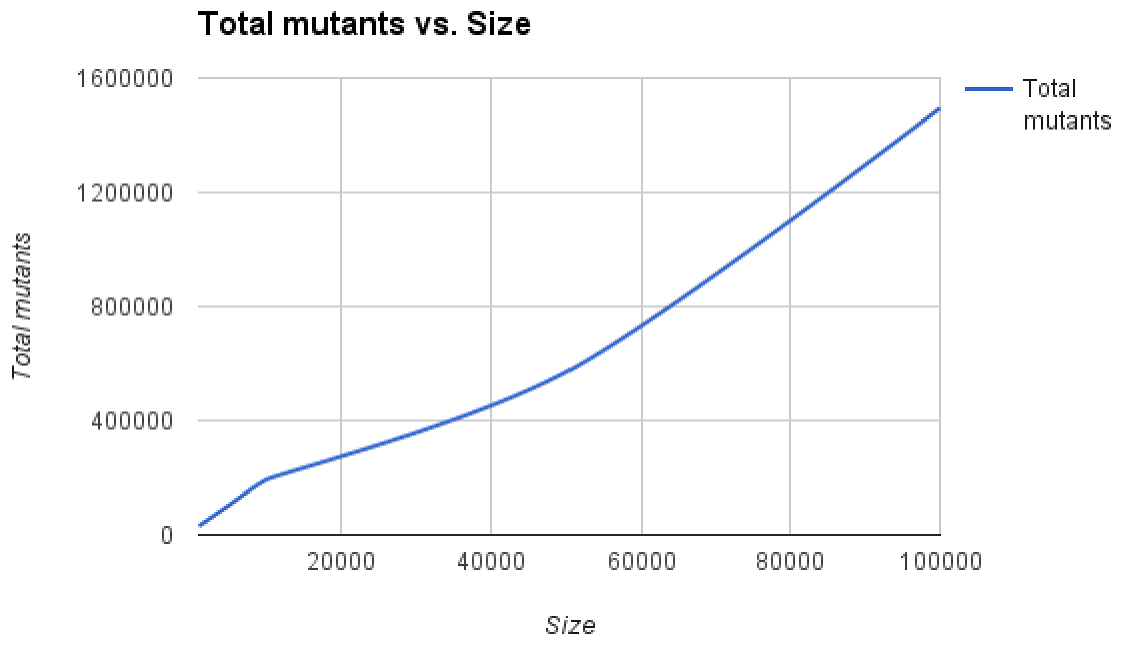


Figure 23. Total number of mutants produced

TABLE XIII

MUTANTS PRODUCED FOR AGPS

Size (LOC)	1,000	5,000	10,000	50,000	100,000
Total mutants	30,927	103,085	192,690	569,187	1,495,167
Class mutants	221	234	830	854	1,618
Traditional mutants	30,706	102,851	191,860	568,333	1,493,549

TABLE XIV

CLASS MUTANTS PRODUCED FOR DIFFERENT AGPS

Size	**1,000**	**5,000**	**10,000**	**50,000**	**100,000**
Class mutants	**221**	**234**	**830**	**854**	**1,618**
IHI	10	9	24	15	25
IHD	0	0	0	4	0
IOD	19	24	45	35	44
IOP	0	0	0	0	0
IOR	11	16	19	15	22
ISI	5	10	2	7	10
ISD	0	0	0	0	0
IPC	0	0	0	0	0
PNC	142	95	553	580	1325
PMD	0	0	27	53	19
PPD	0	0	8	20	18
PCI	0	2	0	2	0
PCC	8	6	18	6	55
PCD	0	0	0	0	0
PRV	0	0	0	0	0
OMR	0	0	0	0	0
OMD	0	0	0	0	0
OAN	0	0	0	0	0
JTI	0	0	0	0	0
JTD	0	0	0	0	0
JSI	17	37	59	52	44
JSD	0	1	14	7	0
JID	9	34	61	58	56
JDC	0	0	0	0	0
EOA	0	0	0	0	0
EOC	0	0	0	0	0
EAM	0	0	0	0	0
EMM	0	0	0	0	0

TABLE XV

TRADITIONAL MUTANTS PRODUCED FOR DIFFERENT AGPS

Size	**1,000**	**5,000**	**10,000**	**50,000**	**100,000**
Traditional	**30,706**	**102,851**	**191,860**	**568,333**	**1,493,549**
AORB	7,612	26,044	48,064	143,636	385,568
AORS	56	216	437	1,757	4,817
AOIU	460	1,339	2,758	6,779	15,893
AOIS	6,778	22,496	42,128	122,020	322,096
AODU	59	123	231	424	1,074
AODS	0	0	0	0	0
ROR	2,619	8,440	15,277	43,057	116,531
COR	346	1,082	1,964	5,494	14,848
COD	0	0	0	0	0
COI	720	2,398	4,345	12,956	34,501
SOR	0	0	0	0	0
LOR	0	0	0	0	0
LOI	1,893	6,219	11,694	33,440	87,653
LOD	0	0	0	0	0
ASRS	0	0	0	0	0
SDL	1,729	5,583	10,925	33,839	85,154
VDL	1,838	6,357	11,881	36,986	93,656
CDL	583	2,013	3,938	11,723	30,571
ODL	6,013	20,541	38,218	116,222	301,187

10.1.2 Degree of parallelism

As described in Section 8, α^5 is able to handle concurrent execution. To determine the right amount of threads the server is able to run, different experiments were executed with the 1K LOC AGP. The timeout was set to 100 seconds and the number of threads was empirically determined, after monitoring the amount of failed executions. Figure 24,Figure 25 and Figure 26 shows the smoothed memory utilization during execution, while Figure 27,Figure 28 and Figure 29 shows the non-smoothed version of the same data.

All the figures shows different spikes. The more the spikes are accentuated, the higher the number of threads that failed. For example, when looking at Figure 29 it becomes clear what is happening: used memory increases to very high value, threads fail due to excessive memory usage and are aborted, and used memory decreases again due to this failure. In particular Figure 29 contains more accentuated spikes than Figure 28, and Figure 28 in turn contains more picks than Figure 27.

The curve produces when executing the AUT using 100 thread has a very smooth profile, therefore it may be considered a good trade-off between precision and speed performances.

Table XVI shows how many mutants executions fails, depending on the number of thread specified for the execution. In particular two kind of situation may lead to a sudden increase of free memory:

- The current application run times out, resulting in an abortion.
- The current application run throws the following exception:

  java.io.IOException: Cannot run program "java": error=24, Too many open files

Using 1,000 thread resulted in less executions timeouts, but only because several runs were aborted before resulting in a timeout.

TABLE XVI

PERFORMANCES TRADE-OFF ON VARIATION OF THREAD NUMBERS

Lines Of Code	100	100	100
Concurrent Threads	1,000	500	100
Execution time (minutes)	17.3	19.65	39.5
# mutants to be executed	22791	22791	22791
# timeout	527	431	845
# ”too many open files” error	13342	7647	46

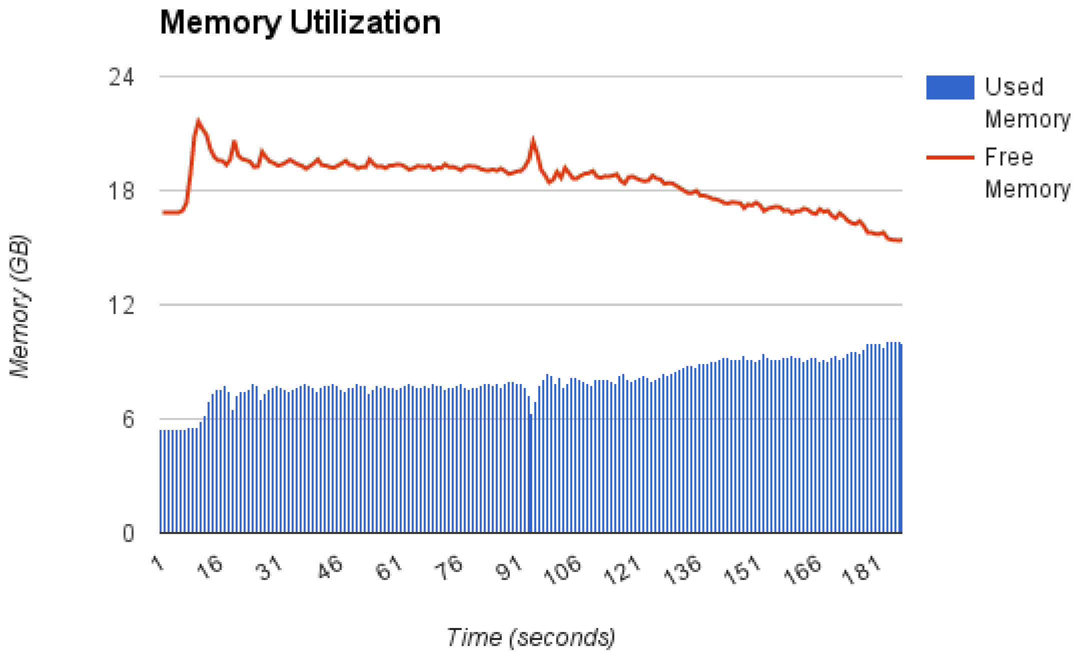


Figure 24. Memory Utilization during execution with 100 thread

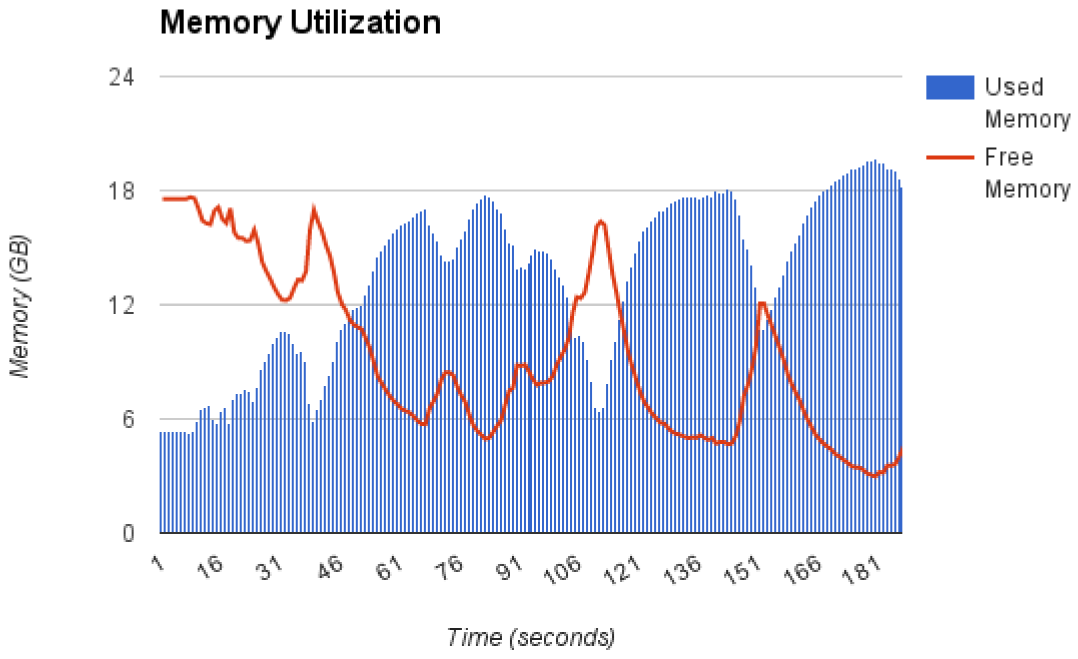


Figure 25. Memory Utilization during execution with 500 thread

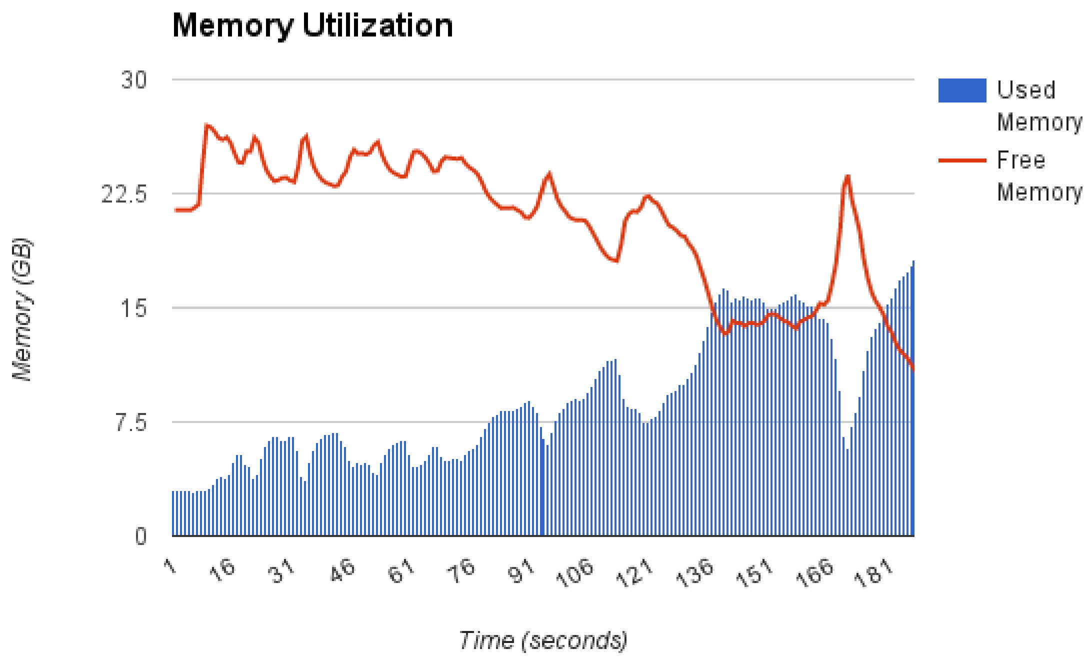


Figure 26. Memory Utilization during execution with 1,000 thread

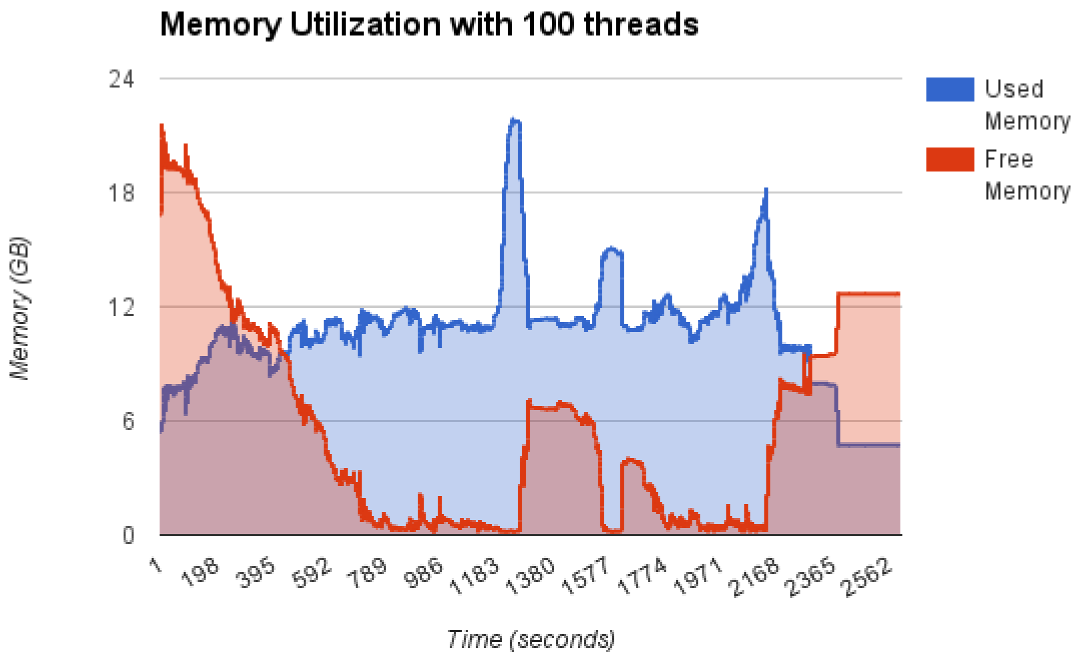


Figure 27. Memory Utilization during execution with 100 thread

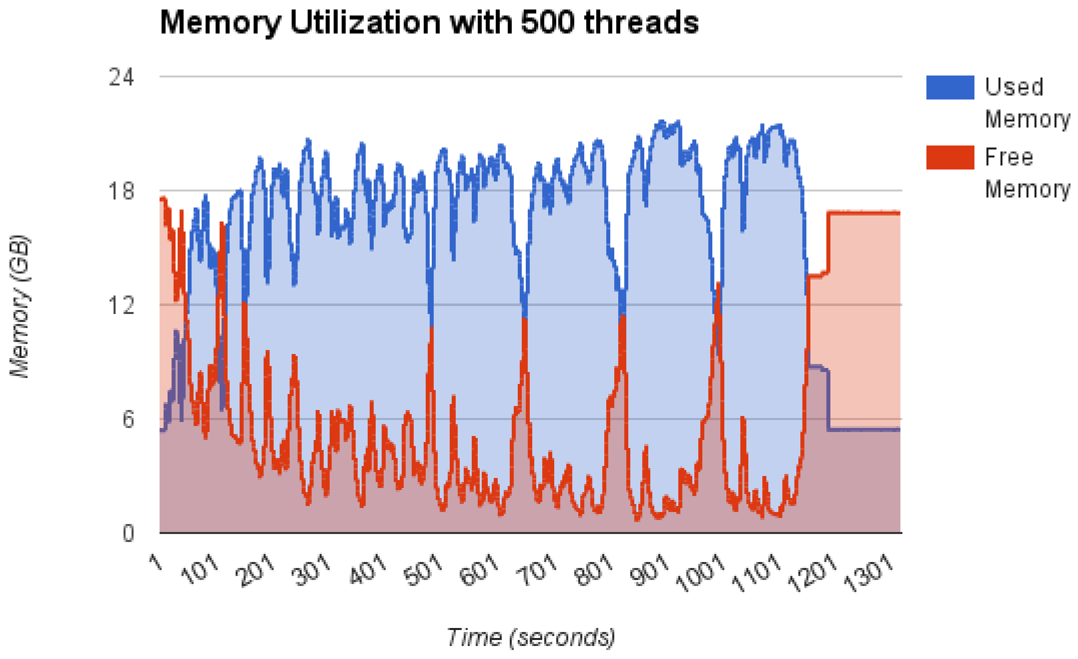


Figure 28. Memory Utilization during execution with 500 thread

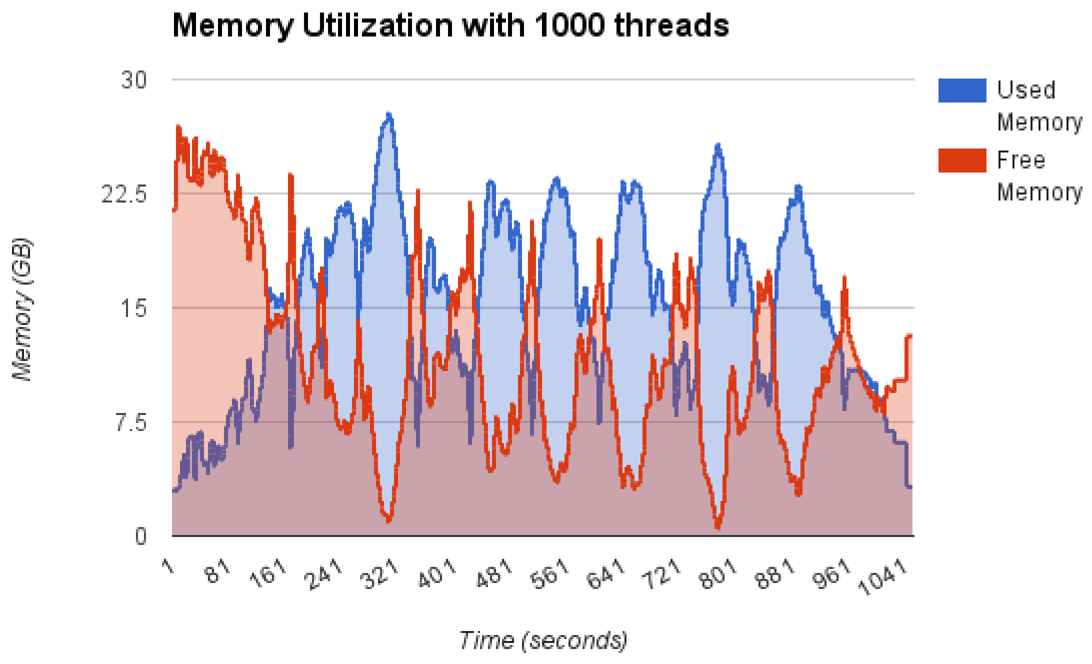


Figure 29. Memory Utilization during execution with 1,000 thread

10.2 'Real world' experiments

As described in Section 9.2 we selected real world applications presenting some field failure reported in the bug tracker. Table XVII reports the different versions selected from the AUT, each one of them corresponding to a different bug. Each one of them was mutated using MuJava, but interestingly enough some application triggered different exceptions while the AST was build, so distinct version of OJ were used as shown in Table XIX. This selected applications will be further analyzed to validate α^5.

TABLE XVII

BUGS DESCRIPTION FOR REPOSITORIES USED

searchbox-io/Jest	Jest/issues/219	Parameters are not applied to configuration
	Jest/issues/197	Double String
	Jest/issues/165	Date format
	Jest/issues/142	Missing field
	Jest/pull/139	Missing field
	Jest/pull/134	Date format
	Jest/pull/133	NPE
	Jest/issues/111	Wrong class of object
	Jest/issues/84	toString() doesn't work on Long class
	Jest/issues/78	Missing field, parameters not assigned
	Jest/issues/68	new Object() created, should use old
	Jest/issues/59	missing encoding, slash will cause a 400 error
	Jest/issues/60	Whitespaces removed, wrong encoding
	Jest/issues/62	ˆ
	Jest/issues/71	ˆ
fommil/matrix-toolkits-java	matrix-toolkits-java/issues/68	Wrong parameter
	matrix-toolkits-java/issues/58	Wrong value
	matrix-toolkits-java/issues/26	Date format
	matrix-toolkits-java/issues/25	Integer.MAX_VALUE
	matrix-toolkits-java/issues/13	Illegal value
	matrix-toolkits-java/issues/12	
	matrix-toolkits-java/issues/10	Wrong formula
peter-lawrey/Java-Chronicle	Java-Chronicle/issues/9	Wrong value
	Java-Chronicle/issues/5	wrong index
mikiobraun/jblas	jblas/issues/56	Double Float
	jblas/issues/42	Row/columns inverted
	jblas/issues/36	Wrong formula
	jblas/issues/29	Parameters are not applied to configuration
vkostyukov/la4j	la4j/issues/162	>32
	la4j/issues/98	Wrong rank?
	la4j/issues/94	ˆ
	la4j/issues/93	Wrong algorithm
	la4j/issues/82	Wrong algorithm
	la4j/issues/64	Infinite loop
	la4j/issues/53	Int Double
	la4j/issues/44	Wrong algorithm
	la4j/issues/16	+ ->-

TABLE XIX

OPENJAVA ERRORS USING DIFFERENT VERSIONS

AUT	TOTAL	Abstract	OJv0	OJv1	OJv2	OJv3	OJv4	OJv5	LOC	Mutated
geometry	261	85	42	52	42	42	**36**	42	73231	140
rxjava	228	45	92	86	92	**83**	158	92	64459	100
ONLP	547	118	152	126	152	**124**	301	152	61049	305
Apache-bcel	389	80	48	**19**	48	29	58	48	30895	290
jest	129	12	9	**8**	9	**8**	20	9	18248	109
fommil	123	24	43	**23**	43	**23**	24	43	14682	76
Chronicle	75	30	18	19	18	**14**	25	18	13163	31
la4j	87	43	8	**3**	8	8	18	8	13045	41
jblas	57	6	8	8	8	**5**	13	8	11839	46
spotify	40	1	**0**	**0**	**0**	**0**	16	**0**	9782	39
jactor	134	30	**1**	2	**1**	**1**	22	**1**	6716	103
classmate	34	8	9	8	9	**7**	13	9	6690	19
hamcrest	74	14	22	21	22	**17**	54	22	6139	43
javasim	41	1	3	3	3	3	**2**	3	3608	38

CHAPTER 11

FUTURE WORKS

We carried out our preliminary experiments on a number of Java applications whose sizes ranged from 10 to 100,000 LOC. The time it takes to inject faults, instrument, and execute the AUTs, and perform differential analyses on symptoms varies from approximately two hours for 1KLOC to less than 300 hours for 100KLOC. The same experiment with 200 VMs in parallel will finish in one hour and it will cost less than \$100 of the cloud time and storage. We already started exploring the issues of performance and scalability, and a scalable alternative to Alchemy may be Tuffy, which will significantly speeds up inference using MLN [336]. In addition, there are many opportunities to parallelise algorithms in Alchemy [337, 338] – recent work suggests using SMT solvers to improve the performance of Tuffy and Alchemy [339], thus, based on the experimental data, we expect that we can reduce the total time to less than a day for 100KLOC AUT. We plan to deploy α^5 in a highly parallel cloud setting where many instances of Alchemy or Tuffy will be running inference on the large MLN fault model, thus combining cloud computing and machine learning for production fault localisation.

The short-term impact of our work will be in the software testing community, where developers and testers will use α^5 to localise production faults automatically. The long-term impact will be on tools for software testing and analysis for commercial applications to allow stakeholders to localise production faults in their software applications with a high degree of automation and precision.

This research contains a strong educational component. Currently, software engineering and courses on probabilistic graphic models (as part of machine learning/data mining curriculum) are often taught without regard to each other, since they are considered orthogonal. However, it has been shown that machine learning and data mining algorithms and techniques can be used as part of solutions to many software engineering problems [340–343]. This research will contribute to forming a holistic view of problems in software engineering that benefit from using machine learning algorithms and possibly lead to the creation of new courses with the central theme to use machine learning and data mining in different software engineering tasks, especially in software testing.

11.1 Improving Effectiveness of the Sensitivity Analysis

The idea of injecting faults for sensitivity analysis of systems and eventually for fault localization is not new – it is routinely used in electrical, electronic, mechanical, automotive and many other industries as well as for software [344–346]. Fault injection was successfully used in validating reliability of file caches [347], comparing functionalities of operating systems [348, 349], web servers [350] and databases [351]. Various research shows that injecting artificial faults or mutants models real faults with a good approximation [352] – it is known as the competent programmer hypothesis [353] [7, 354].

Debroy and Wong proposed strategies for automatically locating faults in a program by using mutation [298, 299]. Papadakis et. al. used mutation for fault localisation [300–303]. Recent work of Moon, Kim and Yoo on MUSE also showed that fault localisation is more effective using mutants [304]. Zhang and Khurshid used mutation to simulate developer edits to localise faults [355]. However, these approaches use program mutation in conjunction with SBFL, whereas we solve a problem of localising *production* faults.

In α^5, we plan to experiment with fault injection to determine a strategy for the increased effectiveness of fault localization. A baseline fault injection strategy is execute a program with some chosen input data and then to apply mutation operators to program statements and expression that lie on the execution path for this input data. However, there are two problems with fault injection that affect its effectiveness: combinatorial explosion and fault interference. A root of the problem with the combinatorial explosion is that applying fault injection or mutation operators indiscriminately to all instructions results in a very large number of injected faults (i.e., mutants). In general, the number of generated mutants is proportional to the number of classes and references in object-oriented programs [356]; however, the number of all combinations of injected faults is very large. Since the Sensitivity Analyser executes each mutant program with different input values for differential analysis, we should address this problem in the future to make α^5 feasible.

The problem of combination explosion of injected faults can be tackled in four ways. First, it's possible to use PreFail, a programmable failure-injection tool that enables testers to create a wide range of policies to prune down the large space of multiple failures [357]. Second, for multiple injected faults (i.e., higher-order mutants (HOM)), we may utilize search-based approaches to identify subsuming HOMs that showed a potential for taming combinatorial explosion [358]. Third, we can utilize ideas from a recent approach called PAIN that shows that it is possible to significantly reduce interference of applications in parallel fault injection

[359]. An idea of parallelising software testing is not new [360–362] and with the availability of cloud computing infrastructures this idea is finally realised for improving the problem of combinatorial explosion [363]. For example, in D-Cloud, multiple copies of the same applications were installed in a large number of virtual machines and independently run without much interference [364, 365].

Finally, the fourth idea is to use algorithms for *combinatorial interaction testing (CIT)*, a field that identifies a small but systematic set of configurations under which to test [366–370]. For example, with a CIT approach, developers choose an interaction strength `t` and compute a covering array, which is a set of configurations such that all possible `t`-way combinations of option settings appear at least once. We hypothesize that using CIT in conjunction with sampling input parameter spaces will significantly reduce the exploration space and enable effective fault injection.

The other problem is that when injecting faults, interference is caused between the existing (unknown) faults in the AUT and the new injected faults. Of course, the issue of interference among faults has been studied to a certain extent [371, 372]. Some production faults linger due to the effect of being obscured by other faults [373]. As stated by one development manager in our interviews, "it makes a big difference to localise a fault in the presence of multiple fault interactions when compared with localization of a single fault, but there is not much difference in the fault localisation effort between say two and five interacting faults," a statement that resonated with an empirical study on the effects of the quantity of faults on fault localization techniques [374]. In future works, we will study the effect of multiple faults, including the effect on injected faults on latent production (unknown) faults: ① a positive effect when an injected fault masks the latent fault or vice versa to produce a correct result and ② when injected fault adds new symptoms.

APPENDICES

Appendix A

REPOSITORIES DESCRIPTION

Repository URL: https://github.com/ReactiveX/RxJava/

Repository Description: RxJava Reactive Extensions for the JVM a library for composing asynchronous and event-based programs using observable sequences for the Java VM.

Repository URL: https://github.com/hamcrest/JavaHamcrest/

Repository Description: Java (and original) version of Hamcrest

Repository URL: https://github.com/Esri/geometry-api-java/

Repository Description: The Esri Geometry API for Java enables developers to write custom applications for analysis of spatial data. This API is used in the Esri GIS Tools for Hadoop and other 3rd-party data processing solutions.

Repository URL: https://github.com/fommil/matrix-toolkits-java/

Repository Description: Java linear algebra library powered by BLAS and LAPACK

Repository URL: https://github.com/googlegenomics/api-client-java/

Repository Description: A command line tool for Google Genomics API queries.

Repository URL: https://github.com/cloudfoundry/java-buildpack-auto-reconfiguration/

Repository Description: Auto-reconfiguration functionality for the Java Buildpack

Repository URL: https://github.com/FasterXML/java-classmate/

Appendix A (continued)

Repository Description: Library for introspecting generic type information of types, member/static methods, fields. Especially useful for POJO/Bean introspection.

Repository URL: https://github.com/laforge49/JActor/

Repository Description: Actors for Java

Repository URL: https://github.com/functionaljava/functionaljava/

Repository Description: Functional programming in Java

Repository URL: https://github.com/thelinmichael/spotify-web-api-java/

Repository Description: A Java wrapper for the new Spotify Web API.

Repository URL: https://github.com/mikiobraun/jblas/

Repository Description: Linear Algebra for Java

Repository URL: https://github.com/vkostyukov/la4j/

Repository Description: Linear Algebra for Java

Repository URL: https://github.com/nmcl/JavaSim/

Repository Description: JavaSim simulation classes and examples

CITED LITERATURE

1. Allspaw, J.: Fault injection in production. Queue, 10(8):30:30–30:35, August 2012.

2. Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T.: What makes a good bug report? SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

3. Jeffrey, V., Larry, M., and Keith, M.: Predicting where faults can hide from testing. Technical report, 1991.

4. Jin, W. and Orso, A.: Bugredux: Reproducing field failures for in-house debugging. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

5. Beizer, B.: Software Testing Techniques. New York, Van Nostrand Reinhold, 2nd edition, 1990.

6. Jones, T. C.: Estimating Software Costs. New York, NY, USA, McGraw-Hill, Inc., 2 edition, 2007.

7. Ammann, P. and Offutt, J.: Introduction to Software Testing. New York, NY, USA, Cambridge University Press, 2008.

8. Myers, G. J.: Art of Software Testing. New York, NY, USA, John Wiley & Sons, Inc., 1979.

9. Leveson, N. G. and Turner, C. S.: An investigation of the therac-25 accidents. Computer, 26(7):18–41, July 1993.

10. Guevara, J., Stegman, E., and Hall, L.: It key metrics data 2010: Key applications measures: Support quality and practices: Current year. December 2009.

11. Evolven: Downtime, outages and failures - understanding their true costs. December 2012.

12. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

13. Upton, D., Hazelwood, K., Cohn, R., and Lueck, G.: Improving instrumentation speed via buffering. In Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA '09, pages 52–61, New York, NY, USA, 2009. ACM.

14. Uh, G.-R., Cohn, R., Yadavalli, B., Peri, R., and Ayyagari, R.: Analyzing dynamic binary instrumentation overhead. In Proceedings of the WBIA Workshop at ASPLOS, WBIA '06, pages 52–61, New York, NY, USA, 2006. ACM.

15. Jin, W. and Orso, A.: F3: Fault localization for field failures. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, pages 213–223, New York, NY, USA, 2013. ACM.

16. Wu, R., Zhang, H., Cheung, S.-C., and Kim, S.: Crashlocator: Locating crashing faults based on crash stacks. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 204–214, New York, NY, USA, 2014. ACM.

17. Cao, Y., Zhang, H., and Ding, S.: Symcrash: Selective recording for reproducing crashes. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 791–802, New York, NY, USA, 2014. ACM.

18. Artzi, S., Kim, S., and Ernst, M. D.: Recrashj: A tool for capturing and reproducing program crashes in deployed applications. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 295–296, New York, NY, USA, 2009. ACM.

19. Clause, J. and Orso, A.: A technique for enabling and supporting debugging of field failures. In Proceedings of the 29th International Conference on Software Engineering, ICSE '07, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

CITED LITERATURE (continued)

20. Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S., and Zhou, Y.: Understanding the propagation of hard errors to software and implications for resilient system design. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

21. Elbaum, S., Chin, H. N., Dwyer, M. B., and Dokulil, J.: Carving differential unit test cases from system test cases. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 253–264, New York, NY, USA, 2006. ACM.

22. Edwards, A., Tucker, S., and Demsky, B.: Afid: An automated approach to collecting software faults. Automated Software Engg., 17(3):347–372, September 2010.

23. Parnin, C. and Orso, A.: Are automated debugging techniques actually helping programmers? In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

24. Orso, A. and Rothermel, G.: Software testing: A research travelogue (2000&#8211;2014). In Proceedings of the on Future of Software Engineering, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

25. Zeller, A.: Why Programs Fail, Second Edition: A Guide to Systematic Debugging. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2nd edition, 2009.

26. Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. C.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw., 82(11):1780–1792, November 2009.

27. Clarke, L. A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng., 2(3):215–222, 1976.

28. DeMillo, R. A. and Offutt, A. J.: Constraint-based automatic test data generation. IEEE Trans. Softw. Eng., 17(9):900–910, 1991.

29. Godefroid, P.: Compositional dynamic test generation. In POPL, pages 47–54, 2007.

30. Majumdar, R. and Sen, K.: Hybrid concolic testing. In ICSE, pages 416–426, 2007.

31. Baresi, L. and Young, M.: Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

32. Peters, D. and Parnas, D. L.: Generating a test oracle from program documentation: work in progress. In ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, pages 58–65, New York, NY, USA, 1994. ACM Press.

33. Richardson, D. J., Leif-Aha, S., and O'Malley, T. O.: Specification-based Test Oracles for Reactive Systems. In Proceedings of the 14th ICSE, pages 105–118, May 1992.

34. Richardson, D. J.: Taos: Testing with analysis and oracle support. In ISSTA '94: Proceedings of the 1994 ACM SIGSOFT ISSTA, pages 138–153, New York, NY, USA, 1994. ACM Press.

35. Dillon, L. K. and Yu, Q.: Oracles for checking temporal properties of concurrent systems. In Proceedings of the ACM SIGSOFT '94, pages 140–153, December 1994.

36. Liu, C., Fei, L., Yan, X., Han, J., Member, S., and Midkiff, S. P.: Statistical debugging: A hypothesis testing-based approach. IEEE Transaction on Software Engineering, 32:831–848, 2006.

37. Arumuga Nainar, P., Chen, T., Rosin, J., and Liblit, B.: Statistical debugging using compound boolean predicates. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, pages 5–15, New York, NY, USA, 2007. ACM.

38. Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K.: Holmes: Effective statistical debugging via efficient path profiling. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

39. Liblit, B., Aiken, A., Zheng, A. X., and Jordan, M. I.: Bug isolation via remote program sampling. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM.

40. Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I.: Scalable statistical bug isolation. In Proceedings of the 2005 ACM SIGPLAN Conference on

Programming Language Design and Implementation, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

41. Hofer, B., Wotawa, F., and Abreu, R.: Ai for the win: Improving spectrum-based fault localization. SIGSOFT Softw. Eng. Notes, 37(6):1–8, November 2012.

42. Hao, D.: Testing-based interactive fault localization. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 957–960, New York, NY, USA, 2006. ACM.

43. Jones, J. A.: Semi-automatic Fault Localization. Doctoral dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2008. AAI3308774.

44. Zuddas, D., Jin, W., Pastore, F., Mariani, L., and Orso, A.: Mimic: Locating and understanding bugs by analyzing mimicked executions. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 815–826, New York, NY, USA, 2014. ACM.

45. Liblit, B.: Cooperative bug isolation. Doctoral dissertation, University of California, Berkeley, 2007.

46. Liblit, B.: Cooperative debugging with five hundred million test cases. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, pages 119–120, 2008.

47. Liblit, B.: Reflections on the role of static analysis in cooperative bug isolation. In Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, pages 18–31, 2008.

48. de Kleer, J. and Kurien, J.: Fundamentals of model-based diagnosis. In Proceedings of the 5th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes, June 2003.

49. Dow, E. and Farr, E.: Differential diagnosis for software systems: A meta-methodology for in-field root cause analysis. In Proceedings of the 7th Proactive Problem Prediction, Avoidance and Diagnosis Conference, Yorktown Heights, NY, April 2009, USA, pages 18–29, 2009.

50. Edwards, A., Tucker, S., Worms, S., Vaidya, R., and Demsky, B.: AFID: an automated fault identification tool. In Proceedings of the ACM/SIGSOFT International

Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, pages 179–188, 2008.

51. Bell, J., Sarda, N., and Kaiser, G.: Chronicler: Lightweight recording to reproduce field failures. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press.

52. Rahman, M.: Droidmln: A markov logic network approach to detect android malware. In Proceedings of the 2013 12th International Conference on Machine Learning and Applications - Volume 02, ICMLA '13, pages 166–169, Washington, DC, USA, 2013. IEEE Computer Society.

53. Chahuara, P., Portet, F., and Vacher, M.: Making context aware decision from uncertain information in a smart home: A markov logic network approach. In Proceedings of the 4th International Joint Conference on Ambient Intelligence - Volume 8309, AmI 2013, pages 78–93, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

54. Cheng, V. and Li, C. H.: Topic detection via participation using markov logic network. In Proceedings of the 2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, SITIS '07, pages 85–91, Washington, DC, USA, 2007. IEEE Computer Society.

55. Souza, C. R. C. and Santos, P. E.: Probabilistic logic reasoning about traffic scenes. In Proceedings of the 12th Annual Conference on Towards Autonomous Robotic Systems, TAROS'11, pages 219–230, Berlin, Heidelberg, 2011. Springer-Verlag.

56. Chatzikonstantinou, G., Kontogiannis, K., and Attarian, I.-M.: A goal driven framework for software project data analytics. In Proceedings of the 25th International Conference on Advanced Information Systems Engineering, CAiSE'13, pages 546–561, Berlin, Heidelberg, 2013. Springer-Verlag.

57. Ferilli, S., Basile, T. M. A., and Di Mauro, N.: Markov logic networks for document layout correction. In Proceedings of the 24th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems Conference on Modern Approaches in Applied Intelligence - Volume Part I, IEA/AIE'11, pages 275–284, Berlin, Heidelberg, 2011. Springer-Verlag.

58. Riedel, S., Chun, H.-W., Takagi, T., and Tsujii, J.: A markov logic approach to biomolecular event extraction. In Proceedings of the Workshop on Current Trends

in Biomedical Natural Language Processing: Shared Task, BioNLP '09, pages 41–49, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

59. Zawawy, H., Kontogiannis, K., Mylopoulos, J., and Mankovskii, S.: Towards a requirements-driven framework for detecting malicious behavior against software systems. In Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '11, pages 15–29, Riverton, NJ, USA, 2011. IBM Corp.

60. Lee, S., Lee, J., Noh, H., Lee, K., and Lee, G. G.: Grammatical error simulation for computer-assisted language learning. Know.-Based Syst., 24(6):868–876, August 2011.

61. Wen, W.: Software fault localization based on program slicing spectrum. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 1511–1514, Piscataway, NJ, USA, 2012. IEEE Press.

62. Liu, Y., Li, W., Jiang, S., Zhang, Y., and Ju, X.: An approach for fault localization based on program slicing and bayesian. In Proceedings of the 2013 13th International Conference on Quality Software, QSIC '13, pages 326–332, Washington, DC, USA, 2013. IEEE Computer Society.

63. Sedlmeyer, R. L., Thompson, W. B., and Johnson, P. E.: Knowledge-based fault localization in debugging: Preliminary draft. In Proceedings of the Symposium on High-level Debugging, SIGSOFT '83, pages 25–31, New York, NY, USA, 1983. ACM.

64. Liu, C., Fei, L., Yan, X., Han, J., and Midkiff, S. P.: Statistical debugging: A hypothesis testing-based approach. IEEE Trans. Softw. Eng., 32(10):831–848, October 2006.

65. Zhang, S. and Zhang, C.: Software bug localization with markov logic. In Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pages 424–427, New York, NY, USA, 2014. ACM.

66. Zeller, A. and Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng., 28(2):183–200, February 2002.

67. Yu, K., Lin, M., Chen, J., and Zhang, X.: Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. J. Syst. Softw., 85(10):2305–2317, October 2012.

CITED LITERATURE (continued)

68. Cleve, H. and Zeller, A.: Locating causes of program failures. In Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM.

69. Misherghi, G. and Su, Z.: Hdd: Hierarchical delta debugging. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 142–151, New York, NY, USA, 2006. ACM.

70. Zeller, A.: Isolating cause-effect chains from computer programs. In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM.

71. Zeller, A.: Yesterday, my program worked. today, it does not. why? In Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7, pages 253–267, London, UK, UK, 1999. Springer-Verlag.

72. Zeller, A.: Search-based program analysis. In Proceedings of the Third International Conference on Search Based Software Engineering, SSBSE'11, pages 1–4, Berlin, Heidelberg, 2011. Springer-Verlag.

73. Jin, G., Zhang, W., Deng, D., Liblit, B., and Lu, S.: Automated concurrency-bug fixing. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pages 221–236, Berkeley, CA, USA, 2012. USENIX Association.

74. Park, S., Vuduc, R. W., and Harrold, M. J.: Falcon: Fault localization in concurrent programs. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

75. Copty, S. and Ur, S.: Toward automatic concurrent debugging via minimal program mutant generation with aspectj. Electron. Notes Theor. Comput. Sci., 174(9):151–165, June 2007.

76. Park, S., Vuduc, R., and Harrold, M. J.: A unified approach for localizing non-deadlock concurrency bugs. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 51–60, Washington, DC, USA, 2012. IEEE Computer Society.

CITED LITERATURE (continued)

77. Choi, J.-D. and Zeller, A.: Isolating failure-inducing thread schedules. In Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '02, pages 210–220, New York, NY, USA, 2002. ACM.

78. Park, S.: Debugging non-deadlock concurrency bugs. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, pages 358–361, New York, NY, USA, 2013. ACM.

79. Park, S., Harrold, M. J., and Vuduc, R.: Griffin: Grouping suspicious memory-access patterns to improve understanding of concurrency bugs. In Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, pages 134–144, New York, NY, USA, 2013. ACM.

80. Steimann, F. and Frenkel, M.: Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In Proceedings of the 2012 IEEE 23rd International Symposium on Software Reliability Engineering, ISSRE '12, pages 121–130, Washington, DC, USA, 2012. IEEE Computer Society.

81. Masri, W.: Fault localization based on information flow coverage. Softw. Test. Verif. Reliab., 20(2):121–147, June 2010.

82. Wong, E., Wei, T., Qi, Y., and Zhao, L.: A crosstab-based statistical method for effective fault localization. In Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST '08, pages 42–51, Washington, DC, USA, 2008. IEEE Computer Society.

83. Wang, X., Cheung, S. C., Chan, W. K., and Zhang, Z.: Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 45–55, Washington, DC, USA, 2009. IEEE Computer Society.

84. Yu, K., Lin, M., Chen, J., and Zhang, X.: Practical isolation of failure-inducing changes for debugging regression faults. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 20–29, New York, NY, USA, 2012. ACM.

85. Yu, K., Lin, M., Gao, Q., Zhang, H., and Zhang, X.: Locating faults using multiple spectra-specific models. In Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, pages 1404–1410, New York, NY, USA, 2011. ACM.

CITED LITERATURE (continued)

86. Liu, C., Zhang, X., and Han, J.: A systematic study of failure proximity. IEEE Trans. Softw. Eng., 34(6):826–843, November 2008.

87. Wotawa, F. and Soomro, S.: Fault localization based on abstract dependencies. In Proceedings of the 18th International Conference on Innovations in Applied Artificial Intelligence, IEA/AIE'2005, pages 357–359, London, UK, UK, 2005. Springer-Verlag.

88. Lucia, Lo, D., Jiang, L., and Budi, A.: Comprehensive evaluation of association measures for fault localization. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

89. Masri, W., Abou-Assi, R., El-Ghali, M., and Al-Fatairi, N.: An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), DEFECTS '09, pages 1–5, New York, NY, USA, 2009. ACM.

90. Hsu, H.-Y., Jones, J. A., and Orso, A.: Rapid: Identifying bug signatures to support debugging activities. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pages 439–442, Washington, DC, USA, 2008. IEEE Computer Society.

91. Fleurey, F., Traon, Y. L., and Baudry, B.: From testing to diagnosis: An automated approach. In Proceedings of the 19th IEEE International Conference on Automated Software Engineering, ASE '04, pages 306–309, Washington, DC, USA, 2004. IEEE Computer Society.

92. Wang, S., Lo, D., Jiang, L., Lucia, and Lau, H. C.: Search-based fault localization. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 556–559, Washington, DC, USA, 2011. IEEE Computer Society.

93. Lucia, Lo, D., and Xia, X.: Fusion fault localizers. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 127–138, New York, NY, USA, 2014. ACM.

CITED LITERATURE (continued)

94. Machado, P., Campos, J., and Abreu, R.: Mzoltar: Automatic debugging of android applications. In Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile, DeMobile 2013, pages 9–16, New York, NY, USA, 2013. ACM.

95. Lei, Y., Mao, X., and Chen, T. Y.: Backward-slice-based statistical fault localization without test oracles. In Proceedings of the 2013 13th International Conference on Quality Software, QSIC '13, pages 212–221, Washington, DC, USA, 2013. IEEE Computer Society.

96. Xie, X., Wong, W. E., Chen, T. Y., and Xu, B.: Metamorphic slice: An application in spectrum-based fault localization. Inf. Softw. Technol., 55(5):866–879, May 2013.

97. Chen, C., Gross, H.-G., and Zaidman, A.: Spectrum-based fault diagnosis for service-oriented software systems. In Proceedings of the 2012 5th IEEE International Conference on Service-Oriented Computing and Applications (SOCA), SOCA '12, pages 1–8, Washington, DC, USA, 2012. IEEE Computer Society.

98. Yu, Z., Hu, H., Bai, C., Cai, K.-Y., and Wong, W. E.: Gui software fault localization using n-gram analysis. In Proceedings of the 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering, HASE '11, pages 325–332, Washington, DC, USA, 2011. IEEE Computer Society.

99. Yu, Z., Bai, C., and Cai, K.-Y.: Mutation-oriented test data augmentation for gui software fault localization. Inf. Softw. Technol., 55(12):2076–2098, December 2013.

100. Ocariza Jr., F. S., Pattabiraman, K., and Mesbah, A.: Autoflox: An automatic fault localizer for client-side javascript. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 31–40, Washington, DC, USA, 2012. IEEE Computer Society.

101. Ma, C., Zhang, Y., Liu, J., and Mengzhao: Locating faulty code using failure-causing input combinations in combinatorial testing. In Proceedings of the 2013 Fourth World Congress on Software Engineering, WCSE '13, pages 91–98, Washington, DC, USA, 2013. IEEE Computer Society.

102. Song, S.: Estimating the effectiveness of spectrum-based fault localization. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 814–816, New York, NY, USA, 2014. ACM.

CITED LITERATURE (continued)

103. Ren, X. and Ryder, B. G.: Heuristic ranking of java program edits for fault localization. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, pages 239–249, New York, NY, USA, 2007. ACM.

104. Alves, E., Gligoric, M., Jagannath, V., and d'Amorim, M.: Fault-localization using dynamic slicing and change impact analysis. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 520–523, Washington, DC, USA, 2011. IEEE Computer Society.

105. Ren, X., Chesley, O. C., and Ryder, B. G.: Identifying failure causes in java programs: An application of change impact analysis. IEEE Trans. Softw. Eng., 32(9):718–732, September 2006.

106. Fouché, S., Cohen, M. B., and Porter, A.: Towards incremental adaptive covering arrays. In The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, ESEC-FSE companion '07, pages 557–560, New York, NY, USA, 2007. ACM.

107. Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., and Cao, H.: Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices. J. Syst. Softw., 90:3–17, April 2014.

108. Guo, H.-F., Qiu, Z., and Siy, H.: Locating fault-inducing patterns from structural inputs. In Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, pages 1100–1107, New York, NY, USA, 2014. ACM.

109. Baudry, B., Fleurey, F., and Le Traon, Y.: Improving test suites for efficient fault localization. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 82–91, New York, NY, USA, 2006. ACM.

110. Hao, D., Pan, Y., Zhang, L., Zhao, W., Mei, H., and Sun, J.: A similarity-aware approach to testing based fault localization. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, pages 291–294, New York, NY, USA, 2005. ACM.

111. Hao, D., Zhang, L., Zhong, H., Mei, H., and Sun, J.: Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study. In Proceedings of the 21st IEEE International Conference on Software

Maintenance, ICSM '05, pages 683–686, Washington, DC, USA, 2005. IEEE Computer Society.

112. Hao, D., Xie, T., Zhang, L., Wang, X., Sun, J., and Mei, H.: Test input reduction for result inspection to facilitate fault localization. Automated Software Engg., 17(1):5–31, March 2010.

113. Hao, D., Zhang, L., Pan, Y., Mei, H., and Sun, J.: On similarity-awareness in testing-based fault localization. Automated Software Engg., 15(2):207–249, June 2008.

114. Yu, Y., Jones, J. A., and Harrold, M. J.: An empirical study of the effects of test-suite reduction on fault localization. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 201–210, New York, NY, USA, 2008. ACM.

115. Dandan, G., Tiantian, W., Xiaohong, S., and Peijun, M.: A test-suite reduction approach to improving fault-localization effectiveness. Comput. Lang. Syst. Struct., 39(3):95–108, October 2013.

116. Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. C.: An empirical study on the usage of testability information to fault localization in software. In Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, pages 1398–1403, New York, NY, USA, 2011. ACM.

117. Kim, S. and Baik, J.: An effective fault aware test case prioritization by incorporating a fault localization technique. In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '10, pages 5:1–5:10, New York, NY, USA, 2010. ACM.

118. Gong, L., Lo, D., Jiang, L., and Zhang, H.: Diversity maximization speedup for fault localization. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 30–39, New York, NY, USA, 2012. ACM.

119. Zhang, X., Gu, Q., Chen, X., Qi, J., and Chen, D.: A study of relative redundancy in test-suite reduction while retaining or improving fault-localization effectiveness. In Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, pages 2229–2236, New York, NY, USA, 2010. ACM.

CITED LITERATURE (continued)

120. Demott, J. D., Enbody, R. J., and Punch, W. F.: Systematic bug finding and fault localization enhanced with input data tracking. Comput. Secur., 32:130–157, February 2013.

121. Jones, J. A. and Harrold, M. J.: Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

122. Steimann, F. and Bertschler, M.: A simple coverage-based locator for multiple faults. In Proceedings of the 2009 International Conference on Software Testing Verification and Validation, ICST '09, pages 366–375, Washington, DC, USA, 2009. IEEE Computer Society.

123. Eric Wong, W., Debroy, V., and Choi, B.: A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw., 83(2):188–208, February 2010.

124. Burger, M. and Zeller, A.: Minimizing reproduction of software failures. In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, pages 221–231, New York, NY, USA, 2011. ACM.

125. Masri, W. and Assi, R. A.: Cleansing test suites from coincidental correctness to enhance fault-localization. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pages 165–174, Washington, DC, USA, 2010. IEEE Computer Society.

126. Bandyopadhyay, A.: Improving spectrum-based fault localization using proximity-based weighting of test cases. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 660–664, Washington, DC, USA, 2011. IEEE Computer Society.

127. Masri, W. and Assi, R. A.: Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol., 23(1):8:1–8:28, February 2014.

128. Bandyopadhyay, A.: Mitigating the effect of coincidental correctness in spectrum based fault localization. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 479–482, Washington, DC, USA, 2012. IEEE Computer Society.

CITED LITERATURE (continued)

129. Bandyopadhyay, A. and Ghosh, S.: Tester feedback driven fault localization. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 41–50, Washington, DC, USA, 2012. IEEE Computer Society.

130. Liu, C. and Han, J.: Failure proximity: A fault localization-based approach. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 46–56, New York, NY, USA, 2006. ACM.

131. Debroy, V., Wong, W. E., Xu, X., and Choi, B.: A grouping-based strategy to improve the effectiveness of fault localization techniques. In Proceedings of the 2010 10th International Conference on Quality Software, QSIC '10, pages 13–22, Washington, DC, USA, 2010. IEEE Computer Society.

132. Wong, W. E., Debroy, V., Li, Y., and Gao, R.: Software fault localization using dstar (d*). In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, SERE '12, pages 21–30, Washington, DC, USA, 2012. IEEE Computer Society.

133. Weiglhofer, M., Fraser, G., and Wotawa, F.: Using spectrum-based fault localization for test case grouping. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pages 630–634, Washington, DC, USA, 2009. IEEE Computer Society.

134. Guo, L., Roychoudhury, A., and Wang, T.: Accurately choosing execution runs for software fault localization. In Proceedings of the 15th International Conference on Compiler Construction, CC'06, pages 80–95, Berlin, Heidelberg, 2006. Springer-Verlag.

135. Xu, J., Zhang, Z., Chan, W. K., Tse, T. H., and Li, S.: A general noise-reduction framework for fault localization of java programs. Inf. Softw. Technol., 55(5):880–896, May 2013.

136. Zhang, L., Kim, M., and Khurshid, S.: Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 40:1–40:4, New York, NY, USA, 2012. ACM.

CITED LITERATURE (continued)

137. Zhang, L., Kim, M., and Khurshid, S.: Localizing failure-inducing program edits based on spectrum information. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, ICSM '11, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.

138. Robetaler, J., Fraser, G., Zeller, A., and Orso, A.: Isolating failure causes through test case generation. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012, pages 309–319, New York, NY, USA, 2012. ACM.

139. Li, Z., Cheng, L., Qiu, X.-S., and Wu, L.: Fault diagnosis for high-level applications based on dynamic bayesian network. In Proceedings of the 12th Asia-Pacific Network Operations and Management Conference on Management Enabling the Future Internet for Changing Business and New Computing Services, APNOMS'09, pages 61–70, Berlin, Heidelberg, 2009. Springer-Verlag.

140. Steinder, M. and Sethi, A. S.: Probabilistic fault localization in communication systems using belief networks. IEEE/ACM Trans. Netw., 12(5):809–822, October 2004.

141. Jeffrey, D., Gupta, N., and Gupta, R.: Fault localization using value replacement. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, pages 167–178, New York, NY, USA, 2008. ACM.

142. Shu, G., Sun, B., Podgurski, A., and Cao, F.: Mfl: Method-level fault localization with causal inference. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST '13, pages 124–133, Washington, DC, USA, 2013. IEEE Computer Society.

143. Parsa, S., Naree, S. A., and Koopaei, N. E.: Software fault localization via mining execution graphs. In Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part II, ICCSA'11, pages 610–623, Berlin, Heidelberg, 2011. Springer-Verlag.

144. Parsa, S., Vahidi-Asl, M., and Asadi-Aghbolaghi, M.: Hierarchy-debug: A scalable statistical technique for fault localization. Software Quality Control, 22(3):427–466, September 2014.

145. Liu, C., Yan, X., Yu, H., Han, J., and Yu, P. S.: Mining behavior graphs for "backtrace" of noncrashing bugs. In Proceedings of the 2005 SIAM International Conference

on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005, pages 286–297, 2005.

146. Liu, C., Yan, X., and Han, J.: Mining control flow abnormality for logic error isolation. In Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA, pages 106–117, 2006.

147. Sözer, H., Abreu, R., Aksit, M., and van Gemund, A. J. C.: Increasing system availability with local recovery based on fault localization. In Proceedings of the 2010 10th International Conference on Quality Software, QSIC '10, pages 276–281, Washington, DC, USA, 2010. IEEE Computer Society.

148. Artzi, S., Dolby, J., Tip, F., and Pistoia, M.: Fault localization for dynamic web applications. IEEE Trans. Softw. Eng., 38(2):314–335, March 2012.

149. Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., and Nguyen, T. N.: Database-aware fault localization for dynamic web applications. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, pages 456–459, Washington, DC, USA, 2013. IEEE Computer Society.

150. Artzi, S., Dolby, J., Tip, F., and Pistoia, M.: Directed test generation for effective fault localization. In Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pages 49–60, New York, NY, USA, 2010. ACM.

151. Artzi, S., Dolby, J., Tip, F., and Pistoia, M.: Practical fault localization for dynamic web applications. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 265–274, New York, NY, USA, 2010. ACM.

152. Al-Kofahi, J., Nguyen, H. V., and Nguyen, T. N.: Fault localization for build code errors in makefiles. In Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pages 600–601, New York, NY, USA, 2014. ACM.

153. Wang, T. and Roychoudhury, A.: Automated path generation for software fault localization. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, pages 347–351, New York, NY, USA, 2005. ACM.

CITED LITERATURE (continued)

154. Delahaye, M., Briand, L. C., Gotlieb, A., and Petit, M.: &#181;til: Mutation-based statistical test inputs generation for automatic fault localization. In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, SERE '12, pages 197–206, Washington, DC, USA, 2012. IEEE Computer Society.

155. Soffa, M. L., Walcott, K. R., and Mars, J.: Exploiting hardware advances for software testing and debugging (nier track). In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 888–891, New York, NY, USA, 2011. ACM.

156. Zuo, Z.: Efficient statistical debugging via hierarchical instrumentation. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 457–460, New York, NY, USA, 2014. ACM.

157. Dean, B. C., Pressly, W. B., Malloy, B. A., and Whitley, A. A.: A linear programming approach for automated localization of multiple faults. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pages 640–644, Washington, DC, USA, 2009. IEEE Computer Society.

158. Kannan, K. and Bhamidipaty, A.: A differential approach for configuration fault localization in cloud environments. In Proceedings of the 2013 IEEE International Conference on Cloud Engineering, IC2E '13, pages 250–257, Washington, DC, USA, 2013. IEEE Computer Society.

159. Dallmeier, V., Lindig, C., and Zeller, A.: Lightweight defect localization for java. In Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag.

160. Gore, R., Reynolds, P. F., and Kamensky, D.: Statistical debugging with elastic predicates. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 492–495, Washington, DC, USA, 2011. IEEE Computer Society.

161. Clark, S. R., Cobb, J., Kapfhammer, G. M., Jones, J. A., and Harrold, M. J.: Localizing sql faults in database applications. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 213–222, Washington, DC, USA, 2011. IEEE Computer Society.

CITED LITERATURE (continued)

162. Hewett, R.: Program spectra analysis with theory of evidence. Adv. Soft. Eng., 2012:1:1–1:1, January 2012.

163. Ghandehari, L. S. G., Lei, Y., Xie, T., Kuhn, R., and Kacker, R.: Identifying failure-inducing combinations in a combinatorial test set. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 370–379, Washington, DC, USA, 2012. IEEE Computer Society.

164. Shakya, K., Xie, T., Li, N., Lei, Y., Kacker, R., and Kuhn, R.: Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST '12, pages 620–623, Washington, DC, USA, 2012. IEEE Computer Society.

165. Wong, W. E., Shi, Y., Qi, Y., and Golden, R.: Using an rbf neural network to locate program bugs. In Proceedings of the 2008 19th International Symposium on Software Reliability Engineering, ISSRE '08, pages 27–36, Washington, DC, USA, 2008. IEEE Computer Society.

166. Zhang, Z., Chan, W. K., Tse, T. H., Yu, Y. T., and Hu, P.: Non-parametric statistical fault localization. J. Syst. Softw., 84(6):885–905, June 2011.

167. Xuan, J. and Monperrus, M.: Test case purification for improving fault localization. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.

168. Jiang, B., Zhang, Z., Chan, W. K., Tse, T. H., and Chen, T. Y.: How well does test case prioritization integrate with statistical fault localization? Inf. Softw. Technol., 54(7):739–758, July 2012.

169. Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. C.: Prioritizing tests for fault localization through ambiguity group reduction. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 83–92, Washington, DC, USA, 2011. IEEE Computer Society.

170. Sun, C.-A., Zhai, Y. M., Shang, Y., and Zhang, Z.: Bpeldebugger: An effective bpel-specific fault localization framework. Inf. Softw. Technol., 55(12):2140–2153, De-

cember 2013.

171. Perez, A., Abreu, R., and Riboira, A.: A dynamic code coverage approach to maximize fault localization efficiency. J. Syst. Softw., 90:18–28, April 2014.

172. Zhang, Z., Jiang, B., Chan, W. K., Tse, T. H., and Wang, X.: Fault localization through evaluation sequences. J. Syst. Softw., 83(2):174–187, February 2010.

173. Ma, C., Tan, T., Chen, Y., and Dong, Y.: An if-while-if model-based performance evaluation of ranking metrics for spectra-based fault localization. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference, COMPSAC '13, pages 609–618, Washington, DC, USA, 2013. IEEE Computer Society.

174. Le, T.-D. B., Thung, F., and Lo, D.: Theory and practice, do they match? a case with spectrum-based fault localization. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, pages 380–383, Washington, DC, USA, 2013. IEEE Computer Society.

175. Debroy, V. and Wong, W. E.: On the equivalence of certain fault localization techniques. In Proceedings of the 2011 ACM Symposium on Applied Computing, SAC '11, pages 1457–1463, New York, NY, USA, 2011. ACM.

176. Gong, C., Zheng, Z., Zhang, Y., Zhang, Z., and Xue, Y.: Factorising the multiple fault localization problem: Adapting single-fault localizer to multi-fault programs. In Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference - Volume 01, APSEC '12, pages 729–732, Washington, DC, USA, 2012. IEEE Computer Society.

177. Le, T.-D. B. and Lo, D.: Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, pages 310–319, Washington, DC, USA, 2013. IEEE Computer Society.

178. DeMillo, R. A., Pan, H., and Spafford, E. H.: Critical slicing for software fault localization. In Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '96, pages 121–134, New York, NY, USA, 1996. ACM.

CITED LITERATURE (continued)

179. Surendran, A. and Samuel, P.: Fault localization using forward slicing spectrum. In Proceedings of the 2013 Research in Adaptive and Convergent Systems, RACS '13, pages 397–398, New York, NY, USA, 2013. ACM.

180. Kusumoto, S., Nishimatsu, A., Nishie, K., and Inoue, K.: Experimental evaluation of program slicing for fault localization. Empirical Softw. Engg., 7(1):49–76, March 2002.

181. Mao, X., Lei, Y., Dai, Z., Qi, Y., and Wang, C.: Slice-based statistical fault localization. J. Syst. Softw., 89:51–62, March 2014.

182. Roychowdhury, S. and Khurshid, S.: Software fault localization using feature selection. In Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering, MALETS '11, pages 11–18, New York, NY, USA, 2011. ACM.

183. Huang, T.-Y., Chou, P.-C., Tsai, C.-H., and Chen, H.-A.: Automated fault localization with statistically suspicious program states. In Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '07, pages 11–20, New York, NY, USA, 2007. ACM.

184. Ruthruff, J. R., Prabhakararao, S., Reichwein, J., Cook, C., Creswick, E., and Burnett, M.: Interactive, visual fault localization support for end-user programmers. J. Vis. Lang. Comput., 16(1-2):3–40, February 2005.

185. Ruthruff, J. R., Burnett, M., and Rothermel, G.: An empirical study of fault localization for end-user programmers. In Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pages 352–361, New York, NY, USA, 2005. ACM.

186. Hofer, B., Riboira, A., Wotawa, F., Abreu, R., and Getzner, E.: On the empirical evaluation of fault localization techniques for spreadsheets. In Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13, pages 68–82, Berlin, Heidelberg, 2013. Springer-Verlag.

187. Ruthruff, J. R., Burnett, M., and Rothermel, G.: Interactive fault localization techniques in a spreadsheet environment. IEEE Trans. Softw. Eng., 32(4):213–239, April 2006.

188. Yoo, S., Harman, M., and Clark, D.: Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. ACM Trans. Softw. Eng. Methodol., 22(3):19:1–19:29, July 2013.

189. Ackling, T., Alexander, B., and Grunert, I.: Evolving patches for software repair. In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO '11, pages 1427–1434, New York, NY, USA, 2011. ACM.

190. Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S.: Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

191. Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C.: A genetic programming approach to automated software repair. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

192. Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

193. Martinez, M., Weimer, W., and Monperrus, M.: Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pages 492–495, New York, NY, USA, 2014. ACM.

194. Kern, C. and Esparza, J.: Automatic error correction of java programs. In Proceedings of the 15th International Conference on Formal Methods for Industrial Critical Systems, FMICS'10, pages 67–81, Berlin, Heidelberg, 2010. Springer-Verlag.

195. Schulte, E., DiLorenzo, J., Weimer, W., and Forrest, S.: Automated repair of binary and assembly programs for cooperating embedded devices. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 317–328, New York, NY, USA, 2013. ACM.

CITED LITERATURE (continued)

196. Arcuri, A.: Evolutionary repair of faulty software. Appl. Soft Comput., 11(4):3494–3514, June 2011.

197. Schulte, E., Forrest, S., and Weimer, W.: Automated program repair through the evolution of assembly code. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

198. Zhang, X., Gupta, N., and Gupta, R.: Locating faults through automated predicate switching. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 272–281, New York, NY, USA, 2006. ACM.

199. Yilmaz, C. and Williams, C.: An automated model-based debugging approach. In Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pages 174–183, New York, NY, USA, 2007. ACM.

200. Sedlmeyer, R. L., Thompson, W. B., and Johnson, P. E.: Diagnostic reasoning in software fault localization. In Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'83, pages 29–31, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

201. Banerjee, A., Roychoudhury, A., Harlie, J. A., and Liang, Z.: Golden implementation driven software debugging. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM.

202. Malik, M. Z., Ghori, K., Elkarablieh, B., and Khurshid, S.: A case for automated debugging using data structure repair. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pages 620–624, Washington, DC, USA, 2009. IEEE Computer Society.

203. Köb, D. and Wotawa, F.: Fundamentals of debugging using a resolution calculus. In Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering, FASE'06, pages 278–292, Berlin, Heidelberg, 2006. Springer-Verlag.

204. Griesmayer, A., Staber, S., and Bloem, R.: Automated fault localization for c programs. Electron. Notes Theor. Comput. Sci., 174(4):95–111, May 2007.

205. Jose, M. and Majumdar, R.: Bug-assist: Assisting fault localization in ansi-c programs. In Proceedings of the 23rd International Conference on Computer Aided Verification, CAV'11, pages 504–509, Berlin, Heidelberg, 2011. Springer-Verlag.

206. Jose, M. and Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

207. Balakrishnan, G. and Ganai, M.: Ped: Proof-guided error diagnosis by triangulation of program error causes. In Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM '08, pages 268–278, Washington, DC, USA, 2008. IEEE Computer Society.

208. Griesmayer, A., Staber, S., and Bloem, R.: Fault localization using a model checker. Softw. Test. Verif. Reliab., 20(2):149–173, June 2010.

209. Jirong, S., Zhishu, L., Jiancheng, N., and Feng, Y.: Priority strategy of software fault localization. In Proceedings of the 6th Conference on WSEAS International Conference on Applied Computer Science - Volume 6, ACOS'07, pages 499–505, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

210. Savor, T. and Seviora, R. E.: Hierarchical supervisors for automatic detection of software failures. In Proceedings of the Eighth International Symposium on Software Reliability Engineering, ISSRE '97, pages 48–, Washington, DC, USA, 1997. IEEE Computer Society.

211. Gopinath, D., Zaeem, R. N., and Khurshid, S.: Improving the effectiveness of spectra-based fault localization using specifications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pages 40–49, New York, NY, USA, 2012. ACM.

212. Sahoo, S. K., Criswell, J., Geigle, C., and Adve, V.: Using likely invariants for automated software fault localization. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. ACM.

213. Roychowdhury, S. and Khurshid, S.: A novel framework for locating software faults using latent divergences. In Proceedings of the 2011 European Conference on

Machine Learning and Knowledge Discovery in Databases - Volume Part III, ECML PKDD'11, pages 49–64, Berlin, Heidelberg, 2011. Springer-Verlag.

214. Deng, F. and Jones, J. A.: Inferred dependence coverage to support fault contextualization. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, pages 512–515, Washington, DC, USA, 2011. IEEE Computer Society.

215. Bohnet, J., Voigt, S., and Döllner, J.: Projecting code changes onto execution traces to support localization of recently introduced bugs. In Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, pages 438–442, New York, NY, USA, 2009. ACM.

216. Ge, N., Nakajima, S., and Pantel, M.: Efficient online analysis of accidental fault localization for dynamic systems using hidden markov model. In Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium, DEVS 13, pages 16:1–16:8, San Diego, CA, USA, 2013. Society for Computer Simulation International.

217. Baah, G. K., Gray, A., and Harrold, M. J.: On-line anomaly detection of deployed software: A statistical machine learning approach. In Proceedings of the 3rd International Workshop on Software Quality Assurance, SOQUA '06, pages 70–77, New York, NY, USA, 2006. ACM.

218. Casanova, P., Schmerl, B., Garlan, D., and Abreu, R.: Architecture-based runtime fault diagnosis. In Proceedings of the 5th European Conference on Software Architecture, ECSA'11, pages 261–277, Berlin, Heidelberg, 2011. Springer-Verlag.

219. Casanova, P., Garlan, D., Schmerl, B., and Abreu, R.: Diagnosing unobserved components in self-adaptive systems. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, pages 75–84, New York, NY, USA, 2014. ACM.

220. Gao, Z., Chen, Z., Feng, Y., and Luo, B.: Mining sequential patterns of predicates for fault localization and understanding. In Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability, SERE '13, pages 109–118, Washington, DC, USA, 2013. IEEE Computer Society.

221. Kaleeswaran, S., Tulsian, V., Kanade, A., and Orso, A.: Minthint: Automated synthesis of repair hints. In Proceedings of the 36th International Conference on Software

Engineering, ICSE 2014, pages 266–276, New York, NY, USA, 2014. ACM.

222. Abreu, R., Zoeteweij, P., and Van Gemund, A. J. C.: A new bayesian approach to multiple intermittent fault diagnosis. In Proceedings of the 21st International Jont Conference on Artifical Intelligence, IJCAI'09, pages 653–658, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

223. Stumptner, M. and Wotawa, F.: A survey of intelligent debugging. AI Commun., 11(1):35–51, January 1998.

224. Zhou, B., Kulkarni, M., and Bagchi, S.: Wukong: Effective diagnosis of bugs at large system scales. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, pages 317–318, New York, NY, USA, 2013. ACM.

225. Mayer, W. and Stumptner, M.: Modeling programs with unstructured control flow for debugging. In Proceedings of the 15th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence, AI '02, pages 107–118, London, UK, UK, 2002. Springer-Verlag.

226. Casanova, P., Garlan, D., Schmerl, B., and Abreu, R.: Diagnosing architectural run-time failures. In Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13, pages 103–112, Piscataway, NJ, USA, 2013. IEEE Press.

227. Abreu, R., Zoeteweij, P., and van Gemund, A. J. C.: An observation-based model for fault localization. In Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA '08, pages 64–70, New York, NY, USA, 2008. ACM.

228. Abreu, R. and van Gemund, A. J. C.: Diagnosing multiple intermittent failures using maximum likelihood estimation. Artif. Intell., 174(18):1481–1497, December 2010.

229. Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A.: Automated fixing of programs with contracts. In Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pages 61–72, New York, NY, USA, 2010. ACM.

CITED LITERATURE (continued)

230. Ceballos, R., Gasca, R. M., Del Valle, C., and Borrego, D.: Diagnosing errors in dbc programs using constraint programming. In Proceedings of the 11th Spanish Association Conference on Current Topics in Artificial Intelligence, CAEPIA'05, pages 200–210, Berlin, Heidelberg, 2006. Springer-Verlag.

231. Mayer, W. and Stumptner, M.: Better debugging through more abstract observations. In Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy, pages 779–780, Amsterdam, The Netherlands, The Netherlands, 2006. IOS Press.

232. Abreu, R., Gonzalez-Sanchez, A., and van Gemund, A. J. C.: Exploiting count spectra for bayesian fault localization. In Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE '10, pages 12:1–12:10, New York, NY, USA, 2010. ACM.

233. Yang, B., Zhang, M., and Zhang, Y.: Applying answer set programming to points-to analysis of object-oriented language. In Proceedings of the 7th International Conference on Advanced Intelligent Computing, ICIC'11, pages 676–685, Berlin, Heidelberg, 2011. Springer-Verlag.

234. Abreu, R., Zoeteweij, P., and van Gemund, A. J. C.: Simultaneous debugging of software faults. J. Syst. Softw., 84(4):573–586, April 2011.

235. Stumptner, M.: Intelligent information processing ii. chapter Model-based Diagnosis and Debugging, pages .26–.27. London, UK, UK, Springer-Verlag, 2005.

236. Stumptner, M.: Using design information to identify structural software faults. In Proceedings of the 14th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence, AI '01, pages 473–486, London, UK, UK, 2001. Springer-Verlag.

237. Abreu, R., Mayer, W., Stumptner, M., and van Gemund, A. J. C.: Refining spectrum-based fault localization rankings. In Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, pages 409–414, New York, NY, USA, 2009. ACM.

238. Bourahla, M.: Model-based diagnostic using model checking. In Proceedings of the 2009 Fourth International Conference on Dependability of Computer Systems, DEPCOS-RELCOMEX '09, pages 229–236, Washington, DC, USA, 2009. IEEE Computer Society.

CITED LITERATURE (continued)

239. Wotawa, F.: On the relationship between model-based debugging and program slicing. Artif. Intell., 135(1-2):125–143, February 2002.

240. Peischl, B. and Wotawa, F.: Error traces in model-based debugging of hardware description languages. In Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging, AADEBUG'05, pages 43–48, New York, NY, USA, 2005. ACM.

241. Wotawa, F., Stumptner, M., and Mayer, W.: Model-based debugging or how to diagnose programs automatically. In Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence, IEA/AIE '02, pages 746–757, London, UK, UK, 2002. Springer-Verlag.

242. Mayer, W. and Stumptner, M.: Abstract interpretation of programs for model-based debugging. In Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI'07, pages 471–476, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

243. Struss, P., Shivashankar, V., and Zahoor, M.: A fault-model-based debugging aid for data warehouse applications. In Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence, pages 419–424, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.

244. Mayer, W. and Stumptner, M.: Model-based debugging – state of the art and future challenges. Electron. Notes Theor. Comput. Sci., 174(4):61–82, May 2007.

245. Mayer, W. and Stumptner, M.: Intelligent information processing ii. chapter Model-based Debugging with High-level Observations, pages 299–309. London, UK, UK, Springer-Verlag, 2005.

246. Mayer, W. and Stumptner, M.: Evaluating models for model-based debugging. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pages 128–137, Washington, DC, USA, 2008. IEEE Computer Society.

247. Liu, C., Lian, Z., and Han, J.: How bayesians debug. In Proceedings of the Sixth International Conference on Data Mining, ICDM '06, pages 382–393, Washington, DC, USA, 2006. IEEE Computer Society.

CITED LITERATURE (continued)

248. Abreu, R., González, A., Zoeteweij, P., and van Gemund, A. J. C.: Automatic software fault localization using generic program invariants. In Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08, pages 712–717, New York, NY, USA, 2008. ACM.

249. Li, H., Liu, Y., Zhang, Z., and Liu, J.: Program structure aware fault localization. In Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices, InnoSWDev 2014, pages 40–48, New York, NY, USA, 2014. ACM.

250. Baah, G. K., Podgurski, A., and Harrold, M. J.: Causal inference for statistical fault localization. In Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, pages 73–84, New York, NY, USA, 2010. ACM.

251. Baah, G. K.: Statistical Causal Analysis for Fault Localization. Doctoral dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2012. AAI3535858.

252. Guo, X., Song, X., Hung, W. N. N., Gu, M., and Sun, J.: Fault localization with partially reliable test results using dempster-shafer theory. In Proceedings of the 2014 Theoretical Aspects of Software Engineering Conference (Tase 2014), TASE '14, pages 58–65, Washington, DC, USA, 2014. IEEE Computer Society.

253. Baah, G. K., Podgurski, A., and Harrold, M. J.: Mitigating the confounding effects of program dependences for effective fault localization. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 146–156, New York, NY, USA, 2011. ACM.

254. Zhao, L., Wang, L., Xiong, Z., and Gao, D.: Execution-aware fault localization based on the control flow analysis. In Proceedings of the First International Conference on Information Computing and Applications, ICICA'10, pages 158–165, Berlin, Heidelberg, 2010. Springer-Verlag.

255. Zhang, Z., Chan, W. K., Tse, T. H., Jiang, B., and Wang, X.: Capturing propagation of infected program states. In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 43–52, New York, NY, USA, 2009. ACM.

256. Feng, M. and Gupta, R.: Learning universal probabilistic models for fault localization. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '10, pages 81–88, New York, NY, USA, 2010. ACM.

257. Mousavian, Z., Vahidi-Asl, M., and Parsa, S.: Scalable graph analyzing approach for software fault-localization. In Proceedings of the 6th International Workshop on Automation of Software Test, AST '11, pages 15–21, New York, NY, USA, 2011. ACM.

258. Naish, L., Lee, H. J., and Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol., 20(3):11:1–11:32, August 2011.

259. Stoerzer, M., Ryder, B. G., Ren, X., and Tip, F.: Finding failure-inducing changes in java programs using change classification. In Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 57–68, New York, NY, USA, 2006. ACM.

260. Peyvandi-Pour, A. and Parsa, S.: Effective software fault localization by statistically testing the program behavior model. In Proceedings of the Second International Conference on Information Computing and Applications, ICICA'11, pages 136–144, Berlin, Heidelberg, 2011. Springer-Verlag.

261. Baah, G. K., Podgurski, A., and Harrold, M. J.: The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. Softw. Eng., 36(4):528–545, July 2010.

262. Baah, G. K., Podgurski, A., and Harrold, M. J.: The probabilistic program dependence graph and its application to fault diagnosis. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, pages 189–200, New York, NY, USA, 2008. ACM.

263. Mariani, L., Pastore, F., and Pezze, M.: Dynamic analysis for diagnosing integration faults. IEEE Trans. Softw. Eng., 37(4):486–508, July 2011.

264. Cellier, P., Ducassé, M., Ferré, S., and Ridoux, O.: Formal concept analysis enhances fault localization in software. In Proceedings of the 6th International Conference on Formal Concept Analysis, ICFCA'08, pages 273–288, Berlin, Heidelberg, 2008. Springer-Verlag.

265. Dhoolia, P., Mani, S., Sinha, V. S., and Sinha, S.: Debugging model-transformation failures using dynamic tainting. In Proceedings of the 24th European Conference on Object-oriented Programming, ECOOP'10, pages 26–51, Berlin, Heidelberg, 2010. Springer-Verlag.

266. Jobstmann, B., Staber, S., Griesmayer, A., and Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci., 78(2):441–460, March 2012.

267. Zhang, C., Liao, J., and Zhu, X.: Probabilistic event-driven heuristic fault localization using incremental bayesian suspected degree. In Proceedings of the 2008 The 9th International Conference for Young Computer Scientists, ICYCS '08, pages 653–658, Washington, DC, USA, 2008. IEEE Computer Society.

268. Li, C., Liu, L., and Pang, X.: A dynamic probability fault localization algorithm using digraph. In Proceedings of the 2009 Fifth International Conference on Natural Computation - Volume 06, ICNC '09, pages 187–191, Washington, DC, USA, 2009. IEEE Computer Society.

269. Novotny, P., Wolf, A. L., and Ko, B. J.: Fault localization in manet-hosted service-based systems. In Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems, SRDS '12, pages 243–248, Washington, DC, USA, 2012. IEEE Computer Society.

270. Sharma, A. B., Chen, H., Ding, M., Yoshihira, K., and Jiang, G.: Fault detection and localization in distributed systems using invariant relationships. In Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), DSN '13, pages 1–8, Washington, DC, USA, 2013. IEEE Computer Society.

271. Nguyen, H., Shen, Z., Tan, Y., and Gu, X.: Fchain: Toward black-box online fault localization for cloud systems. In Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems, ICDCS '13, pages 21–30, Washington, DC, USA, 2013. IEEE Computer Society.

272. Nguyen, H., Tan, Y., and Gu, X.: Pal: Propagation-aware anomaly localization for cloud hosted distributed applications. In Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques, SLAML '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

CITED LITERATURE (continued)

273. Song, L. and Lu, S.: Statistical debugging for real-world performance problems. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014, pages 561–578, 2014.

274. Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M.: Towards highly reliable enterprise network services via inference of multi-level dependencies. In Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '07, pages 13–24, New York, NY, USA, 2007. ACM.

275. Prakash, P., Kompella, R. R., Ramasubramanian, V., and Chandra, R.: dfault: Fault localization in large-scale peer-to-peer systems. In Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware, Middleware '10, pages 252–272, Berlin, Heidelberg, 2010. Springer-Verlag.

276. Kim, J., Yang, Y.-M., Park, S., Lee, S., and Chung, B.: Service fault localization using functional events separation and modeling of service resources. In Proceedings of the 12th Asia-Pacific Network Operations and Management Conference on Management Enabling the Future Internet for Changing Business and New Computing Services, APNOMS'09, pages 462–465, Berlin, Heidelberg, 2009. Springer-Verlag.

277. Natu, M. and Sethi, A. S.: Using temporal correlation for fault localization in dynamically changing networks. Int. J. Netw. Manag., 18(4):301–314, August 2008.

278. Mysore, R. N., Mahajan, R., Vahdat, A., and Varghese, G.: Gestalt: Fast, unified fault localization for networked systems. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14, pages 255–268, Berkeley, CA, USA, 2014. USENIX Association.

279. Steinder, M. and Sethi, A. S.: Probabilistic fault diagnosis in communication systems through incremental hypothesis updating. Comput. Netw., 45(4):537–562, July 2004.

280. Abreu, R., Zoeteweij, P., and Gemund, A. J. C. v.: Spectrum-based multiple fault localization. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

CITED LITERATURE (continued)

281. Janssen, T., Abreu, R., and van Gemund, A. J.: Zoltar: A spectrum-based fault localization tool. In Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime, SINTER '09, pages 23–30, New York, NY, USA, 2009. ACM.

282. Diaz, M. and Frances, D. M.: Bayesian inference using gibbs sampling in applications and curricula of decision analysis. INFORMS Trans. Edu., 14(2):86–95, February 2014.

283. Stern, R. and Kalech, M.: Model-based diagnosis techniques for internet delay diagnosis with dynamic routing. Applied Intelligence, 41(1):167–183, July 2014.

284. Liu, Z. and Han, Z.: Fault diagnosis of electric railway traction substation with model-based relation guiding algorithm. Expert Syst. Appl., 41(4):1730–1741, March 2014.

285. Mishra, N., Kumar Choudhary, A., Tiwari, M. K., and Shankar, R.: Rollout strategy-based probabilistic causal model approach for the multiple fault diagnosis. Robot. Comput.-Integr. Manuf., 26(4):325–332, August 2010.

286. Straszecka, E.: Combining uncertainty and imprecision in models of medical diagnosis. Inf. Sci., 176(20):3026–3059, October 2006.

287. Bae, H., Chun, S.-P., and Kim, S.: Predictive fault detection and diagnosis of nuclear power plant using the two-step neural network models. In Proceedings of the Third International Conference on Advances in Neural Networks - Volume Part III, ISNN'06, pages 420–425, Berlin, Heidelberg, 2006. Springer-Verlag.

288. Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L.: Efficient software-based fault isolation. In Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

289. Denaro, G. and Pezzè, M.: An empirical evaluation of fault-proneness models. In Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pages 241–251, New York, NY, USA, 2002. ACM.

290. Ren-Wu, Y. and Jin-Ding, C.: Fault diagnosis of power electronic circuit based on random forests algorithm and ar model. In Proceedings of the 2009 Second International

Conference on Information and Computing Science - Volume 01, ICIC '09, pages 285–288, Washington, DC, USA, 2009. IEEE Computer Society.

291. Liu, J. and Tian, W.: A novel fault diagnosis model research for electronic circuit. In Proceedings of the 2Nd International Asia Conference on Informatics in Control, Automation and Robotics - Volume 2, CAR'10, pages 5–8, Piscataway, NJ, USA, 2010. IEEE Press.

292. Misera, S., Vierhaus, H. T., and Sieber, A.: Simulated fault injections and their acceleration in systemc. Microprocess. Microsyst., 32(5-6):270–278, August 2008.

293. Calvano, J. V., De Mesquita Filho, A. C., Alves, V. C., and Lubaszewski, M. S.: Fault models and test generation for opamp circuits&mdash;the ffm. J. Electron. Test., 17(2):121–138, April 2001.

294. Azarian, A. and Siadat, A.: A global modular framework for automotive diagnosis. Adv. Eng. Inform., 26(1):131–144, January 2012.

295. Biamonte, J. D., Allen, J. S., and Perkowski, M. A.: Fault models for quantum mechanical switching networks. J. Electron. Test., 26(5):499–511, October 2010.

296. Tao, J.: Design of resource space model in fault diagnosis knowledge of rotating machinery. In Proceedings of the 2008 Fourth International Conference on Semantics, Knowledge and Grid, SKG '08, pages 501–502, Washington, DC, USA, 2008. IEEE Computer Society.

297. Koller, D. and Friedman, N.: Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning. The MIT Press, 2009.

298. Debroy, V. and Wong, W. E.: Using mutation to automatically suggest fixes for faulty programs. In Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pages 65–74, Washington, DC, USA, 2010. IEEE Computer Society.

299. Debroy, V. and Wong, W. E.: Combining mutation and fault localization for automated program debugging. J. Syst. Softw., 90:45–60, April 2014.

300. Papadakis, M. and Le Traon, Y.: Using mutants to locate "unknown" faults. In Proceedings of the 2012 IEEE Fifth International Conference on

CITED LITERATURE (continued)

Software Testing, Verification and Validation, ICST '12, pages 691–700, Washington, DC, USA, 2012. IEEE Computer Society.

301. Papadakis, M. and Traon, Y. L.: Effective fault localization via mutation analysis: a selective mutation approach. In Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014, pages 1293–1300, 2014.

302. Papadakis, M., Delamaro, M. E., and Traon, Y. L.: Proteum/fl: A tool for localizing faults using mutation analysis. In 13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013, pages 94–99, 2013.

303. Papadakis, M. and Traon, Y. L.: Using mutants to locate "unknown" faults. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012, pages 691–700, 2012.

304. Moon, S., Kim, Y., Kim, M., and Yoo, S.: Ask the mutants: Mutating faulty programs for fault localization. In IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, pages 153–162, 2014.

305. Cai, H., Jiang, S., jie Zhang, Y., Zhang, Y., and Santelices, R.: Sensa: Sensitivity analysis for quantitative change-impact prediction. IEEE SCAM '14, pages 110–119, Washington, DC, USA, 2014. IEEE Computer Society.

306. Wainwright, H. M., Finsterle, S., Jung, Y., Zhou, Q., and Birkholzer, J. T.: Making sense of global sensitivity analyses. Comput. Geosci., 65:84–94, April 2014.

307. Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., and Tarantola, S.: Global Sensitivity Analysis: The Primer. New York, NY, USA, Wiley-Interscience; 1 edition, 2008.

308. Qi, D., Roychoudhury, A., Liang, Z., and Vaswani, K.: Darwin: An approach to debugging evolving programs. ACM Trans. Softw. Eng. Methodol., 21(3):19:1–19:29, July 2012.

309. Richardson, M. and Domingos, P.: Markov logic networks. Mach. Learn., 62(1-2):107–136, February 2006.

CITED LITERATURE (continued)

310. Parag, P.: Markov Logic: Theory, Algorithms and Applications. Doctoral dissertation, Seattle, WA, USA, 2009. AAI0821714.

311. Gong, L., Zhang, H., Jiang, L., and Lo, D.: Interactive fault localization leveraging simple user feedback. In Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM), ICSM '12, pages 67–76, Washington, DC, USA, 2012. IEEE Computer Society.

312. Ramamurthi, A., Roy, S., and Srikant, Y. N.: Probabilistic dataflow analysis using path profiles on structure graphs. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 512–515, New York, NY, USA, 2011. ACM.

313. Luckow, K., Păsăreanu, C. S., Dwyer, M. B., Filieri, A., and Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pages 575–586, New York, NY, USA, 2014. ACM.

314. Wang, J., Byrnes, J., Valtorta, M., and Huhns, M.: On the combination of logical and probabilistic models for information analysis. Applied Intelligence, 36(2):472–497, March 2012.

315. Claret, G., Rajamani, S. K., Nori, A. V., Gordon, A. D., and Borgström, J.: Bayesian inference using data flow analysis. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pages 92–102, New York, NY, USA, 2013. ACM.

316. Fatima, J.: Clinical Decision Support System: Differential Diagnosis. Germany, LAP Lambert Academic Publishing, 2012.

317. Fischer, C. and Gregor, S.: Forms of reasoning in the design science research process. In Proceedings of the 6th International Conference on Service-oriented Perspectives in Design Science Research, DESRIST'11, pages 17–31, Berlin, Heidelberg, 2011. Springer-Verlag.

318. Jeff offutt home page. https://cs.gmu.edu/~offutt/. Accessed: 2015-10-05.

319. Mujava home page. https://cs.gmu.edu/~offutt/mujava/. Accessed: 2015-10-05.

320. Description of class mutation mutation operators for java. `https://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`. Accessed: 2015-10-05.

321. Description of method-level mutation operators for java. `https://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf`. Accessed: 2015-10-05.

322. Oj (f.a.k.a. openjava) : An extensible java. `http://openjava.sourceforge.net/`. Accessed: 2015-10-05.

323. Bitwise and bit shift operators (the java tutorials; learning the java language; language basics). `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html`. Accessed: 2015-10-05.

324. Javassist, java bytecode engineering toolkit since 1999. `http://jboss-javassist.github.io/javassist/`. Accessed: 2015-10-05.

325. Combinatorial interaction testing portal. `http://cse.unl.edu/~citportal/`. Accessed: 2015-10-05.

326. sun.misc: Unsafe.java. `http://www.docjar.com/html/api/sun/misc/Unsafe.java.html`. Accessed: 2015-10-05.

327. sun.tools.javac.main : Java glossary. `http://mindprod.com/jgloss/javacmain.html`. Accessed: 2015-10-05.

328. Eclipse corner article: Abstract syntax tree. `https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`. Accessed: 2015-10-05.

329. Help - eclipse platform. `http://help.eclipse.org/mars/index.jsp?topic=\%2Forg.eclipse.jdt.doc.isv\%2Freference\%2Fapi\%2Forg\%2Feclipse\%2Fjdt\%2Fcore\%2Fdom\%2FAST.html`. Accessed: 2015-10-05.

330. Eclipse java development tools (jdt). `http://www.eclipse.org/jdt/`. Accessed: 2015-10-05.

331. Alchemy: Open source ai. `https://alchemy.cs.washington.edu/`. Accessed: 2015-10-05.

332. Abductive reasoning - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Abductive_reasoning#Deduction.2C_induction.2C_and_abduction`. Accessed: 2015-10-05.

333. Selenium - web browser automation. `http://www.seleniumhq.org/`. Accessed: 2015-10-05.

334. Cloc – count lines of code. `http://cloc.sourceforge.net/`. Accessed: 2015-10-05.

335. Hussain, I., Csallner, C., Grechanik, M., Xie, Q., Park, S., Taneja, K., and Hossain, M.: Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. Software: Practice and Experience, 2014.

336. Niu, F., Ré, C., Doan, A., and Shavlik, J.: Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. Proc. VLDB Endow., 4(6):373–384, March 2011.

337. Domingos, P.: Parallelising Alchemy, 2006 (accessed January 10, 2015).

338. Beedkar, K., Del Corro, L., and Gemulla, R.: Fully parallel inference in markov logic networks. In 15th GI-Symposium Database Systems for Business, Technology and Web (BTW 2013), Magdeburg, Germany, 2013. Bonner Kllen.

339. Chaganty, A., Lal, A., Nori, A. V., and Rajamani, S. K.: Combining relational learning with smt solvers using cegar. In Proceedings of the 25th International Conference on Computer Aided Verification, CAV'13, pages 447–462, Berlin, Heidelberg, 2013. Springer-Verlag.

340. Rashid, E., Patnayak, S., and Bhattacherjee, V.: A survey in the area of machine learning and its application for software quality prediction. SIGSOFT Softw. Eng. Notes, 37(5):1–7, September 2012.

341. Zhang, D. and Tsai, J. J. P.: Machine Learning Applications in Software Engineering. World Scientific Pub Co Inc, February 2005.

342. Zhang, D. and Tsai, J. J. P.: Advances in Machine Learning Applications in Software Engineering. IGI Global, February 2007.

CITED LITERATURE (continued)

343. Haran, M., Karr, A. F., Orso, A., Porter, A. A., and Sanil, A. P.: Applying classification techniques to remotely-collected program execution data. In ESEC/SIGSOFT FSE, pages 146–155, 2005.

344. Duraes, J. A. and Madeira, H. S.: Emulation of software faults: A field data study and a practical approach. IEEE Trans. Softw. Eng., 32(11):849–867, November 2006.

345. Cotroneo, D. and Natella, R.: Fault injection for software certification. IEEE Security and Privacy, 11(4):38–45, July 2013.

346. Voas, J. M. and McGraw, G.: Software Fault Injection: Inoculating Programs Against Errors. New York, NY, USA, John Wiley & Sons, Inc., 1997.

347. Ng, W. T. and Chen, P. M.: The design and verification of the rio file cache. IEEE Trans. Comput., 50(4):322–337, April 2001.

348. Duraes, J. and Madeira, H.: Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. Transactions of IEICE (Institute of the Electronics, Information and Communication Engineers), 86(12):2563–2570, 2003.

349. Koopman, P. and DeVale, J.: The exception handling effectiveness of posix operating systems. IEEE Trans. Softw. Eng., 26(9):837–848, September 2000.

350. Durães, J., Vieira, M., and Madeira, H.: Dependability benchmarking of web-servers. In Computer Safety, Reliability, and Security, 23rd International Conference, SAFECOMP 2004, Potsdam, Germany, September 21-24, 2004, Proceedings, pages 297–310, 2004.

351. Vieira, M. and Madeira, H.: A dependability benchmark for oltp application environments. In VLDB, pages 742–753, 2003.

352. Andrews, J. H., Briand, L. C., and Labiche, Y.: Is mutation an appropriate tool for testing experiments? In Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

353. Duraes, J., Madeira, H., Cotroneo, D., and Natella, R.: On fault representativeness of software fault injection. IEEE Transactions on Software Engineering, 39(1):80–96, 2013.

CITED LITERATURE (continued)

354. Mathur, A. P.: Foundations of Software Testing. Addison-Wesley Professional, 1st edition, 2008.

355. Zhang, L., Zhang, L., and Khurshid, S.: Injecting mechanical faults to localize developer faults for evolving software. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 765–784, 2013.

356. Mateo, P. R., Usaola, M. P., and Offutt, J.: Mutation at system and functional levels. ICSTW '10, pages 110–119, Washington, DC, USA, 2010. IEEE Computer Society.

357. Joshi, P., Gunawi, H. S., and Sen, K.: Prefail: A programmable tool for multiple-failure injection. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

358. Jia, Y. and Harman, M.: Constructing subtle faults using higher order mutation testing. 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 0:249–258, 2008.

359. Winter, S., Schwahn, O., Natellay, R., Suri, N., and Cotroneo, D.: No pain, no gain? the utility of parallel fault injections. In Proceedings of the 37th International Conference on Software Engineering, ICSE '15, page to appear, New York, NY, USA, 2015. ACM.

360. Duarte, A., Cirne, W., Brasileiro, F., and Machado, P.: Gridunit: Software testing on the grid. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 779–782, New York, NY, USA, 2006. ACM.

361. Yoon, I., Sussman, A., Memon, A., and Porter, A.: Testing component compatibility in evolving configurations. Inf. Softw. Technol., 55(2):445–458, February 2013.

362. Lastovetsky, A.: Parallel testing of distributed software. Inf. Softw. Technol., 47(10):657–662, July 2005.

363. Al-qadhi, M. and Keung, J.: Cloud-based support for global software engineering: Potentials, risks, and gaps. In Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices, InnoSWDev 2014, pages 57–64, New York, NY, USA, 2014. ACM.

CITED LITERATURE (continued)

364. Banzai, T., Koizumi, H., Kanbayashi, R., Imada, T., Hanawa, T., and Sato, M.: D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGRID '10, pages 631–636, Washington, DC, USA, 2010. IEEE Computer Society.

365. Hanawa, T., Banzai, T., Koizumi, H., Kanbayashi, R., Imada, T., and Sato, M.: Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems. In Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, ICSTW '10, pages 428–433, Washington, DC, USA, 2010. IEEE Computer Society.

366. Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C.: The aetg system: An approach to testing based on combinatorial design. IEEE Trans. Softw. Eng., 23(7):437–444, July 1997.

367. Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C.: The AETG system: an approach to testing based on combinatorial design. Trans. Soft. Eng., 23(7):437–44, 1997.

368. Yoon, I., Sussman, A., Memon, A., and Porter, A.: Testing component compatibility in evolving configurations. Information and Software Technology, 55(2):445 – 458, 2013.

369. Reisner, E., Song, C., Ma, K.-K., Foster, J. S., and Porter, A.: Using symbolic evaluation to understand behavior in configurable software systems. In ICSE, pages 445–454, 2010.

370. Song, C., Porter, A., and Foster, J. S.: itree: efficiently discovering high-coverage configurations using interaction trees. In ICSE, pages 903–913, 2012.

371. Debroy, V. and Wong, W. E.: Insights on fault interference for programs with multiple bugs. In Proceedings of the 2009 20th International Symposium on Software Reliability Engineering, ISSRE '09, pages 165–174, Washington, DC, USA, 2009. IEEE Computer Society.

372. Jones, J. A., Bowring, J. F., and Harrold, M. J.: Debugging in parallel. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA '07, pages 16–26, New York, NY, USA, 2007. ACM.

CITED LITERATURE (continued)

373. DiGiuseppe, N. and Jones, J. A.: Fault interaction and its repercussions. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance, ICSM '11, pages 3–12, Washington, DC, USA, 2011. IEEE Computer Society.

374. DiGiuseppe, N. and Jones, J. A.: On the influence of multiple faults on coverage-based fault localization. In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, pages 210–220, New York, NY, USA, 2011. ACM.

VITA

NAME	Davide Pagano
EDUCATION	
	Master of Science in Computer Science, University of Illinois at Chicago, Dec 2015, USA
	Master of Science in Computer Engineering, Polytechnic of Milan, Dec 2015, Italy
	Bachelor of Science in Computer Engineering, Polytechnic of Milan, Jul 2013, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2013 - TOEFL examination (102/120)
	A.Y. 2014/15 One Year of study abroad in Chicago, Illinois
SCHOLARSHIPS	
Fall 2015	Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend
Spring 2015	Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend
Fall 2014	Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend
PUBLICATIONS	
2011	Pagano Francesco, and Pagano Davide. ”Using in-memory encrypted databases on the cloud.” In Securing Services on the Cloud (IWSSC), 2011 1st International Workshop on, pp. 30-37. IEEE, 2011.
2012	Damiani Ernesto, Francesco Pagano, and Davide Pagano. ”iPrivacy: a Distributed Approach to Privacy on the Cloud.” International Journal on Advances in Security 4.3 and 4 (2012): 185-197. (2012).

VITA (continued)

2015	Pagano Davide, Mikel Vuka, Marco Rabozzi, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. "Thermal-aware floorplanning for partially-reconfigurable FPGA-based systems." In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pp. 920-923. EDA Consortium, 2015.

WORK EXPERIENCE AND PROJECTS

Sep 2009 - Sep 2012	Stage at P&P Informatics as Software Engineer and Software Developer
2013	Interdisciplinary project, Integration of Genetic Association Database in Genomic and Proteomic Data Warehouse: Genomic and Proteomic Data Warehouse (GPDW) collects biomedical data distributed over a large number of databases and integrate them with each other, checking consistency, allowing scientist to perform cross-checking the information contained in different databases with a few simple steps.
2013	Information Systems: Aim of the project was to design the technological components of a system highlighting the architecture, the systems for the management of data and communication networks for the construction of an information system in a company. Use Case, ER Diagram, UX Model, Boundary-Control-Entity (BCE), Logic DB Diagram were produced.
2013	Software Engineering, Horse Fever: The project involved the design and implementation of a client and a server talking through Socket and RMI, where client players can use both a command line or a GUI to play the board game. Strong UML documentation was produced, including Class Diagrams, Use Case Diagrams, Sequence Diagrams.
2014	Software Engineering, Travel Dreams: The project involved the design and implementation of an e-commerce system to support its sale process. The system was developed on the JEE platform using JavaServer Faces (JSF), EclipseLink (JPA) and Glassfish. In particular, we used EJBs to develop the business logic with a web application user interface and MySQL to program the database. Strong UML documentation was produced including Requirements Analysis and Specification Document (RASD), Design document (DD), Function Point and COCOMO approach to estimate effort in the project.

VITA (continued)

2014	Statistical Natural Language Processing, Polarization of Internet slang words previously unseen in formal dictionaries: The success of the Internet and the increase of variety of communication forms has led to a language evolution and with that new words have been created. In this project, we polarized new terms coming from the Internet slang, in particular from the Urban Dictionary website. To reach our goal, different classifiers like Support Vector Machines, Naive Bayes, Maximum Entropy, Logistic Regression have been trained and ensembled to achieve better results.
2014	Data Mining, Kaggle Competition: MLSP 2014 Schizophrenia Classification Challenge: Development of a classification method to detect schizophrenia from data coming from brain scans of patients, exploiting the Python library scikit-learn and the Java tool Weka.
2014	Formal methods for concurrent and real-time systems: formal specification of hybrid car sharing system using logic formulas: Designed a modular TRIO specification of a parallel Hybrid Electric Vehicle (HEV) including safety requirements on braking performance and a specification for the car-sharing system for parallel HEVs, and verified them formally by using the automated tool Zot.
2014	High Performance Processor Systems: Thermal-Aware Floorplanning for Partially-Reconfigurable FPGA-based Systems: Field Programmable Gate Arrays (FPGAs) systems are being more and more frequent in high performance applications. Temperature affects both reliability and performance, therefore its optimization has become challenging for system designers. We developed a novel thermal aware floorplanner based on both Simulated Annealing (SA) and Mixed- Integer Linear Programming (MILP).
2014	Logic and Algebra: DiffieHellmanMerkle key exchange from the algebraic point of view: Analyzed the DiffieHellmanMerkle key exchange algorithm from the algebraic point of view of finite fields.
2015	Compiler Design, C- compiler: Implemented a compiler written in C++ able to generate LLVM IR code for programs written using the "C-" programming language. This included constructing the Abstract Syntax Tree and the Symbol Table after parsing the source code with the appropriate Bison grammar and Flex lexer.